Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN
Vakgroep Informatica
Laboratorium voor Programmeerkunde

Maintaining causality between design regularities and source code

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Andy Kellens

Academiejaar 2006 - 2007

Promotoren: Prof. Dr. Theo D'Hondt en Prof. Dr. Kim Mens

Nederlandstalige samenvatting

Als één van de technieken om de intrinsieke complexiteit van software systemen te trotseren introduceren programmeurs (bewust of onbewust) structurele broncode regulariteiten in een systeem. Deze regulariteiten beschrijven de verschillende eigenschappen waar de broncode van het systeem moet aan voldoen zoals bijvoorbeeld het correct gebruik van naam- en codeer-conventies, bepaalde afhankelijkheden in het ontwerp van het systeem, evenals het consistent gebruik van een wederkerend sjabloon in de implementatie van een bepaald concept.

Dit gebruik van regulariteiten heeft als doel het begrip van de broncode te vergemakkelijken, de communicatie van de intentie van een programmeur met de andere teamleden te bevorderen, een bepaalde beproefde oplossing te introduceren, enzovoort. Alhoewel regulariteiten in causaal verband staan met de broncode van een systeem is deze causale band echter impliciet en niet ondersteund door de programmeertaal. Als gevolg kan de evolutie van de broncode of de regulariteiten ervoor zorgen dat beide artefacten niet meer gesynchroniseerd zijn. Deze asynchronisatie kan dan ook leiden tot inconsistenties en fouten in de implementatie van een systeem.

In deze doctoraatsverhandeling stellen we het model van *intensional views en constraints* voor als een oplossing om het bovenstaande probleem te verhelpen. Dit model biedt programmeurs een conceptueel raamwerk om de diverse regulariteiten in een systeem expliciet te documenteren en eventuele afwijkingen tussen deze documentatie en de broncode te identificeren. Onze techniek berust op het gebruik van een classificatie-mechanisme (*intensional views*) om de entiteiten in de broncode te groeperen die bijdragen tot de implementatie van een bepaald concept in het systeem. De effectieve regulariteiten die deze concepten in de broncode reguleren worden gedocumenteerd door middel van verifieerbare *intensional constraints* te declareren over bovengenoemde *intensional views*.

Vanuit een methodologisch standpunt benaderen we de ontwikkeling van software vanuit een andere invalshoek door deze gedocumenteerde regulariteiten op te nemen als een integraal onderdeel van het ontwikkelingsproces. Als ondersteuning om de causaliteit tussen regulariteiten en broncode te behouden stellen we dan ook voor om de documentatie gecreëerd met intensional views en constraints te *co-ontwerpen* en *co-evolueren* met de broncode in het systeem. In plaats van de implementatie en de documentatie als twee afzonderlijke entiteiten te beschouwen resulteert deze methodologie in de gezamenlijke ontwikkeling van beide artefacten zodat deze op elkaar kunnen afgestemd worden en afwijkingen zo snel mogelijk geïdentificeerd kunnen worden gedurende het ontwikkelen.

Ter validatie van ons onderzoek voorzien we het *IntensiVE* ontwikkelingshulpmiddel. Dit hulpmiddel is een concrete implementatie van het model van intensional views voor de

VisualWorks Smalltalk omgeving en maakt het mogelijk om verifieerbare documentatie te creëren voor regulariteiten in Smalltalk en Java programma's. Eén van de peilers tijdens de ontwikkeling van deze tool was de integratie met de omliggende ontwikkelingsomgeving om een doorgedreven ondersteuning voor onze methodologie te voorzien.

Verder illustreren we in deze verhandeling hoe de bekomen resultaten kunnen aangewend worden voor het verhelpen van het fragiele pointcut probleem, een significant evolutieprobleem binnen het onderzoeksdomein van aspect-gericht programmeren. Dit probleem, dat gerelateerd is aan het behouden van de causaliteit tussen regulariteiten en broncode, wordt veroorzaakt door een sterke koppeling tussen zogenaamde pointcutexpressies en de structuur van de broncode van een systeem. We bieden een uitbreiding van het model van intensional views aan, namelijk *model-gebaseerde pointcuts*, die het mogelijk maakt om pointcutexpressies los te koppelen van de broncode van het systeem en uit te drukken in termen van een conceptueel model, geconstrueerd door middel van intensional views en intensional constraints. Op deze manier kunnen we ondersteuning aanbieden op het niveau van dit conceptueel model om de evolutieconflicten die resulteren in de fragiliteit van de pointcut op te vangen en te verhelpen.

Acknowledgments

I would have never been able to finish this dissertation without the tremendous support of a number of people.

First of all I would like to thank my promoter, Theo D'Hondt. I owe him my deepest gratitude for kindling an interest in research in me and for offering me the opportunity and freedom to pursue this interest. Although I delivered him the bulk of this thesis text over a very short period, he was still able to swiftly provide me with comments that opened up new perspectives and that drastically improved the quality of the text. Although it is totally unrelated to this dissertation, I would also like to thank him for introducing me to the books of Neal Stephenson.

I am also deeply indebted to Kim Mens, my co-promoter. Kim instigated the topic of this dissertation during an informal discussion we had late 2003. Ever since, he has been my "partner in crime" in our joint research on intensional views (and aspect mining). It will be very hard for me to repay him for the countless hours he spent on reading and discussing my text. Working together with Kim has been an awful lot of fun and I hope we can continue our cooperation in the future.

Also a big "thank you" needs to go to Johan Brichau for proofreading this dissertation. Especially his remarkable talent at playing devil's advocate has proven to be invaluable for improving this dissertation. Now that finally I have some time again, I can start fixing the bugs Johan has discovered while using IntensiVE :-)

I thank Yann-Gaël Guéhéneuc, Stéphane Ducasse, Ann Nowé, Geert-Jan Houben and Wolfgang De Meuter for finding the time to be on my thesis committee and for their insightful comments and suggestions.

I would like to thank my colleagues and ex-colleagues at PROG for providing a stimulating environment to work in. So thank you Adriaan Peeters, Bram Adams, Brecht Desmet, Charlotte Herzeel, Coen De Roover, Christian Devalez, Dirk Deridder, Dirk Van Deun, Elisa Gonzalez, Ellen Van Paesschen, Isabel Michiels, Jessie Dedecker, Johan Fabry, Jorge Vallejos, Kris Gybels, Kris De Schutter, Linda Dasseville, Pascal Costanza, Peter Ebraert, Sofie Goderis, Stijn Mostinckx, Stijn Timbermont, Thomas Cleenewerck, Tom Van Cutsem and Wolfgang De Meuter. Not only are you very fun people to work with, I most definitely also enjoyed the times we ended up at the KK or at the movies. Also many thanks to our secretaries, Lydie Seghers, Brigitte Beyens and Simonne De Schrijver, for their support in helping me deal with the intricacies of the university administration.

A special "thank you" needs to go to Bruno De Fraine for his expert knowledge of LaTeX and for helping me out when I ran into trouble typesetting this document.

I also want to thank my friends (I hope I don't forget anyone) Fré, Koen, "Clarry", Geert-Jan, Annick, Ans, Caroline, Serge and Nathalie for constantly enquiring how this dissertation was progressing and for occasionally providing me with a welcome distraction from my work.

Last but certainly not least, I wish to thank my parents for allowing me to obtain a higher education in the first place and for unconditionally supporting me.

The work reported on in this dissertation was funded by a PhD scholarship provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)".

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the techniques developers use to deal with the inherent complexity of software systems is to systematically introduce regularities in the source code of a system. These regularities express different conventions, idioms and patterns that are used to communicate the developers' design intent or to regulate the implementation of a particular concern by relying on a proven solution. However, since these regularities are neither explicitly documented in the source code nor are they first-class entities of the development process, the causal connection between the regularities and the implementation can be severed during development. Consequently, changes to either the regularities or the source code can result in mismatches which might lead to an inconsistent or incorrect implementation. In this dissertation we advance an approach that emphasizes this causality between regularities and source code by turning regularities into an integral and explicit part of the development process. We attain this integration by offering a formalism and a methodology which enables the co-design and co-evolution of regularities and the source code by making the regularities and the causal connection with the implementation explicit and verifiable. Furthermore, our approach strives at maintaining this causal link throughout the development cycle.

1.1 Research context

The term "design regularity" denominates a broad range of properties concerning the *structure* of the *source code* of a program. Perhaps the simplest and most common of such structural source-code regularities[1] are naming schemes and coding conventions. For example, a widely accepted naming convention in Java is prefixing the name of all *getter* methods with the string "get". Similarly, developers make rigorous use of programming language idioms [Cop92, Bec97], such as the implementation template for a "double-dispatch protocol" or the prototypical implementation of reference counting in C++, as a means of implementing particular recurring concepts in the source code.

[1]We use both terms "design regularities" and "structural source-code regularities" as synonyms throughout this dissertation. However, the former term reflects the use of regularities as verifiable design documentation while the latter term has the connotation of the technical use of regularities.

At a higher level of abstraction, developers use regularities such as object-oriented design patterns [GHJV95], design heuristics [Rie96] and architectural patterns [BMR$^+$96] to introduce well-known, proven solutions for given implementation problems. Furthermore, anti-patterns [BMT99, BMMM98] and bad smells [FBB$^+$99] are used to identify and to resolve often-recurring programming or design mistakes.

Developers consciously or unconsciously introduce a proliferation of such regularities into the source code as a means to deal with the intrinsic complexity of large software systems [Bro87]. These regularities serve as a governing principle [Min91] to regulate how particular domain-specific, application-specific or implementation-specific concerns are reflected in the implementation. In other words, these regularities are introduced with the goal of capturing a developer's design intent [HH06].

A first reason for introducing such regularities is to *convey this intent* between developers. A developer wants to make concepts that are implicit in the implementation clear to other developers. To achieve this communication, the developer will try to introduce a kind of "code pattern" in the source code that characterizes this concept and makes it explicit. For example, a developer can use intention-revealing names or a specific idiom when implementing a certain concept. Second, the meticulous application of these design regularities aids in obtaining more *uniform* source code. By adhering to the regularities that govern a system, stylistic conformance is obtained throughout the source code. This results in the source code becoming more *comprehensible* and *maintainable*. Third, developers often introduce regularities such as idioms and design patterns to provide a *proven, recurrent solution* for a particular problem in the implementation or design of a system.

Furthermore the *correct functioning* of the system can sometimes depend on whether or not developers correctly adhere to certain regularities in the implementation. For example, object-oriented frameworks [JF88] can impose a number of regularities that instantiations of the framework must adhere to in order to function correctly. One example of such a regularity – as described by Johnson [Joh92] - stems from the HotDraw framework [Bra92] for graphical editors. Without going into detail, when a developer customizes this framework by creating a new composite figure, the developer *must* provide an initialize method that adds the components of the composite to the figure.

Analogously, aspect-oriented programming (AOP)[KLM$^+$97] relies on compliance of the source code with certain regularities that govern that system. Aspect-orientation offers novel language constructs for modularizing so-called crosscutting concerns [TOHS99a], i.e. concerns that do not align with the decomposition of the system. An aspects consist of an *advice*, that implements the crosscutting behavior and a *pointcut description* that selects the events during the execution of a program which this advice is invoked at. To select these events, such pointcuts rely heavily on how a certain concept in the source code is structured [KMBG06]. Consequently, if a developer deviates from the regularities that govern this concept, this can lead to erratic behavior of the aspects [KS04, SG05].

The above examples illustrate some of the applications of regularities. Independently of whether these regularities are used to communicate a developer's design intent, to improve comprehensibility of the source code, as a recurrent solution to a particular implementation problem or to compensate for the lack of proper abstraction mechanisms, regularities encode knowledge about how certain concepts are assumed to be implemented. Consequently, these regularities are causally linked with how the concepts are actually manifested in the source code. To obtain a consistent implementation of these regularities, it is important that this causal connection between the regularities and the source code is maintained during development.

1.2 Problem statement

Maintaining the causality between regularities and source code is not a trivial task. The different regularities that govern a system are only implicitly available in the source code. Consequently, it is not guaranteed that when changes are made to the source code of a system or when regularities that govern the system are altered, that the regularities and the implementation remain synchronized. In other words, during the development process of a system the causal connection between regularities and source code can be severed.

While developers can alleviate this problem by *manually* maintaining the regularities in a system, this tends to be quite tedious and error-prone. It requires a developer to meticulously adhere to the multitude of regularities that govern a system. Upon evolution of a system a developer needs to ensure that the different regularities remain respected in the implementation or that changes in the regularities are correctly propagated in the source code.

Consequently, there is a need for approaches that support these regularities during the development of a system. Such an approach must provide facilities for turning the regularities that are *implicitly* present throughout the implementation of a system *explicit* to the developers. Furthermore, this approach must also turn the causal link between the regularities and the source code explicit and provide a means to maintain this causality upon evolution. In order to sustain this causal link, the approach must make it possible to verify the validity of the regularities with respect to the source code and provide support for resolving any discrepancies that result from evolution of the system (i.e. of the source code or its regularities).

In literature, we can find a substantial body of research that has been devoted to alleviating the aforementioned problem. Although a proliferation of approaches exists, we can distinguish between three different categories:

- **Code checkers**: Approaches such as Lint [Joh79], LCLint [EGHYM94], CheckStyle [Che06], FindBugs [HP04], and many others offer tool support for detecting a wide range of frequently-made errors, bad coding style, violations of platform-specific requirements and so on. While these tools offer a convenient way to verify such regularities, they are designed to support a particular kind of regularity. Consequently, they are less suited for supporting e.g. domain-specific or application-specific regularities;

- **Meta-programming systems**: Meta-programming systems such as for example SOUL [Wuy01], CCEL [DMR92] and Law-governed systems [Min96] offer developers a dedicated language for expressing meta-programs that reason about the source code of a system. These languages can be applied to implement "checkers" that verify the consistency of particular regularities with respect to the source code. While these languages are sufficiently generic to implement checkers for a wide variety of regularities, these meta-programs are often created in an ad-hoc fashion and do not explicitly document the regularities;

- **Architectural and design conformance checkers**: Architectural and design conformance checkers provide a *dedicated* means to express a high-level description of a software system (i.e. design patterns, architectural patterns, ...) and identify discrepancies between this description and the implementation of a system. Examples of such approaches are Reflexion Models [MNS95], Ptidej [Gu2] and virtual software classifications [Men00]. Similarly to code checkers, these approach emphasize the support for particular kinds of regularities only.

1.3 Research goal

In this dissertation, we advance a novel approach for maintaining the causal link between regularities and the implementation of a system. Our approach aims at maintaining this causality by turning the implicit regularities into an integral, first-class part of the development process. We attain this integration by supporting the co-design and co-evolution of regularities and source code:

- **co-design**: The term co-design stems from engineering where the design of an artifact is often subject to multiple perspectives and the design process needs to take all of these perspectives equally into account. A typical example of co-design is the development of mobile phones, in which both the perspective of hardware and software must be considered. The hardware and software are not independently developed, but the development of both artifacts is tightly coupled and causally connected. Co-design between the regularities and the source code implies that we do not consider the regularities subordinate to the source code but rather treat them as equal to this source code. We propose a scheme in which the regularities are turned into an explicit entity of the development process and this entity is developed in unison with the source code;

- **co-evolution**: Evolution of the system can result in changes to both the regularities and the source code. Upon evolution, it is imperative that these changes do not break the causality between both artifacts. Similar to co-design, we cannot consider the evolution of either regularities or source code in isolation. Instead, regularities and source code must co-evolve [DDMW00], i.e. evolve simultaneously. Changes to the system can result in an interplay between regularities and source code in which both have to be updated.

Our approach realizes this support for regularities by:

- Introducing a formalism for documenting regularities. This formalism is sufficiently expressive to create verifiable documentation for a wide range of different kinds of regularities. As such, this documentation serves as a means to make the regularities and the causal connection with the source code *explicit*;

- Proposing a lightweight methodology that complements our formalism. This methodology presents a set of guidelines that offer a structured way of documenting regularities using our formalism. Furthermore, the methodology incorporates the co-design of regularities and source code in the development cycle by describing a process in which the documented regularities and the source code are developed and refined in unison. Analogously, our methodology supports the co-evolution of regularities and implementation by advocating a test-often philosophy in which infringements of the causal link between regularities and source code are identified and resolved early on during the development process.

1.4 Blueprint of our approach

In what follows, we give a brief overview of our approach. First, we explain the conceptual contribution of our work, namely co-designing and co-evolving regularities and source code as a means to maintain the causal link between both artifacts. Second, we discuss the model of intensional views, the formalism we propose as a means for creating explicit, verifiable documentation of regularities. This model provides a technical platform for supporting co-design and co-evolution. Third, we briefly discuss our lightweight methodology. Fourth, we provide a description of IntensiVE – the concrete instantiation of our model of intensional views and constraints – and relate how this tool suite supports our methodology. We conclude this section by discussing how an extension of the approach we propose in this dissertation can serve as a means to deal with the fragile pointcut problem, one of the open evolution problems within aspect-oriented programming.

1.4.1 Co-design and co-evolution of regularities and source code

At a conceptual level, our approach advances a new metaphor for developing software that is focussed around the concepts of co-design and co-evolution as a means to maintain the causality between regularities and source code. To this end, we aim at incorporating first-class, explicit documentation of the different regularities that govern a system into the development process. This active documentation provides a verifiable representation of the causal link between the regularities that are present in a system and the actual manifestation of these regularities in the source code of the system. By co-designing and co-evolving this documentation with the actual implementation of the system, both artifacts can be tailored towards each other throughout the development process.

By supporting co-design and co-evolution during the development or evolution of the system, our approach takes into account that changes to the source code can also impact the documented regularities and vice versa. Since the causal link between these regularities and the source code is explicit and verifiable, this causal link can aid in maintaining consistency between regularities and source code whenever changes are made to the system. For instance, if the source code of the system is altered, the active documentation can be used to verify whether these changes respect the different regularities governing that part of the system. Analogously, if the documented regularities are changed, this active link can aid in updating the source code in order to correctly reflect these changed regularities. As such, this interplay between designing and evolving regularities and source code in unison supports maintaining the causality between both artifacts.

The nature of this documented and verifiable causal link can be both descriptive as well as prescriptive. During the earlier phases of the implementation cycle, the documented regularities as well as the source code can be subjective to quite a lot of changes. In such a situation we envision a descriptive use of the documented causal link. During the development, both the documentation as well as the source code are updated and geared towards each other as a means to iteratively refine the source code and the documented regularities. Over time, as the documentation for the different regularities becomes more stable, this documentation can be used in a prescriptive way. Upon the application of changes to the source code, the documentation serves to verify that these changes do not violate any of the documented regularities.

1.4.2 Intensional views and constraints

As a technical means to express the causal link between regularities and source code, we propose the model of intensional views and constraints. This model lies at the core of our approach and offers a means for creating verifiable and explicit documentation for regularities. The formalism of intensional views and constraints makes an explicit distinction between the implementation of a certain concept that is governed by a set of regularities and the actual regularities themselves.

Intensional views In our approach, we document the source-code entities implementing a concept which a regularity is applicable to using an *intensional view*. Such an intensional view offers a classification mechanism that groups the source-code entities that belong to the implementation of that concept. An intensional view can consist of a set of classes, methods, variables, packages, and so on. For example, if a developer wishes to document the naming convention that in Java "getter methods", i.e. methods that provide access to a field, should be prefixed "get-", an intensional view *getter methods* must be created that groups these getter methods.

A key characteristic of intensional views is that the set of source-code entities belonging to the view is not specified by manually enumerating these entities. Instead, an intensional view is defined by means of an executable description which, upon execution, yields the set of source-code entities that belong to the intensional view. For example, a possible executable description for the *getter methods* intensional view we discussed above is to retrieve all methods that consist of a single expression that returns the value of a field, which is a prototypical implementation of a getter method.

This means of defining an intensional view is an enabling factor for supporting evolution of the regularities and the source code. When the system evolves, the executable description makes it possible to automatically classify source-code entities that belong to a particular intensional view. For instance, if a getter method would be added to the system that adheres to the executable description, this method would be classified by the *getter methods* intensional view without having to update this intensional view manually.

Intensional constraints While an intensional view creates explicit documentation for the concept which a regularity is applicable to, this actual regularity is captured by means of *intensional constraints*. Such intensional constraints can be imposed on one or more intensional views and implement a verifiable condition that is applicable to the entities belonging to an intensional view. To illustrate this concept, consider again the example of the naming scheme for getter methods. In order to document that the name of all these accessor methods should be prefixed "get-", we impose an intensional constraint over the intensional view that states that *for all* methods in the *getter methods* view, the name of the method must be prefixed with the string "get".

Supporting the causal link between regularities and source code Intensional views and constraints provide a sufficiently expressive medium for documenting various kinds of regularities ranging from naming and coding conventions over programming idioms to regularities at a higher level of abstraction such as the interactions between different class hierarchies, and so on. Our formalism aids in maintaining the causal link between regularities and source code by explicitly documenting both the source-code entities that are governed by a regularity as well as the actual regularity itself. Furthermore, we provide *active* documentation of the causal link: our documentation is expressed in terms of the actual source-code entities in the system. Moreover, the documentation we create is verifiable with respect to the source code. Due to the descriptive definition of intensional views, this causal link can also be verified whenever the system evolves.

1.4.3 Lightweight methodology

We complement our formalism of intensional views and constraints with a lightweight methodology. This methodology proposes a number of practical guidelines concerning the definition of intensional views and the selection and implementation of the correct kind of intensional constraints offered by our formalism. Furthermore, the methodology also describes how the documentation created using intensional views and constraints can be integrated into the development process to support the co-design and co-evolution of the regularities and the source code.

Our methodology supports the co-design of regularities and source code by describing a step-wise, iterative process for refining the documented regularities and the implementation of a system. A developer documents the regularities that govern a system using intensional views and constraints, thus making them explicit and verifiable. By iterating between the verification of this explicit documentation and the gradual refinement of *both* the regularities and the source code, the regularities and source code are simultaneously developed. Consequently, the causal link between regularities and source code is turned into an integral part of the development cycle.

Our methodology proposes a test-often philosophy to support the co-evolution of the documented regularities and the source code, i.e. to maintain the causal link between both artifacts. The verification of the causal link between regularities and source becomes a part of the standard testing cycle of the developer. The goal of this test-driven verification is to identify discrepancies between regularities and source code as soon as possible at development time.

When discrepancies are detected, a developer harmonizes the regularities and the implementation by studying the source code related to the discrepancies as well as the documented regularities and refining them in unison.

1.4.4 IntensiVE

As a concrete instantiation of the model of intensional views and constraints we have developed the *Intensional Views Environment*, or *IntensiVE* for short. This research prototype is devised as an extension to the Cincom VisualWorks Smalltalk development environment [Cin07]. The tool suite consists of a number of sub-tools that enable the definition and manipulation of intensional views and constraints over Smalltalk and Java programs. Our tool suite supports Smalltalk and the declarative meta-programming language SOUL [Wuy01] as the languages which can be used to specify the executable description of intensional views. Furthermore, IntensiVE provides support for verifying the intensional views and constraints with the source code and provides detailed feedback concerning discrepancies.

IntensiVE was devised with support for co-design and co-evolution in mind. This support is obtained by tightly integrating the tool suite with the surrounding development environment. The intensional views and constraints defined on a system are first-class entities in the development environment that can be accessed by other tools in the environment. Furthermore, in order to stimulate the co-design of the regularities and the source code, IntensiVE's tight integration with the development environment makes it possible to browse and adapt the source code related to the entities belonging to intensional views as well as entities that are reported as discrepancies directly from within the tool suite.

The IntensiVE tool suite supports the test-often philosophy proposed in our methodology by integrating with the unit testing framework of VisualWorks. Consequently, the set of intensional views and constraints can be verified simultaneously with the unit tests that are defined on a system. The documented regularities can thus be considered structural regularity tests. The combination of this test-driven verification and the aforementioned integration of IntensiVE with the VisualWorks development environment results in the tool suite supporting the co-evolution of the regularities and the source code.

1.4.5 Model-based pointcuts

The fragile pointcut problem, one of the open evolution problems within the field of aspect-oriented programming, presents a particular instantiation of the problem of maintaining causality between regularities and source code. The quantification mechanism employed by aspect-oriented pointcut languages introduces a tight coupling between pointcut expressions and how the source code of a base program is structured [KMBG06]. Consequently, seemingly safe modifications to the base program can result in erratic behavior of the aspects imposed on this program [KS04, SG05].

We propose to alleviate this fragile pointcut problem by means of *model-based pointcuts*. This model-based pointcut mechanism strives to decouple the actual pointcut definition from the implementation structure of the concepts in the source code which the pointcut relies on as well as to render the causal link between these concepts and the implementation structure explicit and verifiable. As such, these model-based pointcuts can be considered to be a particular instantiation of the general scheme of co-design and co-evolution we propose in this dissertation. By designing and evolving the explicit assumptions pointcut developers make about the base program in unison with the source code of this base program we alleviate the fragile pointcut problem.

These model-based pointcuts are also supported by our formalism of intensional views and constraints and the IntensiVE tool suite. In particular, we provide an extension of the CARMA pointcut language [GB03] in which pointcuts can be expressed in terms of intensional views representing different concepts in the implementation of the base program. By imposing intensional constraints over these views, this instantiation of model-based pointcuts renders the causal link between the source code and the concepts explicit and verifiable and aids in maintain causality upon evolution of the source code.

1.5 Contributions

This dissertation presents the following main contributions:

- An approach that supports the causal link between regularities and source code by considering the regularities an integral part of the development process and supporting the co-design and co-evolution of regularities and implementation;
- The model of intensional views and constraints as a formalism that offers a structured way of documenting a wide variety of regularities that govern application-specific, implementation-specific or domain-specific concepts. This documentation renders both the regularities as well as the causal link with the implementation explicit and verifiable;
- A lightweight methodology that complements the model of intensional views and that describes a set of guidelines for documenting regularities using our formalism. This methodology advocates the co-design and co-evolution of regularities and source code;
- The IntensiVE tool suite that provides a concrete instantiation of our model of intensional views. By integrating tightly with the surrounding development environment and with the unit testing framework, this prototype tool suite puts our model and methodology into practice;
- The approach we advance in this dissertation is sufficiently general to support other evolution problems related to maintaining causality between regularities and source code. As a concrete example of this generality, we offer *model-based pointcuts* as a technique to alleviate the *fragile pointcut problem*, one of the open evolution problems within aspect-oriented research.

1.6 Overview of the dissertation

This dissertation is structured as follows:

Chapter 2 We start this dissertation by taking an in-depth look at the concept of structural source-code regularities and discuss a number of properties that characterize these regularities. In this chapter we also provide a literature study of related work by surveying other approaches that support the documentation and verification of structural source-code regularities.

Chapter 3 In this chapter we introduce our model of intensional views and constraints as a formalism that can be used to create verifiable documentation of regularities. We introduce this contribution independently of any implementation language by providing a formal specification – inspired by relational tuple calculus – of the different concepts that are part of our model.

Chapter 4 After having introduced our formal model of intensional views and constraints, we present one concrete instantiation of our model, namely the IntensiVE tool suite. This is our research prototype that is implemented in VisualWorks Smalltalk and that tightly integrates with this development environment. In chapter 4 we also introduce our methodology. This methodology describes how the formalism of intensional views and our tool suite can be put into practice and how they support co-design and co-evolution of regularities and source code. We conclude this chapter by providing a comparison of our work with the approaches we discussed in Chapter 2.

Chapter 5 As a validation the expressiveness of our approach, we demonstrate how our methodology and tool suite can be applied in order to create verifiable documentation of a wide variety of regularities. In particular, we document the various regularities underlying the implementation of nine concrete instantiations of object-oriented design patterns.

Chapter 6 We perform an experiment in which, following our methodology, we document the regularities underlying the implementation of three software systems in order to demonstrate how our approach supports the co-design and co-evolution of regularities and the source code. We assess the impact of evolution on these three systems and discuss how our approach aids in maintaining the causality between the regularities and the source code.

Chapter 7 Our approach is sufficiently general to support other problems within software evolution. We demonstrate the applicability of our approach for alleviating one such evolution problem, namely the fragile pointcut problem. We introduce an extension to aspect-oriented programming, namely *model-based pointcuts*, that builds on our model of intensional views and the IntensiVE tool suite to alleviate the fragile pointcut problem.

Chapter 8 To conclude this dissertation we discuss some of the limitations of our approach and implementation and hint at how we can overcome these limitations. We also propose a number of directions of future research.

Chapter 2

Existence and support for structural source-code regularities

This chapter consists of three parts:

- In Section 2.1 we characterize the concept of structural source-code regularities. We provide a definition for such regularities, discuss a number of their properties and propose an initial taxonomy of the different kinds of regularities. We conclude this section with an overview of the requirements that an approach for supporting the use of structural source-code regularities throughout the development process must fulfill;
- Section 2.2 discusses some related approaches that propose the use of a classification mechanism to group conceptually related software entities. Since a similar use of a classification mechanism lies at the heart of our approach, we give an overview of these approaches;
- We finish this preliminary chapter in Section 2.3 with an overview of the state of the art in approaches which support the use of structural source-code regularities in the development process.

The goals of this chapter are two-fold. The first goal is to establish a common terminology which will be used throughout this dissertation. The second goal is to provide an overview of existing literature related to our work, such that we can position us in the domain.

2.1 Structural Source-code Regularities

Structural source-code regularities is a general term that encompasses the different conventions and patterns that govern the source code of a system. Structural source-code regularities range from low-level conventions like naming conventions and coding conventions, over the idiomatic implementation of certain concepts to patterns at the design or architectural level of a software system. For example, conventions like e.g. “all classes in the hierarchy of class `Command` must have a name starting with prefix *Command*”, “accessor methods must all be implemented according to the same implementation idiom", "the entities in the presentation layer are not allowed to refer to entities in the database layer" are considered to be structural source-code regularities. In what follows we give a more precise definition of the term along with a number of examples. Moreover, we discuss a number of properties of structural source-code regularities.

2.1.1 Definition

In this section we informally define the concept of structural source-code regularities. The concept of regularities in software is not novel. Minsky defines a regularity in his work on law-governed architecture [Min96] as:

> "any global property of a system; that is, a property that holds true for every part of the system, or for some significant and well defined subset of its parts ".

This definition of a regularity is quite general and encompasses numerous properties which should be upheld in a software system. Such regularities are not limited to invariants in the source code of a system but can also dictate how objects must interact at run-time, how different threads need to be synchronized and so on. Regularities are also not limited to the implementation but can also exist in artifacts at different stages of the development process, can express interactions between artifacts at those different stages or can even guide the development process itself by governing how the system may be changed by a developer.

Following his definition, Minsky requires a regularity to be a quantified property over the software system. For instance, according to this definition, the property 'class `B` must implement a method named `x`' is not a regularity, because it is limited to a single artifact in the software. As such, it does not fulfill the property of being valid for a 'significant' subset of the system, i.e. it does not make a quantified statement about the system. However, 'all subclasses of class `B` must implement a method named `x`' is considered to be a regularity since it is quantified over a significant and well-defined subset of the system, namely the property must hold for *all* subclasses of a class `B`.

Minsky's definition of a regularity serves as an informal, intuitive way to describe regularities. However it leaves room for discussion whether certain properties are considered to be a regularity or not. Especially quantifying when a part of the system is significant enough such that a property is considered to be a regularity heavily depends on an observer of the system. Moreover, this definition suggests that regularities need to be upheld at any given moment in time. In practice however, this does not always seem feasible as regularities often are violated while a developer is changing the system, or they may even evolve themselves as the system evolves. As such, we feel that the definition is too restrictive in this regard.

Although remaining informal as well, in our work we take a more pragmatic stance when defining regularities. In particular, we focus throughout this dissertation on a specific kind of regularities, namely *structural source-code regularities*. We define a structural source-code regularity as:

Definition 1. *"any decidable property of the structure of the source code of a software system which must remain true after evolution of the system".*

A structural source-code regularity is a predicate over the source code of a system expressing an evolution invariant on the structure of that source code. Note that in our definition we define structural regularities as properties of the *structure* of the source code of a certain system. As such, we do not limit ourselves to reasoning about the source code at a lexical level (e.g. a string representation of the code), but rather assume the presence of a structured representation of the source code, for instance of the form of an abstract syntax tree, which we define invariants over.

In a sense, our definition of regularities is more general than Minsky's. First of all, we do not restrict the scope of regularities to properties which hold only for an entire system or a – by the developer deemed significant – subset of the system. We do not require a regularity to be quantified over the entire system. Although the majority of the regularities we discuss throughout this dissertation are applicable to a large part of the system, we do not wish to exclude regularities which are local to a very small part of the source code.

Second, our definition explicitly takes evolution of the software system into account. We do not require that a regularity is valid at all times. For instance, *during* the process of changing the source code of a system, not all structural source-code regularities necessarily have to remain true. However, we state that regularities must be upheld *after* an evolution of the system, i.e. after a set of changes to the source code has been applied.

Note that we take a broad view of evolution. We do not restrict evolution to small changes to the source code of a system such as for instance the application of a refactoring, but we also consider more intrusive alterations of the system to be covered by the term evolution. Moreover, evolution of a software system is not limited to changes in the *source code* of a system. Over time, the other artifacts associated with a software system can also evolve. For instance, the documentation of the system might also have to be updated due to changes in the requirements, the design, the structural source-code regularities, and so on.

Structural source-code regularities is a broad term that captures a large number of properties of the source code ranging from coding conventions, naming conventions and programming idioms over dependencies between entities in the implementation of design patterns to high-level regularities at an architectural level. In general, structural source-code regularities express a constraint which is applicable to part of the source code of a system and which governs how that part of the source code is structured. As such, they describe in a broad sense the conventions which the developers of a system must adhere to.

2.1.2 Properties of structural source-code regularities

In this section we discuss a number of properties of structural source-code regularities. While it is by no means our goal to provide a complete overview of every kind of regularity, we focus on a number of dimensions in the problem space of structural source-code regularities. We illustrate these properties by means of a number of examples of structural source-code regularities or of groups of regularities.

Scope of structural source-code regularities

The scope of a structural source-code regularity is the extent in the source code which the regularity is applicable to. This can range from *global* regularities which are applicable to the entire system, to *local* regularities which are restricted to e.g. a single class hierarchy or a small group of software entities.

- **Global:** such regularities describe a constraint which is system-wide. It must be respected by all entities of a certain kind, such as for instance all classes, all methods, and so on, in the source code. Examples of global structural source-code regularities can typically be found in domains like best practice patterns [Bec97], anti-patterns [BMT99, BMMM98], design heuristics [Rie96] and bad smells [FBB$^+$99]. One example of a global structural source-code regularity is that in C++ programs, developers indicate a private field by prefixing it with an underscore. This property is applicable to all private fields in the software system and is thus considered a global regularity;
- **Medium:** Certain regularities are not applicable to the entire system but rather apply to a large part of a system and stretch across a number of class hierarchies or modules. One group of regularities which have a medium scope are the regularities which arise from the use of design patterns. For instance, the Visitor design pattern [GHJV95] is governed by a number of regularities which dictate how entities from two different class hierarchies must interact;
- **Local:** local structural source-code regularities are confined to a single class hierarchy or a small set of source-code entities. For example, a regularity which expresses that all methods in a certain class hierarchy must be implemented using a certain programming idiom, is considered to be local.

The scope of a regularity is not an ideal criterion to discriminate between multiple kinds of structural source-code regularities. First of all, it heavily depends on a developer's interpretation of how to decide whether a certain regularity is considered to be global or local. Second, not all instances of one particular kind of structural source-code regularities share the same scope. As an example, consider naming conventions: while there exist naming conventions which are applicable to the entire system, naming conventions are also often used to regulate the usage of e.g. method names in a single class hierarchy or a single module.

The scope of structural source-code regularities however learns us that regularities are inherently crosscutting. In general, structural source-code regularities describe a constraint over multiple artifacts in the source code, often orthogonal to the decomposition of a system.

Abstraction level in the implementation

Another property of structural source-code regularities is that they are generally prevalent at a particular abstraction level in the implementation. Similar to the work of Buschmann et al. [BMR$^+$96] on pattern-oriented software architecture, we distinguish between three abstraction levels which regularities can occur at:

- **architectural level:** regularities which express how the different subsystems of a system are related or may interact. One example of such a regularity is that of a layered architecture, in which an entity X in one layer can only communicate with an entity Y if Y is in a layer directly above or below the layer of X;

- **design level:** regularities which encode how the different entities within a subsystem should interact. Typical examples of such regularities are the different conventions and dependencies underlying the implementation of design patterns;

- **language level:** low-level regularities expressing a regularity in a particular part of a subsystem. Examples of these regularities are programming idioms such as the correct use of a double dispatch protocol, accessors for fields, ...

While the above abstraction levels are often associated with other software artifacts such as for instance UML diagrams which express the design of a system or the description of the architecture of a system using an architectural description language, we restrict ourselves solely to the source code of a system. As such, the abstraction level of a structural source-code regularity expresses the level of abstraction at which a regularity manifests itself in the *source code* of a system.

Although it might seem at first sight that the scope of a regularity and its abstraction-level are correlated, this is not always the case. While regularities at the architectural level tend to be global, this does not imply that structural regularities at the language level are local. For instance, naming conventions, which are typically found at the language level, can describe a constraint which is globally applicable as well as dictate how program entities in a specific class hierarchy should be named. Similarly, regularities expressing the use of programming idioms can govern a global property (e.g. all field accesses must happen using a getter method) as well as specify the usage of a specific instantiation of an idiom (e.g. all "accept methods" in an instantiation of the *Visitor* design pattern [GHJV95] must be implemented by a double dispatch protocol).

Applicability of structural source-code regularities

Structural source-code regularities can be characterized according to their applicability. While certain regularities are quite universal and can be applied to almost any software system, there also exist regularities the applicability of which is limited to a given development team or one specific application or domain.

- **Environment-specific regularities:** group constraints which are applicable to a wide range of software systems, independent of a single development team or the application-domain of the system. They generally encode knowledge concerning frequently occurring errors, language idioms, and so on. While they are not always applicable, their applicability is generally restricted to a certain implementation language, programming paradigm or implementation platform;

- **Team-specific regularities:** these regularities express the coding conventions and naming conventions used by a specific developer or team of developers;

- **Application-specific regularities:** describe the dependencies, conventions, inheritance constraints, ... typical for a singular system or a particular application domain. Such regularities describe the constraints which must be adhered to when implementing an application-specific or domain-specific concern in a certain software system. For instance, a set of regularities which express that in a graphical framework the actions for drawing figures need to be implemented by means of the Command design pattern, are considered to be application-specific regularities.

Note that the applicability of a regularity is strongly correlated with how often it is reused throughout different software projects. Environment-specific regularities often only depend on the implementation language used in a project or the platform which the application must run on. Therefore, throughout numerous projects, often the same set of structural source-code regularities will occur. Team-specific regularities represent the coding style and good practice patterns a team of software developers should adhere to. As such, they too are often reusable over multiple projects. Structural source-code regularities which dictate how certain concerns in a specific application should be structured or how they relate, encode knowledge which usually is not portable to other projects.

Note however that even application-specific regularities often share large similarities throughout different software systems. Although the same instance of a regularity is not applicable to two software systems, these similarities between structural source-code regularities can lie at the basis of a kind of "pattern language" which describes how certain types of regularities can be documented.

2.1.3 Taxonomy of structural source-code regularities based on functionality

The properties of structural source-code regularities we discussed in the section above provide a characterization of the different kinds of regularities. In this section we discuss a taxonomy of structural source-code regularities found in literature based on the function the regularities serve in the development process. This taxonomy was presented by Chowdhurry et al. [CM93] and was later also adopted by Bokowski [Bok99]. Although in their work they use the term constraints over source code, this taxonomy is equally applicable to structural source-code regularities.

This taxonomy divides regularities into three categories: stylistic regularities, implementation regularities and design-level regularities:

- **stylistic regularities:** this category encompasses all regularities which are concerned with the use of *names* in the system or any other constraint that is semantics-neutral and governs a syntactical property of the source code. Such regularities are used to make programs more readable by enforcing a uniform look and coding style;

- **implementation regularities:** this category consists of regularities which are used to detect infractions against frequently occurring bugs, pitfalls and programming errors. For instance, the regularity that all classes in Smalltalk which implement the = operation must also understand a message `hash`, is an implementation regularity. Other examples of implementation regularities are for instance the constraints which must be adhered to for an application to run in a specific environment;

- **design-level regularities:** rules governing the correct use of class hierarchies, components, frameworks, design patterns, ...

Design-level regularities are further subdivided into coding conventions, inheritance constraints and usage constraints:

- **coding conventions:** regularities which *structurally* ease readability and maintainability of the source code. For instance, a possible coding convention in Smalltalk is that all accesses to instance variables must occur via a getter method;
- **inheritance constraints:** these regularities express contracts between a class and its subclasses. E.g. a constraint like "all subclasses of `ClassA` must override a method `methodA`";
- **usage constraints:** such regularities express how certain classes or collections of classes are to be used. E.g. in an instantiation of the Factory design pattern [GHJV95] all clients must use the factory to create a product.

2.1.4 Importance of regularities in the software development process

In his seminal paper "No silver bullet" [Bro87], Brooks states that software development is inherently hard due to the complex nature of software systems and the lack of unified engineering principles which are prevalent throughout other engineering disciplines. As Minsky observed [Min96], the proper and meticulous use of regularities in software systems can be considered as a kind of engineering principle which aids in dealing with the complexity of software engineering.

Regularities can be considered to be a kind of pattern which is present in the source code of a system and which describes how a certain concept, functionality or concern in the system is represented in the source code. One motivation for introducing such regularities in the source code is to improve the communication of the intention of one developer to another. Rigorous use of naming conventions and other stylistic regularities to create a uniform coding style makes the source code of the system more readable [Bec97]. When a certain concern is characterized by a number of regularities, and these regularities are upheld throughout the implementation of the concern, it becomes easier for a developer to identify and understand the implementation of the concern in the source code.

However, as we discussed earlier, not all regularities are stylistic in nature. During the development of a software system, certain constraints, or regularities, must be upheld in order for the system to behave properly. Especially in libraries, frameworks, ... it is imperative that certain regularities are not violated in order for the system not to show incorrect behavior [Bok99, Min91]. For instance, in a framework, certain usage constraints and inheritance regularities govern the correct instantiation of the framework. If these regularities are not upheld, an instantiation of the framework might behave erratically. Similarly, certain development platforms like J2EE impose different regularities to be respected in software systems in order to prevent erratic behavior at runtime [EMS$^+$04].

With the advent of aspect-oriented software development, this need for consistent use of regularities becomes even more important. Due to the nature of the quantification mechanisms employed by pointcut languages, developers of aspect-oriented programs often rely on the fact that certain concepts in the system are characterized by stylistic or structural constraints. For instance, an aspect developer who wishes to capture the concept of a getter-method might assume that all getter-methods are implemented by the Java convention in which the name of the method starts with the prefix `get-`. In such cases, the proper functioning of the application relies on this naming convention. If a developer does not respect such a stylistic regularity, this can have an impact on the software behaving correctly. This fragility of aspect-oriented programs with respect to seemingly safe modifications of the base program has been dubbed the *fragile pointcut problem* [KS04, SG05, KMBG06].

2.1.5 Supporting structural source-code regularities during software development

A major drawback of regularities is that they are not an integral part of the programming language. Although programming languages impose a number of constraints on programs by means of syntax and semantics, the majority of regularities are only implicitly present in the implementation. While structural source-code regularities can appear in the source code in an ad-hoc way, i.e. developers unknowingly introduce them as a means to make their source code more readable, often teams of developers regulate the use of certain conventions and constraints which need to be upheld in the implementation.

However, due to their crosscutting nature and the fact that development environments typically do not provide support for documenting and enforcing them, over time the consistency and coherent use of structural source-code regularities can diminish. Changes in the source code can result in the regularity no longer being respected. This can lead to source code becoming more difficult to read and bugs being introduced if the violated regularities have an impact on the correct functioning of the system.

It is thus important that correct and meticulous use of structural source-code regularities is supported during the development of the system. An approach for maintaining the causal connection between regularities and source code must support the following functionality:

- **Explicit documentation:** A key part of supporting structural source-code regularities lies in their being explicitly documented. This documentation makes it possible to make the implicitly-available regularities in the source code *explicit* to developers and turns the regularities into *first-class artifacts* in the development process. This explicit documentation serves as a means to communicate the different structural source-code regularities underlying a system between different developers;

- **Verification of regularities:** Although documenting the regularities aids in making developers aware of their existence, this alone does not suffice to ensure that the regularities are properly adhered to throughout the implementation. In addition, support is needed to verify the validity of the implementation of a system with respect to structural source-code regularities. For an approach to be able to support the verification of regularities imposes a number of requirements:

  - **Connection with source code:** A causal link must exist between the documentation of regularities and the implementation. Documentation must be expressed in terms of actual source-code entities like classes, methods, fields, and so on. If such a link does not exist between the documentation and the source code, it is not possible to verify the consistency of the documentation with respect to the source code;

  - **Verifiability:** Since the documentation of regularities needs to be verifiable, a natural language description of the regularities does not suffice. What's more, the medium used to express the regularities must be rich enough to capture a wide range of regularities. Preferably, the same approach can be used to capture naming conventions and coding conventions as well as programming idioms, architectural regularities, design dependencies, . . . ;

- **Support for co-design and co-evolution:** The terms *co-design* and *co-evolution* [DDMW00] has been used in literature to indicate the situation in which a number of (software) artifacts coexist that are causally linked and in which changes to one of the artifacts have as a direct result that the other artifacts might need to be updated as well in order to keep all of the artifacts synchronized. In other words, the development and evolution of any particular artifact cannot be considered in isolation from the other artifacts.

  In the case of supporting structural source-code regularities during the development process, we encounter a similar situation. The regularities that govern the implementation of a system are causally linked to the source code of that system. While the regularities express how certain concepts in the source code are supposed to be structured, the source code contains the actual manifestation of these regularities. Consequently, this causal link must be maintained during the development of the system.

  First of all, the documentation and verification of the regularities must not be limited to a single version of the system. During evolution, it must remain possible to verify the validity of the documentation of the structural source-code regularities with respect to the new implementation of the system. In other words, it must be possible to verify whether or not the source-code entities that where added or modified during the evolution step respect the documented regularities.

  Conversely, not only the implementation of the system evolves over time. As we have mentioned earlier, changes in the requirements, execution environment, design, conventions and so on can have as consequence that the structural source-code regularities that govern a system are altered. As such, an approach for supporting structural source-code regularities must be able to deal with evolving regularities as well.

  In order to deal with this interplay between the documentation of structural source-code regularities on the one hand and the implementation of a system on the other hand, an approach must thus provide support for co-design and co-evolution between the two.

2.2 Software classifications

One of the cornerstones of our approach is the use of a software classification mechanism in order to group a set of conceptually related source-code entities which are characterized by a number of structural source-code regularities. The idea of using a classification mechanism to model groups of related entities is certainly not novel. In literature, various approaches exist which employ classifications in order to document different types of concerns in source code.

In the following sections we give an overview of some of the software classification models that have been proposed in literature. In Chapter 4, we will revisit these classification models and provide a comparison with the classification model we use in the approach we advance in this dissertation.

2.2.1 (Virtual) software classifications

In his PhD dissertation [DH98], Koen De Hondt introduced the notion of *software classifications* as a means to create online documentation of software artifacts in order to provide support for software evolution. Software classifications provide a lightweight and conceptually simple approach: a software classification is defined as a collection of software entities. For instance, applied to an object-oriented programming language, a software classification can contain a number of classes, fields, methods, and so on. In De Hondt's work, these software classifications are used to capture the extent of important concerns in the source code. For a developer who needs to change the software, these classifications can be used as a means to browse the implementation of a concern that needs to be altered.

The classifications in De Hondt's work can be specified in two ways. A first way is by explicitly enumerating software artifacts belonging to the classification. For the second way, De Hondt introduced the notion of a *virtual software classification*. A user does not enumerate the artifacts belonging to the classification in order to define this kind of classification, but rather provides an executable description (i.e. a query over the source code). Upon execution of this description, the elements belonging to the virtual software classification are calculated.

Wuyts provided tool support for this classification model of De Hondt by means of the StarBrowser [Wuy02, WD04]. StarBrowser is an extension to the Cincom VisualWorks Smalltalk environment and enables the classification of Smalltalk objects. These classifications can be created either by manually dragging and dropping elements into a classification, or by means of a specification written using a Smalltalk program or a SOUL [Wuy01] logic query.

Kim Mens extended the concept of virtual software classification in his PhD thesis [Men00], in which classifications lie at the heart of an approach to co-evolve an architectural description with the source code of a system using logic meta programming. Mens's approach differentiates between an architectural description language and an architectural mapping language. The description language describes the high-level concepts which make up the architecture of the system and the relations between these concepts. In order to specify the concepts in the architecture, constructs for ports, filters and links are also provided. In the architectural mapping language, virtual software classifications are created which relate the concepts from the architectural description to the implementation of the system and which allow keeping this architectural description synchronized with the source code of the system.

2.2.2 Concern Manipulation Environment

The Concern Manipulation Environment (CME) [HOST05a, HOST05b, CHK$^+$05, TCH$^+$04] is a tool suite implemented as part of the Eclipse [IBM] development environment. The goal of the CME is to provide an environment for supporting aspect-oriented software development [KLM$^+$97]. In CME, concerns are treated as first-class entities and are accessible throughout the entire software development cycle. The *ConMan* [HOST05a] component of the CME offers a classification model bearing similarities to the classification models discussed above. While in our work we will focus on the classification of source-code entities, the CME approach takes a broader view and allows the classification of artifacts from all cycles of the software development process. The classification model of the CME distinguishes between the following concepts:

- **Concern space:** top-level part of the model containing concerns, relationships, software artifacts, constraints, ...;
- **Concern:** the traditional notion of a concern. Concerns can be specified both by a query and by an enumeration;
- **Concern context:** specialization of a concern which relations and constraints can be attached to;
- **Relationship:** relationships represent other concern model elements which a certain concern model element is related to. E.g. for a class C, the relationships can include all classes which refer to C, all classes which extend C, and so on;
- **Constraint:** a constraint groups together a number of constrained elements.

2.2.3 Cosmos

Sutton and Rouvellou proposed Cosmos [SR02], an approach with a similar goal as the Concern Manipulation Environment, namely the documentation and manipulation of concerns in order to provide support for a multi-dimensional separation of concerns [TOHS99a] throughout the development process. They propose an elaborated classification schema consisting of "logical" and "physical" concerns. While logical concerns represent concepts of interest in a system, physical concerns represent the actual software artifacts which logical concerns apply to.

Cosmos contains five kinds of logical concerns, which aid in creating a structured representation of the different concerns in a software system:

- **classifications:** represent high-level concepts in the system and consist of a number of Cosmos classes. For instance, concerns such as functionality, state, configurability are considered to be classifications;

- **classes:** classes are used to group other concerns and consist of other classes or instances. A class belongs to a classification. For instance, core functionality, internal-consistency functionality, and so on are all classes belonging to the functionality classification;
- **instances:** these are the leaf concerns which do not classify any other concerns. For instance, in a graphical editor a number of examples of instances belonging to the core functionality class are drawing of figures, moving of figures, positioning, ...;
- **properties:** properties are concerns which characterize other concerns. Examples of properties are concurrency, synchronization, and so on;
- **topics:** these are arbitrary collections of concerns which generality cut across the entire set of classifications, such as for instance logging, caching, ...

These logical concerns classify "physical" concerns such as instances (specific source code entities such as e.g. classes, methods, ...), collections (groups of physical concerns) and attributes (particular properties of instances and collections which are deemed interesting in logical concerns). Cosmos not only makes different concerns in the software explicit, but also provides support for declaring how the different concerns are related. Moreover, Cosmos contains the notion of predicates, which are used to keep the concern schema created with Cosmos internally consistent.

2.2.4 Concern graphs

Robillard and Murphy present an approach to document concerns in software using a mechanism called *Concern Graphs* [RM02] which classifies a concern by storing the structure of that concern. A Concern Graph is a graph consisting of three kinds of vertices, namely classes, fields and methods. The edges in the graph represent common relations between vertices such as *calls*, *reads*, *declares*, and so on. A developer constructs a concern graph by querying the source code for a source-code artifact which (partially) implements the concern, and then by iteratively extending the concern graph with other source-code artifacts which also take part in the implementation of the concern, until the full extent of the concern is captured by the graph. To support the use of concern graphs, the Feature Exploration and Analysis tool (FEAT) [RM03] is provided which extends the Eclipse development environment and which provides support for displaying and browsing concern graphs. Moreover, FEAT also supports the construction of concern graphs by providing a developer query facilities which can be used for browsing the set of source-code artifacts that is related to a given artifact. This way, the combination of querying the source code and inspecting the results of the query is used to iteratively construct a concern graph.

Concern graphs offer an interesting approach in that they provide a methodology in which classifications are constructed using a combination of querying the source code and enumerating related source-code artifacts. In more recent work, Robillard extended [Rob06] the model of concern graphs with a mechanism which compares the source-code artifacts belonging to a concern graph after evolution with the artifacts belonging to the same graph before evolution.

By defining a number of heuristics, he proposes to use this information as a means to detect errors made during the evolution process.

Closely related to concern graphs, Robillard reports on ConcernMapper [RWW05], a simple Eclipse plugin which allows for the extensional creation of classifications by means of dragging and dropping source code entities into those classifications. The goal of this tool is to complement the relation-based classifications of FEAT with a mechanism to document concerns extensionally without having to reason about the structure of the implementation of the concern.

2.2.5 Conceptual Modules

Baniassad and Murphy presented *Conceptual Modules* [BM98], a classification mechanism used to support various kinds of reengineering tasks. A conceptual module consists of a name, representing a concept in the source code of a software system which is documented by the conceptual module, and a number of lines of source code which implement the documented concept. From these lines of code, a representation of the concept is derived in terms of:

- **input variables:** variables which are defined external to the lines of code belonging to the conceptual module, but which are used within the module;
- **local variables:** variables which are used and/or defined only from within the lines of code belonging to the module;
- **output variables:** variables which are defined in the conceptual module, and used external to it;
- **calls to procedures:** the procedures which are called from within the source code belonging to the conceptual module.

In addition to providing an interface for accessing the information which is associated with a conceptual module, a query language is provided which can be used to verify whether, either directly or indirectly, two conceptual modules use the same variables, whether two conceptual modules overlap or whether one conceptual module is contained within another.

Conceptual modules are a highly specialized classification mechanism which can be used to identify how the implementation of different concepts in the source code of a system interact. While this technique offers a limited amount of information concerning the entities in the conceptual modules, a number of case studies have been performed using this technique which show its aptness for supporting reengineering tasks.

2.2.6 Mylar

Mylar [KM05a] by Kersten et al. is an extension to the Eclipse Java Development Tools (JDT) and the AspectJ Development Tools (AJDT). Both JDT as well as AJDT make use of nested classifications in order to provide a developer means for navigating the source code of a piece of software. For instance, each package in a Java program is visualized by a classification containing the classes belonging to the package. In turn, these classes are also represented by a classification containing the methods of a class.

Kersten et al. remarked that when the size of a program increases, the amount of information belonging to these classifications can become overwhelming to a developer. To alleviate this problem, they propose the use of a degree-of-interest (DOI) model which limits the scope of the entities visible in the classifications to those related with the current task of the developer. This DOI model attributes an interest value to each element of a classification: if an element is selected or edited, the interest value will increase; over time, if the element is no longer selected or no changes to it are made, the interest value of the element will gradually decay. Only the elements with an interest value above a certain threshold will be shown in the editor.

While Mylar is implemented as an Eclipse plugin, the idea behind this technique is complementary with almost all classification mechanisms discussed above. The use of a degree-of-interest model to highlight only relevant entities in the visualization of a classification can be considered an important factor in improving the scalability of classification mechanisms.

2.2.7 Summary

In the previous sections we gave a brief overview of a number of approaches which use a particular classification mechanism to support a task during the software development process. While these approaches do not target the specific problem of supporting structural source-code regularities throughout the implementation of a software system, they have proven in the past to be a valuable technique for creating different kinds of documentation. As such, we will also employ a classification mechanism in our approach to document groups of related source-code entities. While some approaches such as CME and Cosmos provide an elaborated scheme for documenting and managing concerns in general in a system, approaches such as Conceptual Modules offer a more dedicated, task-specific usage of a classification mechanism. This results in that classification mechanisms we discussed above often differing in the constructs they offer to a developer.

However, all of the approaches we discussed share the following commonalities. Regardless of the complexity and extent of the constructs they offer, the core idea behind these approaches is to use classification mechanisms in order to group software artifacts which are deemed related for performing some task. Furthermore, we see that there are two ways in which the classifications can be specified. Either a developer manually enumerates the elements belonging to a classification. This mode of defining a classification is also called *extensional*; or the classification is defined by means of a query, which upon execution yields the artifacts belonging to a classification. Such a means of specifying a classification is often dubbed *intensional*. While the former way of defining a classification has as an advantage that this can be done by e.g. simply dragging and dropping certain artifacts in a classification, the latter way, namely intensionally defining a classification, offers the benefit of being more robust with respect to evolution. Whenever the underlying set of software artifacts changes, the artifacts belonging to a classification can be recomputed by the query defining the classification.

2.3 Support for structural source-code regularities

In this section, we give an overview of the state of the art in tools, methodologies and techniques which provide the documentation and/or verification of structural regularities in the source code of systems. We divide this section in three major parts. In Section 2.3.1 we take a look at a group of tools which check for infringements of good coding style, programming idioms, and so on. Section 2.3.2 describes a number of approaches which provide a developer with a language in which constraints can be imposed over the system. The third part (Section 2.3.3) of this section consists of a description of a number of approaches which verify conformance of high-level architectural views of the system with respect to the source code.

2.3.1 Code checkers

A first group of approaches we discuss are *code checkers*. These approaches check for violations of particular programming idioms (e.g. correct implementation of comparison of objects), frequently occurring bugs (e.g. use of a variable before it is initialized), coding style infringements (e.g. improper use of names) and so on.

Lint

Historically, Lint [Joh79] is one of the first code style checkers. It consists of a fixed set of rules which can be applied to a C [KR88] program and which inform a developer of often-occurring problems in the source code of that program. More specifically, Lint informs the user of unused variables and functions, the use of variables before they are set, unreachable portions of a program, improper use of return values of functions, advanced type errors, incorrect use of type casts, “strange” constructions, deprecated syntax and statements for which the semantics are different dependent on the compiler which is used. Lint offers a lightweight approach in detecting these problems: instead of using a complex, computation-intensive analysis of the C program, it makes use of a number of approximations and heuristics to detect violations. As such, it provides a conservative estimate of possible violations of the Lint rules in the source code of a program.

LCLint

Evans et al. propose LCLint [EGHYM94], a Lint-like approach based on the Larch [GHG$^+$93] formal specification languages and tool suite. Similar to Lint, the goal of their approach is to find common bugs in C code. However, in order to refine the analysis of the program, they propose an approach in which the typical C header files (`.h`) are substituted with `.lcl` files in which variables and functions can be annotated in order to provide additional information which can be used by the checker. For example, by explicitly specifying that a given structure in a program is an abstract, unmutable data type, additional checks can be performed to guarantee that all clients of that type use the proper functions to modify that data type. In addition to checking the proper use of abstract types, LCLint is able to detect occurrences of unauthorized usage of global variables, undocumented modifications of the state of the arguments of a function and missing initializations for actual parameters or use of uninitialized formal parameters.

FindBugs

FindBugs [HP04] is an open-source tool which strives to find possible bugs in Java code. It is based on the assumption that certain code idioms are likely to be an error. Based on this assumption, it detects a whole range of possible bugs such as the absence of a required `super` send in implementors of the Cloneable interface, suspicious use of the `equals()` method to compare incomparable objects, the invariant that objects implementing `equals()` should also implement `hashCode()`, and so on.

Code style checkers

CheckStyle [Che06] is an open-source tool which checks for a large number of violations on proper Java coding style. Amongst others, CheckStyle finds inconsistencies with respect to standard Java naming conventions, use of white space, size violations, duplicate code, bad smells, ... While the set of rules which are checked by CheckStyle is fixed, the tool offers to a developer the possibility to tweak the parameters of each of the rules.

Similarly, PMD [PMD06] is another tool which checks Java code for unused fields, empty blocks in if, while, try-catch, ... statements, unused method parameters, and so on. On top of the pre-specified set of rules, developers can specify custom rules using the XPath [CD99] query language.

Pattern-Lint

Sefika et al. present *Pattern-Lint* [SSC96], a tool for checking conformance of the implementation of design patterns and a number of architectural styles such as client-server and pipe-filter. A developer selects a certain pattern in the tool and provides the mapping of the roles of the pattern to elements in the source code of the program. Pattern-Lint will then report on any inconsistencies between the pattern and the implementation using both a static as well as a dynamic analysis.

P^3

The P^3 (Practical Preprocessor for Programming conventions) system [DC03] is a preprocessor for Java that checks class files for 60 different code and design conventions. Amongst others, P^3 checks for violations of syntactical constraints such as the use of public variables, uniform location of line breaks and braces, and so on. Particular for P^3 is the fact that the system is not restricted to detecting the violations. After the detection, it makes use of a *neural network* in order to provide an ordered set of (semi-automated) corrective actions in order to rectify violated constraints in the source code. The system also provides a scripting language which allows the detectable violations to be customized.

Summary

The code checkers presented above can be classified into two groups. A first group consists of approaches such as Lint, LCLint and FindBugs whose primary goal is to detect a number of infringements of implementation regularities in source code. Such regularities check for common mistakes, violations of environment constraints, language idioms, and so on. The second group, which CheckStyle, PMD and P^3 belong to do not focus on finding bugs but rather aim at finding entities in the source code which exhibit a "bad coding style", i.e. stylistic regularities.

These approaches share the disadvantage that they do not provide *explicit* documentation for the different structural source-code regularities present in a software system. Although they provide support for verifying a set of rules defining a number of regularities, this set is often internal to the associated tool and is thus less suitable as a means of documentation of a given system. Furthermore, with the exception of PMD and FindBugs, these tools offer a limited, non-customizable set of rules. Although these sets of rules are considered to be generally applicable and allow the verification of a range of system-wide constraints, they can only be used to support specific kinds of structural source-code regularities. For instance, these approaches are not suited for documenting and verifying regularities which are specific to one particular domain or application. Finally, while these techniques support evolution by making it possible to verify the set of rules with respect to evolved versions of a software system, they do not support the evolution of the regularities themselves. Since for most approaches the set of rules is not easily accessible nor mutable, the support these approaches offer for co-design and co-evolution is seriously hampered.

By no means is the list of code checkers we discussed above complete. There exists a proliferation of tools and approaches both from the open-source community as well as commercial products that offer functionality comparable to the approaches we discussed. For example, tools such as Decor [MGL06], iPlasma [MMM$^+$05], RevJava [Flo02] and the *Software Source Code Static Quality Analysis* [M S07] tool apply metrics to a software system to detect design defects, bad smells, and so on. Lint4J [JUt07] provides a set of Lint-like rules for Java programs.

2.3.2 Meta-programming approaches

In this section we describe a number of approaches which offer a developer language support for expressing regularities in terms of the source code of a system. Regularities can be declared using these approaches as constraints which are imposed on the source code and which can be verified with respect to this source code.

Query-based approaches

The Smalltalk Open Unification Language (SOUL) [Wuy01] is an implementation of a Prolog-like [DEDC96] language on top of the Smalltalk [GR89] object-oriented language. Wuyts et al. have shown that the declarative paradigm is well-suited for reasoning about source code in general and for implementing support for verifying source-code regularities in particular [MMW01, WM06]. To this end, SOUL offers a tight, symbiotic integration with

```
1 drawViolations(?class) if
2   subclassOf(?class,[Figure]),
3   not(methodWithNameInClass(?method, draw, ?class))
```

Figure 2.1: Example of a SOUL query

the underlying Smalltalk language such that during evaluation of logic queries, Smalltalk code can also be executed. Moreover, a library of logic predicates named *LiCoR* is provided that offers a complete reification of Smalltalk programs complemented with predicates for reasoning over the code.

An example of a SOUL program is shown in Figure 2.1. This program implements the design constraint that in a graphical editor all subclasses of the class `Figure` must override a method `draw`. The constraint consists of a rule `drawViolations`. The head of the rule contains the logic variable `?class` (variables in SOUL are indicated by a question mark). In line 2, a first condition is imposed which requires the variable `?class` to be bound to subclasses of the class `Figure` (the brackets around `Figure` are used to execute Smalltalk code, as such the actual `Figure` class from Smalltalk is bound to the variable `?class`). Line 3 further restricts the bindings of `?class` to those classes which do not implement a method named `draw`. When executing the `drawViolations` rule, the variable `?class` will be bound to all classes in the `Figure` hierarchy which do not implement `draw`, i.e. the set of classes which violate the constraint we wished to express.

Using SOUL, or other declarative meta-programming languages such as Tyruba [De 98], JQuery [JD03] and CodeQuest [HVd06], it is possible to implement meta-programs expressing a whole range of verifiable structural regularities.

CCEL

```
// All subclasses of class Figure must override a method called draw
DrawConstraint(
 Class P | (P.name() == "Figure");
 Class C | (C.is_descendant(P));
 MemberFunction P::function | (function.name() == "draw");

 Assert([MemberFunction C:function2;|function2.redefines(function)]);
)
```

Figure 2.2: Example of a CCEL constraint

The C++ Constraint Expression Language (CCEL) [DMR92] is a meta-language for C++ [Str86]. The goal of CCEL is to provide developers with a meta-language for implementing checkers for design, implementation and stylistic constraints, similar to our goal of verifying structural source-code regularities. CCEL adopts an object-oriented model representing C++ programs and offers a constraint language capable of expressing predicate logic-like constraints. An example of a CCEL constraint is shown in Figure 2.2. This constraint is the CCEL version of the constraint we described in the section above, namely that all subclasses of the class `Figure` must override a method `draw`. Lines 3 and 4 create two new variables **P** and **C** which are respectively bound to a class with name `Figure` and the subclasses of this class. Line 5 binds a variable **function** to a method with name `draw` implemented in the class bound to variable **P** (i.e. the class `Figure`). Variables in CCEL are typed and can be of the type 'Class', 'Field' or 'MemberFunction'. The part after the ''|' in lines 3–5 provides a number of conditions which bindings of the variable need to fulfill. In CCEL, variables are universally quantified by default. Line 7 shows the assertion which must be true in order to have the constraint upheld. In the example, this assertion states that there should exist a binding for a variable **function2** of type 'MemberFunction' such that this binding redefines the method bound to variable **function**. Note that the use of brackets indicates that the variable **function2** is existentially qualified.

At a technical level, the implementation of CCEL suffers from a number of limitations. For instance, not the entire source code of a C++ program is reified at the level of the model in terms of which a developer can write constraints. More specifically, CCEL does not support reasoning about the internals of methods. Moreover, the constraints in CCEL are not first-class entities. It is not possible to refer from within one constraint to another constraint, nor does CCEL provide a modularization mechanism for constraints. As a consequence, multiple constraints over the same source-code entities require a developer to repeat parts of the code implementing the constraint. E.g. if we wish to write a new constraint over all the subclasses of `Figure`, this would require us to repeat lines 3 and 4 in our new constraint.

SCL

```
for F: subclasses(class("Figure")) holds
   method("draw", F)
```

Figure 2.3: Example of a SCL constraint

Hou and Hoover present SCL (Structural Constraint Language) [HH06]. Similar to supporting structural source-code regularities, the goal of SCL is to provide a means for enforcing the non-functional design intent of a developer (such as for instance coding style, design patterns, idioms, and so on) in the source code of a system. Using SCL, developers impose their design intent as constraints on a representation of a program. SCL offers a specification language based on first-order predicate logic complemented with a set of predicates for reasoning about source code. An object-oriented program is represented as a graph in which the nodes are source-code entities (classes, methods, fields, ...); the edges connecting two nodes represent the relations between the source-code entities (such as calling relations, and so on). The library of predicates for reasoning about source code offers a reification of these relations between source-code entities. Hou et al. provide two implementations of SCL, respectively supporting the C++ and Java languages.

Figure 2.3 illustrates an example of a SCL constraint. This constraint expresses the SCL variant of the example constraint we discussed for SOUL and CCEL, namely it verifies that all subclasses of the class `Figure` override a method `draw`. Line 1 retrieves all subclasses of the class `Figure` and binds these subclasses to the variable `F`. The constraint expresses that for all these subclasses it must hold that they contain a method named `draw` (line 2).

Attribute Extension

```
addTo ClassDecl {
  syn boolean isFigureSubclass =
    (superClassBinding() =/= null and
    (superClassBinding().globalName() = "Figure" or
    superClassBinding().isFigureSubclass));

  eq DeclList.isFigureSubclass = isFigureSubclass;
}

addTo DeclList {
  error string missingDrawMethod =
   if isFigureSubclass and not implementsDrawMethod
   then "Subclass of Figure does not implement draw"
   else "";
}
```

Figure 2.4: Example of an Attribute Extension check

In her work on *Attribute Extension* [Hed97a] Görel Hedin focusses on the verification of conventions which underly object-oriented libraries, frameworks and design patterns [Hed97b]. She proposes a technique that extends the parser of a language such that checks for conventions, which are verified at compile-time, can be implemented. As such, Attribute Extension allows for extending the name and type analysis for a programming language, in a way that application-specific constraints can be verified. Attribute Extension consists of the following four components:

- **Base grammar interface:** an object-oriented representation of a context-free grammar of the base language which the checks are defined over;
- **Extension grammar:** an attribute grammar which extends the production rules from the base grammar interface with attribute declarations and equations. These extensions can respectively be used to propagate information throughout the parse tree of a program or to calculate properties of single program entities;
- **Attribute comments:** an annotation mechanism that makes it possible to attach additional semantical or contextual information to program entities from within the source code of the underlying base program;
- **Error attributes:** these implement the actual checks.

Figure 2.4 shows an example of an Attribute Extension check. Similar to the example we used in the previous sections, we wish to express that subclasses of *Figure* should implement a method `draw`. Lines 1–8 of the example extend the class definition nodes in the base grammar with information about subclasses of `Figure`. Lines 2–5 introduce an attribute *isFigureSubclass* which is true if the class itself or one of its superclasses is the `Figure` class. Line 7 propagates the result of this synthesized attribute to the declarations of the class. Lines 10–14 show a second extension of the grammar. This time declaration lists are extended with an error attribute *missingDrawMethod* which is triggered if the *isFigureSubclass* attribute is true and the *implementsDrawMethod* attribute is false. For reasons of brevity we did not include the implementation of the *implementsDrawMethod* attribute in our example.

Although attribute extension is expressive enough to implement checkers for a wide range of regularities, it can often be tedious to implement a constraint verifying a regularity. Attribute extension offers an extensible meta-model that extends the parser of a program with a means to specify constraints over the source code of a program. However, due to the cross-cutting nature of certain regularities, the fact that the constraints defined using Attribute Extension align with an abstract syntax tree of a program make this approach a less suitable candidate for documenting structural source-code regularities.

Law-governed architecture

```
sent(P, transferAmount(A), T) :-
     positive(A)@P,
     do(addAmount(A)@T),
     do(removeAmount(A)@P).
```

Figure 2.5: Example of a law-governed system

Naftaly Minsky proposes *Law-governed architecture (LGA)* [Min96, MP97, Min91, MP00] as a technique to enforce regularities in software. He does not limit himself to structural regularities in the source code of a system but also considers regularities which need to be adhered to at run-time. LGA imposes a law over the software system, i.e. a global and explicit set of rules which are enforced by the environment that manages a software project. Laws in LGA can be used to regulate different kinds of regularities: it has been used to enforce the architectural regularities underlying multi-tier architecture, to ensure encapsulation, and so on. Also, rules can be specified which express how the law of the system itself can evolve, how the system needs to be configured. LGA rules can even be used to manage the actual development process (e.g. by allowing developers only access to the modules they are implementing).

Figure 2.5 shows an example of a rule in law-governed system. LGA uses a Prolog-like language which rules are defined in that state the laws which need to be adhered to in a system. In general, rules can be *permission-based* by e.g. stating which interactions are allowed or *prohibition-based* by declaring interactions which are not allowable in the system. Our example shows a rule governing the correct behavior in a banking system. The rule expresses that, in order for a banktransfer from a bankaccount P to an account T to be legal, a positive amount (line 2) A needs to be added to account T (line 3) and be subtracted from account P (line 4).

Rules in LGA can also trigger actions, redirect message sends, omit message sends, invoke error messages and so on. As such, the law of the system is not a kind of passive, verifiable documentation but rather is part of the implementation of a system. While our example shows a run-time constraint, LGA also offers a (limited) set of logic primitives to reason about more structural relations in software.

IRC

```
1  public class SystemOutChecker implements IRC {
2   ByteCodePointcut pc = new BytecodePointcut(
3     new AndFilter(
4      FieldAccessDeclaringClassFilter.create("java.lang.System"),
5      new OrFilter(
6        FieldAccessNameFilter.create("out"),
7        FieldAccessNameFilter.create("err"))));
8   public List check(ClassFile cf){
9      List inst = pointcut.getInstructions(cf);
10     return IRStatus.create(instrs, RESTRICTION,
11               DESCRIPTION, IRStatus.ERROR);
12 } }
```

Figure 2.6: Example of an IRC check

Eichberg et al. propose, IRC (Implementation Restriction Checker) [EMS+04], a platform for implementing checkers that enforce system-wide properties. IRC starts from the observation that checking system-wide properties can be considered to be crosscutting concerns. As such Eichberg makes use of aspect-oriented technology [KLM+97]. In particular the BAT framework [BE], which provides AOP-facilities at the level of Java bytecode, is used.

An example of a check implemented using IRC is shown in Figure 2.6 (example adopted from [EMS+04]). A check in IRC is a class (in our example `SystemOutChecker`) which implements the interface `IRC`. Each checking class must implement a method `check` which, given a class file, returns a list of violations (these are instances of `IRStatus`). In our example, this is reflected in lines 8–11. The actual violations are found using a BAT pointcut (lines 2–7). The pointcut in our example selects all the byte code instructions referring to `java.lang.System` (line 4). This set of instructions is further reduced to those who either access the field `out` or the field `err` (lines 6 and 7). The result of the check in Figure 2.6 is thus that a warning is generated for all locations which access `out` or `err` on `Java.lang.System`.

While IRC is able to express a wide range of regularities, it does not offer a structured approach for documenting and verifying those regularities. Each check is a separate Java program which uses the IRC framework in which the explicit set of violations is captured using a BAT pointcut. While this shows numerous similarities with the SOUL language we discussed above, the declarative nature of the latter renders it more expressive in order to reason about software. Due to the fact that it is based on aspect-oriented technology, IRC is however able to enforce dynamic regularities by weaving in checks which are verified at run-time.

CoffeeStrainer

```
1  public abstract class Figure {
2   public void draw() {
3      /* body of draw */
4      }
5    /*
6    private AUserType getFigureClass() {
7      return Naming.getClass("Figure");
8      }
9    private AMethod getDrawInClass(AUserType class) {
10     return Naming.getInstanceMethod(class, "draw", new AType[0]);
11     }
12   private boolean overrides(AMethod m1, AMethod m2) {
13     if (m1==null) return false;
14     if (m1.getOverriddenMethod()==m2) return true;
15     else return overrides(m1.getOverriddenMethod(), m2);
16    }
17    public boolean checkCorrectlyOverridden(AUserType c) {
18     rationale = "all subclasses of Figure must override draw";
19     return implies(c.getSuperclass()==getFigureClass(),
20            overrides(getDrawInClass(c), getDrawInClass(getExample()))
21            )
22     }
23     */
24 }
25
```

Figure 2.7: Example of a CoffeeStrainer constraint

Bokowski proposes *CoffeeStrainer* [Bok99], a system for statically checking user-specified constraints on Java programs. Rather than providing a new constraint language, CoffeeStrainer uses Java in order to define constraints. CoffeeStrainer provides the implementor of a constraint with a reification of Java AST trees which is accessible from within the Java code.

An example of a CoffeeStrainer constraint is shown in Figure 2.7. This constraint implements the design regularity that all subclasses of `Figure` must override a method named `draw`. Note that in CoffeeStrainer, the constraints are implemented as a comment in the class over which they impose a constraint. Lines 6–11 implement two auxiliary methods which respectively return the class `Figure` and a method named `draw` for a given class. Lines 12–16 implement a predicate that checks whether a method `m1` overrides a method `m2`. Lines 17–22 implement the actual constraint. The constraint is a rationale which describes the constraint and a condition which expresses if a class `c` has the `Figure` class as its superclass. This implies that `c` implements a method `draw` which overrides the method with the same name in the `Figure` class.

MJ

```
1  sm lockunlock {
2     state decl anyobject v;
3
4  start:
5    { v."lock"(...) } ==> v.locked;
6
7  v.locked:
8    { v."unlock"(...) } ==> v.stop
9    | \$end_of_path\$ ==> {
10     /* error */ };
11 }
```

Figure 2.8: Example of a MJ checker for locking/unlocking

MJ [BE03] is an innovative meta-compilation approach for implementing bug-checkers in Java. The result of an MJ check is a compiler extension which can be applied to user code in order to flag violations of certain rules. MJ offers support for checking regularities based on a static data-flow analysis. A check in MJ is written down as a state machine. If this state machine reaches an inconsistency, this inconsistency is flagged.

Figure 2.8 shows an example of a check in MJ, adopted from [BE03]. This check verifies whether within a software system, a call to a `lock` method is always paired with a call to `unlock`. The state machine *lockunlock* found in Figure 2.8 consists of two states: *start* and *v.locked*. Line 2 declares a state variable `v` of type `anyobject`. The state machine starts out in the *start* state. If in a procedure a call of a message `lock` occurs, then *v.locked* becomes the current state. If in this state, an `unlock` message occurs, the state machine stops and the regularity is not violated (line 8). If however the data-flow exits the current procedure before `unlock` was sent, an error is raised (line 9).

While MJ is able to express a variety of regularities which depend on data-flow, it does not provide support for other kinds of regularities like naming conventions, inheritance regularities, and so on.

Summary

The sections above presented an overview of a number of examples of meta-programming approaches. The greatest common denominator of these approaches is that they offer language facilities that allow exploring and reasoning about the source-code of a system. Depending on the expressiveness of the provided language, and the richness of the underlying representation of the software over which the language reasons, these approaches can be used to document a wide variety of structural (and sometimes even more dynamic) regularities and verify conformance of these regularities with the source code. Such regularities are documented by creating a meta-program that analyzes the source code of a system, and retrieves the set of source-code artifacts that violate that particular regularity.

While these meta-languages provide an excellent technical platform on which one can graft an approach for documenting and verifying structural source-code regularities, such languages by themselves do not enable the documentation of the regularities, nor do they aid in incorporating these documented regularities in the development process. Although the different structural source-code regularities can be verified by means of a meta-program, such a meta-program does not provide structured and explicit documentation for this regularity.

In Section 2.1.5 we also specified the requirement that an approach for aiding the consistent use of structural source-code regularities during the development process must support the co-design and co-evolution of the documentation and the implementation in order not to deal only with changes in the implementation, but also with changes in the regularities. While, with the exception of law-governed architecture that supports co-evolution, none of these meta-programming approaches we discussed in the above sections directly support such a methodology, this does not mean that they fundamentally inhibit the co-design and co-evolution of the documented regularities and the implementation of a system. As we have discussed above, these meta-languages offer a developer a means to implement checkers for structural source-code regularities. While some of these checkers enforce the documented regularities (e.g. IRC, Attribute Extension), it is not unthinkable to incorporate these languages in an approach for supporting structural source-code regularities as facilities used to reason about source code.

2.3.3 Architectural conformance checkers

In this section we discuss a number of approaches which check conformance of a specification of the architecture of a system with the underlying implementation. While the domain of these approaches is broader than verifying structural source-code regularities, a number of such regularities typically occur at an architectural level. As such, we include these kinds of approaches in our discussion.

Reflexion Models

Murphy et al. present *Reflexion Models* [MNS95] as an approach to compare a high-level model of a software system with the actual implementation. A developer using reflexion models starts out by creating a high-level model of interest of a system. This model consists of the different modules of which the system is composed and the dependencies (i.e. the calling relations that should occur) between these modules. At the same time, a source-code model is extracted from the system. To this end, the developer needs to declare *extensionally* how the modules from the high-level model map onto sets of entities from the source-code of the system. Based on this high-level model and the source model, the approach then calculates a reflexion model that highlights the discrepancies between the high-level, conceptual model and the source code.

This technique provides facilities for verifying one particular kind of architectural source-code regularity. More specificly, it allows for the verification of regularities that express how certain modules should interact. This is achieved by comparing a high-level description of the different modules in the system and their interactions with the actual implementation of the system.

Tool support for design patterns

Florijn et al. present an approach and corresponding tool [FMv97] for supporting design patterns [GHJV95] in object-oriented languages. While their approach focusses on providing tool support for documenting, composing and generating instances of design patterns, it also makes it possible to express the relationships between the different roles of a design pattern. Moreover, the conformance of the implementation of a pattern to the prototype of the pattern can be verified. When inconsistencies are encountered, their approach even provides support for automatically resolving these inconsistencies.

Florijn's approach is based on the notion of *fragments*: a representation of a design element together with a number of roles which are associated with a fragment. Instances of a pattern are described by means of a *fragment graph* which relates the different roles of a design pattern to concrete classes, methods, and so on in the implementation. Moreover, the graph also contains information about how the different source-code entities are related (hierarchical information, usage relations, and so on). Both the prototypical structure and the common parts of the behavior of a design pattern are represented by a *fragment graph*. A particular instance of a design pattern contains a *parent* role which associates it with the prototype of the pattern. To detect inconsistencies in the implementation of a design pattern, constraints are imposed on the fragment graph which express how the different roles of the pattern are related and should interact.

Ptidej

Guéhéneuc et al. propose Ptidej (Pattern Trace Identification, Detection and Enhancement in Java) [Gu2, GAA01]. This approach aims at supporting the correct use of design patterns in a system. To this end, Ptidej allows for the identification of design patterns and the detection and resolution of inconsistencies between the patterns and the implementation.

Ptidej encompasses a number of components:

- **Meta-model:** a fully reified meta-model is used to formalize the actual design patterns and the relations which exist between the components of the design patterns. The same meta-model is used to describe the concrete architecture of the system. Ptidej provides support for visualizing these meta-models, thus making it possible for a developer to extract the design of a system;

- **Caffeine:** Caffeine [GDN02] is a dynamic analysis tool for Java which can be used to augment the static information in the meta-model with more detailed information about the actual interactions between the different entities in the system;

- **Explanation-based constraint solver:** In order to detect instances of a design pattern and infractions against the correct implementation of that pattern, a description of the various components of a design pattern and the interactions between these components is transformed into a constraint satisfaction problem.

  By solving this constraint satisfaction problem with respect to the actual architecture of the system, instances of a design pattern can be detected. Moreover, the constraint solver reports on any constraints which need to be relaxed in order to find a solution to the constraint satisfaction problem. These relaxed constraints can be an indication of an inconsistency between a design pattern and an implementation of that pattern;

- **Program transformations:** For each of the constraints in the constraint satisfaction problem, a transformation rule is associated which can be used to automatically rectify the inconsistency in the implementation with respect to a particular design pattern.

In Ptidej, the (abstract) design pattern is encoded in the meta-model. This representation of the pattern is then used to detect instances of the design pattern in the implementation and possible defects in the pattern's implementation. As such, the actual design pattern is documented explicitly, but the instances of the pattern are documented implicitly.

Two-tier programming

Eden et al. propose *two-tier programming (TTP)* [EKF03], a *conceptual* framework for preventing architectural drift and architectural erosion. Their solution is based on providing, on top of the implementation of a system, a causally connected second level which represents the architecture of a system. Using TTP a system consists of:

- **first-order tier:** The actual implementation of the system;
- **second-order tier:** A high-level representation of architectural concepts and dependencies;
- **association mapping:** Linking the high-level concepts in the second-order tier to entities in the first-order tier.

In [EKF03], a concrete instantiation is shown of a two-tier program, in which an implementation of the template method design pattern [GHJV95] in C++ is considered the first-order tier. For the second-order tier, a formal specification of the template method pattern is provided in LePUS [EH99, Ede02], a modeling language for object-oriented architectures. Moreover, a mapping is specified which associates entities from the C++ program to the different roles of the design pattern.

TTP prevents architectural drift by not allowing a developer to simultaneously changing both the first-order as well as the second-order tier. This way, support is offered for co-evolving the specification of the architecture in the second-order tier with the implementation in the first-order tier. After changes in one tier are made, it is checked whether the two tiers remain consistent. If not, the developer can synchronize both tiers.

Virtual Software Classifications

In Section 2.2.1 we already discussed the technique of virtual software classifications and in particular we took a look at the classification mechanism underlying this technique. Mens et al. [MWD99, Men00] describe how virtual software classifications can be used to specify the architecture of a software system at a high level and how this specification can be used to prevent the architecture from drifting from the implementation. They apply the Turing-complete logic programming language SOUL in order to describe the components in the architecture and their inter-relations. Each component in the architecture aligns to a virtual classification, which is defined by a SOUL query. Predicates are used to express relations between two classifications, i.e. architectural components. Since the architecture is expressed in terms of the implementation, it is possible to verify conformance between the architecture and the source code, thus aiding in co-evolving both artifacts.

Summary

The architectural conformance checkers we discussed above share as a commonality that they all document a part of the structure of a system at a higher level of abstraction. Whether they provide a description of the different modules in the system and their interactions, or of the different design patterns that are used in a system, the structural source-code regularities that can be codified using these techniques all manifest themselves in the source code at the design or architectural level. This allows these techniques to provide specialized support for some particular kind of regularity. For instance, techniques such as Ptidej and the work of Florijn are not only able to detect infringements against the regularities underlying a design pattern, but also provide automated support to resolve such infringements.

These architectural conformance checkers, and especially two-tier programming and virtual software classification, propose a methodology in which co-evolution of the documentation of the regularities and the implementation is supported. While these approaches only focus on supporting a specialized kind of structural source-code regularities, they offer support for situations in which upon evolution not only the implementation but also the regularities need to be adapted.

2.4 Conclusion

In this chapter we introduced the common terminology concerning structural source-code regularities which we will use throughout this dissertation. We discussed some of the properties of regularities and gave an overview of software classification mechanisms. Furthermore, we provided a survey of the different kinds of approaches that support verifying the validity of structural source-code regularities with respect to the source code of a system. Although a fairly large body of research exists, and we by no means claim to provide a complete overview of this work, we can distinguish between three kinds of approaches:

- **Code checkers:** these tools provide a reusable set of generally applicable constraints which can be used to detect common errors, bad style, etc, in the source code of a system;

- **Meta-programming approaches:** these approaches provide a language which reasons about a program and allows for the specification of verifiable constraints over this program;

- **Architectural conformance checkers:** this category encompasses a number of approaches which verify the validity of a high-level description of a program with the implementation.

In the next two chapters we propose a novel technique that supports the documentation and verification of structural source-code regularities and integrates the regularities into the development process. While in Chapter 3 we approach our model of intensional views from a conceptual angle, Chapter 4 introduces IntensiVE, our tool suite that provides a concrete instantiation of the model of intensional views. Our approach both complements and advances the state of the art by:

- **Offering a generic approach for documenting regularities**: The approach we will introduce in the next chapter does not focus on supporting a specific kind of regularity but rather aims at providing a *general platform* for documenting structural source-code regularities. Consequently, our approach supports the documentation of naming conventions, stylistic constraints, design requirements, language idioms and so on.

  While such an approach comes at the cost that it does not provide the specialized support some of the existing work offers, such as e.g. the automatic resolution of conflicts in the implementation of design patterns or the optimized analysis performed by code checkers, our approach however does offer the advantage that a single medium and tool suite can be used to document and verify a multitude of different kinds of regularities;

- **Providing a structured means for documenting regularities**: Our approach offers a structured means for creating verifiable documentation of structural source-code regularities. In our methodology, we put forward a number of guidelines that describe how the different concepts in our formalism can be used to document structural source-code regularities. Moreover, the documentation created using our approach results in a *first-class* representation of the documented regularities as well as the source-code entities to which the regularities are applicable, thus making them *explicit* to the developer;

- **Actively supporting co-design and co-evolution**: Our approach and methodology have been tailored to actively support the co-design and co-evolution of the structural source-code regularities and the implementation of a system. Both our formalism as well as our methodology have been tailored to support this integration of regularities into the development process. Consequently, our approach supports simultaneous development and evolution of documented regularities and source code while maintaining the causal connection between both.

Chapter 3

Intensional Views Model

In this chapter we introduce the model of Intensional Views and Intensional Constraints. We take a look at the model from a conceptual perspective and provide definitions for the concepts in our model using a formalism inspired by relational tuple calculus [Cod70]. For an actual implementation of our model, we defer to Chapter 4, where we describe IntensiVE, the tool suite that supports the model of intensional views and constraints.

The motivation behind approaching this matter from a more theoretical viewpoint is twofold. First off, expressing the concepts behind intensional views and intensional constraints using a formalism allows us to introduce the model independently from the underlying software model over which it reasons, the implementation language of the tool suite and the query language which lies at the heart of our approach. As such, we can abstract away from implementation details and technicalities and focus on the conceptual framework our model offers. Second, the formal specification of the model enables us unambiguously to describe the concepts underlying intensional views. Moreover, this specification highlights a number of requirements which concrete implementations of the model must adhere to. These requirements lie at the basis of some of the design decisions we will discuss in chapter 4.

3.1 Concepts of Intensional Views

Before we give a formal definition of our model of intensional views, we introduce the important concepts of our model by means of an example. This example will be used as a running example throughout this chapter.

Figure 3.1 shows the UML diagram of the implementation of a small banking system. This system consists of two class hierarchies: `Account` and `Card`. An `Account` has two fields named `owner` and `balance` together with accessor methods for retrieving the contents of a field (`getOwner()`, `getBalance()`), mutator methods for altering the contents of a field (`setOwner(..)`, `setBalance(..)`), and a method for transferring money from one account to another. In our system, there are two types of concrete accounts namely `CheckingAccount` and `SavingsAccount`. We have two kinds of cards in our example: `BankCard` and `CreditCard`, which are associated with a `CheckingAccount`. Similar to the implementation of accounts, for each of the fields of our classes there exists an ac-

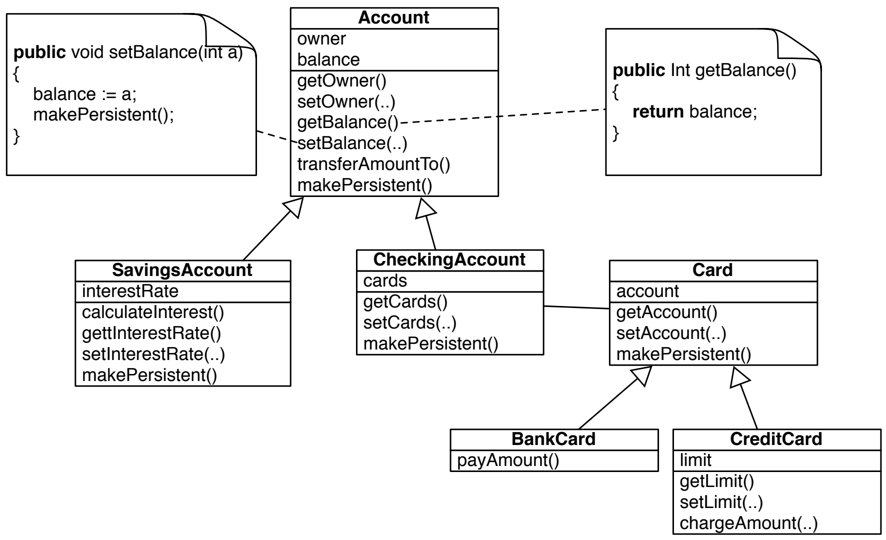


Figure 3.1: Running example: banking system

cessor and a mutator method. What's more, all classes in our system understand a message `makePersistent()` which writes the state of an object to the database.

The implementation of an accessor and a mutator method in a Java-like syntax is shown in Figure 3.1. These methods follow a number of conventions. The accessor method `getBalance()` follows the naming convention that all accessor methods' names start with the prefix `get-`. Moreover, each accessor methods consists of a single statement which returns the value of the field. For mutator methods (such as `setBalance(..)`), a similar naming convention is used. Such methods start with a prefix `set-`. Since we want to make all changes to the state of objects persistent, each mutator method must also include a call to the `makePersistent()` method.

Our model of intensional views contains two kinds of concepts:

- **Intensional Views:** intensional views are sets of source code entities (classes, methods, fields, ...) which conceptually belong together. Rather than specifying them by explicitly *enumerating* the entities belonging to the view, the view is defined by means of an *intension*, an executable query which, upon evaluation, results in the entities belonging to the view;

- **Constraints:** in order to document the structural source-code regularities governing source code, our model provides constructs for defining different kinds of constraints (unary constraints, binary relations, n-ary relations) which can be imposed on the intensional views that are defined over the source code of a system.

We illustrate these concepts by declaring a number of intensional views and constraints on our running example (Figure 3.2) and show how these intensional views map to the software entities from the banking system (Figure 3.3).

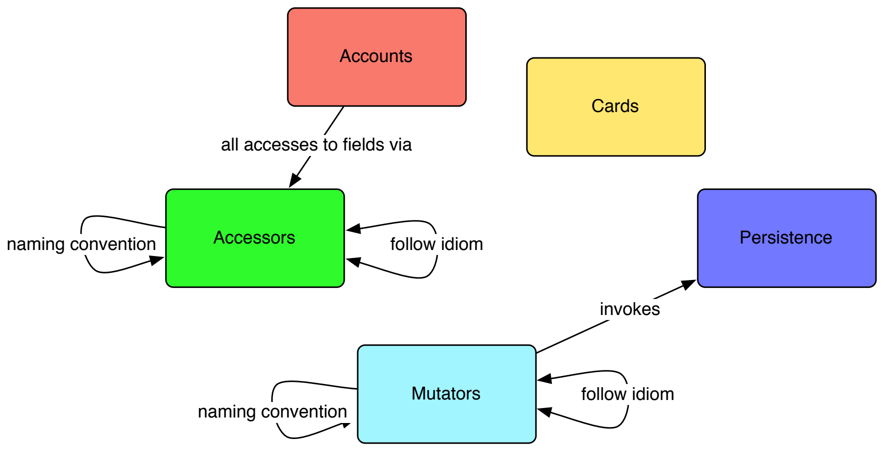


Figure 3.2: Example intensional views and constraints on banking system.

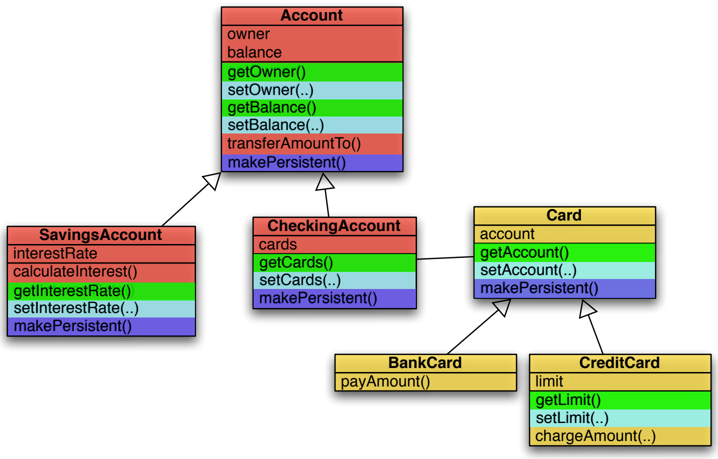


Figure 3.3: Mapping of the intensional views and constraints onto the running example. The colors match those of the intensional views shown in Figure 3.2.

We define five intensional views on our banking system:

- **Accounts:** this intensional view groups all subclasses of `Account`;
- **Cards:** contains all subclasses of `Card`;
- **Accessors:** this intensional view captures all accessor methods by e.g. grouping all methods starting with “get-”;
- **Mutators:** all methods which alter the value of a field, i.e. all methods starting with the prefix “set-”;
- **Persistence:** all methods which save the state of an object to the database.

Note that these intensional views align with domain concerns of the system (the Accounts and Card views) as well as with more implementation-oriented concerns such as accessors, mutators and persistence. The example also contains a number of structural source-code regularities which we express by means of constraints over the intensional views:

- In addition to the convention that accessor and mutator methods respectively start with the prefix “get-” or “set-”, these methods are also characterized by an implementation idiom. Accessor methods consist of a single statement which returns the value of the field; mutator methods must contain an assignment to a field;
- Moreover, accessor and mutator methods also have to respect a naming convention which states that the name of the field they are accessing/mutating is part of the method name. This regularity and the regularity above are documented using *unary constraints*. Such constraints impose a condition which is applicable to the entities belonging to a single intensional view;
- In order for the system to be in a consistent state, all changes to the state must be followed by a call to the persistence mechanism which registers the changes in the database. To express such a design requirement, our model contains the notion of *binary relations*. For instance, in the example we have a binary relation which expresses that all entities belonging to the `Mutators` intensional view must contain a call to an entity in the `Persistence` view.

3.2 Notations and conventions

We start the formal specification of the model of intensional views by presenting the notations and conventions which we will be using throughout this chapter.

3.2.1 Representation of concepts

The concepts we introduce in this chapter are represented by a tuple grouping the different components of the concepts, together with a number of domain constraints, well-formedness

Concept: Function fun	
Components:	**Domain constraints:**
$fun = (f, Var, body)$	$f \in Identifier$ $Var \subseteq Variables$ $body \in MathExpr$
Well-formedness constraints:	**Shortcuts for selector functions:**
$variables(body) \subseteq Var$	$fun_f = f$ $fun_{var} = Var$ $fun_{body} = body$

Figure 3.4: Example of our representation of concepts, applied to mathematical functions

criteria and auxiliary functions. To illustrate our notation, consider the concept of a 'mathematical function' (e.g. $f(x, y) = x^2 + y$). Such a function consists of three components namely the function symbol f, a set of variables $\{x, y\}$, and the function body $x^2 + y$. Figure 3.4 shows how this concept of a mathematical function is expressed using our notation. It consists of five parts:

- **Concept:** The name of the concept we are defining (in the example a Function fun);
- **Components:** A tuple containing the components which the concept consists of. In our example we have a tuple named fun containing a function symbol f, a set of variables Var and a function body $body$;
- **Domain constraints:** Constraints which indicate which domain the components of the concept belong to. In our example we require f to be an identifier, Var a subset of the domain of all variables and $body$ an element of the set of mathematical expressions;
- **Well-formedness constraints:** Additional constraints an instance of the concept needs to adhere to in order to be a valid instance. For instance, for a function to be valid we require that all variables in the function body are part of Var;
- **Shortcuts for selector functions:** A number of auxiliary functions which take an instance of the concept as input and return one of the components. E.g. the function fun_{body} returns the function body component of a function fun.

3.2.2 Representation of a population using tuples

The population of an intensional view – i.e. the entities which belong to a view – are represented by means of n-tuples. Each n-tuple consists of n artifacts which belong together and

{(**class**: `Account`,	**method**: `getOwner`,	**field**: `owner`),
(**class**: `Account`,	**method**: `getBalance`,	**field**: `balance`),
(**class**: `SavingsAccount`,	**method**: `getInterestRate`,	**field**: `interestRate`),
(**class**: `CheckingAccount`,	**method**: `getCards`,	**field**: `cards`),
(**class**: `Card`,	**method**: `getAccount`,	**field**: `account`),
(**class**: `CreditCard`,	**method**: `getlimit`,	**field**: `limit`)}

Figure 3.5: The set of tuples belonging to the *accessor methods* view

describe an element of a view. For instance, a number of examples of 2-tuples are:

$$(\text{``}Harry\text{''}, 2500)$$
$$(\text{``}Frank\text{''}, 1900)$$
$$(\text{``}Sally\text{''}, 2300)$$

These tuples represent for example an association of the name of a person with the salary of that person. In classic set-theory, the order in which the artifacts appear in a tuple is important. For instance, the tuple $(\text{``}Harry\text{''}, 250)$ is different from the tuple $(250, \text{``}Harry\text{''})$. In order to improve readability of the tuples, we opt to use "named" n-tuples which contain tags mapping a certain attribute of a tuple to a value, similar to the way the population of databases are often described. In this notation, we can express our example 2-tuples as follows:

$$(\textbf{name:}\text{``}Harry\text{''}, \textbf{salary:}2500)$$
$$(\textbf{name:}\text{``}Frank\text{''}, \textbf{salary:}1900)$$
$$(\textbf{name:}\text{``}Sally\text{''}, \textbf{salary:}2300)$$

Using this notation, the order of the artifacts in the tuples no longer important. For instance, the first tuple $(\textbf{name:}\text{``}Harry\text{''}, \textbf{salary:}2500)$ and the tuple $(\textbf{salary:}2500, \textbf{name:}\text{``}Harry\text{''})$ are considered to be identical.

3.3 Intensional Views

In this section we introduce the model of intensional views. We start the section by introducing two concepts which lie at the heart of intensional views, namely the notions of *intension* and *extension* of a view.

3.3.1 The extension of an intensional view

At the heart of our model lies the concept of an *intensional view*. In a spirit similar to the software classification mechanisms which we discussed in Chapter 2, such as for instance the Concern Manipulation Environment (CME) by Harrison et al. [HOST05a] or the work of Koen De Hondt [DH98], intensional views are a classification mechanism which group related software entities such classes, methods, variables, packages, etc.

The model of intensional views is independent of the underlying (representation of the) software entities that are classified by intensional views. As such, an intensional view can group any number of classes, variables, methods, signatures, sequence diagrams, objects, and so on which are conceptually related. These software entities are represented by an n-tuple. The set of all n-tuples that belong to an intensional view is called the *extension* of that intensional view.

For example, in our running example we encountered an intensional view named *Accessors* which groups a representation of all accessor methods in the banking system. Each accessor method is represented by a 3-tuple that combines the class, the method and the name of the field that the method is accessing. Applied to the running example, the set of all tuples for the *accessor methods* view is shown in Figure 3.5.

Each 3-tuple consists of a mapping from an attribute name ('class', 'method' or 'field') to a software entity. For instance, the first tuple we see in Figure 3.5 associates the attribute 'class' to the `Account` class [1], the attribute 'method' to a method named `getOwner` and the property 'field' to the value `owner`.

Definition 1. *More formally, we say that if we denote the (finite) universe of all software entities in a particular software system as $\mathcal{U}$; the set of all attribute names as $\mathcal{A}$, we can then define an n-tuple t with attributes A over the universe $\mathcal{U}$ as the following function:*

$$t_A : \mathcal{A} \mapsto \mathcal{U}$$

This function t_A is a partial function that maps a particular attribute to a value of the universe $\mathcal{U}$ if this attribute is part of A. For extracting the value for a certain attribute a from a tuple t_A, we utilize the following notation:$t.attribute$.

For example, suppose we have the tuple $t_1 = ($**name:**$\text{“}Harry\text{”},$**salary:**$2500)$, then $t_1.name = \text{“}Harry\text{”}$ and $t_1.salary = 2500$. Moreover, we define the arity of an n-tuple t as $arity(t) = n$.

We thus require from each n-tuple that it associates with every attribute a that is part of the attributes A of the tuple a value from the universe $\mathcal{U}$; for all other attributes, such an association does not exist.

Definition 2. *We denote the set of all n-tuples with attributes A as $\mathcal{T}_A$.*

Definition 3. *We define the extension Ext of an intensional view with respect to a set of attributes A as a subset of all possible tuples with attributes A.*

$$Ext_A \subseteq 2^{\mathcal{T}_A}$$

3.3.2 The intension an intensional view

Now that we have defined what the extension of an intensional view – the set of tuples belonging to that view– looks like, we are going to take a look at how to specify such a set in our model. Mathematically, there are two ways of specifying a set: extensionally or intensionally.

[1]Depending on the underlying representation of the software entities, the association of an attribute may be the actual software entity or an identifier representing that entity

A set is defined extensionally by explicitly enumerating all elements in the set. For example, we can define the set E of all even numbers extensionally as follows:

$$E = \{0, 2, 4, 6, 8, \ldots\}$$

However, the same set can also be specified intensionally by means of a description:

$$E = \{2x \mid x \in \mathbb{N}\}$$

In our model we opted for the latter approach. The set of tuples belonging to an intensional view is specified by means of an *intension*: an executable query which, upon evaluation, yields the tuples belonging to that view, i.e. the extension of the intensional view. Since the model of intensional views is independent of the underlying query language used to define the intension, we will not go into more detail about the query language here. In Chapter 4 we take a look at a number of query languages which our implementation of the intensional views model supports. However, our model does impose a number of requirements on the query language used to specify the intension.

Definition 4. *A query q_A can be considered to be a predicate that, from the set of all possible tuples with attributes A, selects a subset for which a certain condition holds. We denote the set of all queries that verify a condition for a tuple with attributes A as $\mathcal{Q}_A$. As such, we specify:*

$$\mathcal{Q}_A = \mathcal{T}_A \rightarrow Boolean$$

In what follows, we assume the existence of a function $eval_A$ that given as input a query q_A will result in all the tuples from $\mathcal{T}_A$ for which the query holds. The goal of this function is to provide a means to evaluate the intension of an intensional view and obtaining the set of n-tuples that are captured by this intension. While the actual semantics of this function $eval_A$ depend on the underlying query language used to specify the intension q_A, we can intuitively define this function as:

Definition 5. *We define this function $eval_A$ as:*

$$eval_A : \mathcal{Q}_A \rightarrow 2^{\mathcal{T}_A} : q \rightarrow \{t \in \mathcal{T}_A | q(t)\}$$

When evaluating this function for a particular intension q, it will restrict the set of all possible tuples with attributes A to those which the intension q holds for. For a particular instantiation of the model of intensional views using a concrete query language, an $eval_A$ function must be provided that reflects the semantics of this query language. We repeat that we deliberately do not define the actual semantics of this $eval_A$ function, because we are only formalizing the model of intensional views, and this model is independent of the actual query language being used (e.g. we can use a query language based on predicate logic, on regular expressions, etc.). However we impose the requirement on such query languages namely that the evaluation of queries in the chosen query language satisfies the definition given above. For that reason, in the examples that follow, we express the queries in natural language. In Chapter 4 we give examples of queries written in a Prolog-like query language.

To illustrate such an intension of an intensional view, consider the following example:

Concept: Intensional View V	
Components:	**Domain constraints:**
$V = (Attr, query, Parents, Incl, Excl)$	$Attr \subseteq \mathcal{A}$ $query \in \mathcal{Q}_{Attr}$ $Parents \subseteq \mathcal{V}$ $Incl, Excl \subseteq \mathcal{T}_{Attr}$
Well-formedness constraints:	**Shortcuts for selector functions:**
$V \notin Parents$ $Incl \cap eval(V) = \emptyset$ $Excl \subseteq eval(V)$	$V_{Attr} = Attr$ $V_{query} = query$ $V_{Parents} = Parents$ $V_{Incl} = Incl$ $V_{Excl} = Excl$

Figure 3.6: Definition of intensional view V

Example 1.

$A = \{\text{class}, \text{method}, \text{field}\}$

$q =$ *"the **method** starts with the prefix get-, is implemented in a specific **class**, and returns the value of a specific **field**"*

This example shows a possible intension (in natural language) which can be used to define the set of tuples belonging to the *Accessors* intensional view. Suppose we have an evaluation function $eval_A$ for such a natural language query, then the result of applying that $eval_A$ function to the above intension will result in the set of tuples (this set is shown in Figure 3.5) which associates each of the attributes ('class', 'method' and 'field') to a corresponding software entity.

3.3.3 Definition of an intensional view

Now that we have introduced the notions of *extension* and *intension* of an intensional view, we can give a definition of the concept *intensional view*. Note that this definition focusses on defining the **form** of an intensional view. In a later section we will discuss the **semantics** of such a view. We present the concept of an intensional view in a top-down manner: we first introduce the concept as a whole and will then go into detail about the parts of the definition and the rationale behind these parts.

With $\mathcal{V}$ the set of all intensional views, the definition of an intensional view V can be found in Figure 3.6. In our definition, an intensional view V consists of five components:

- $Attr$: a set of attributes;
- $query$: a query, i.e. an intension, that is part of $\mathcal{Q}_{Attr}$;
- $Parents$: a set of parent views;
- $Incl$: a set of tuples which are to be explicitly included in the extension of an intensional view;
- $Excl$: a set of tuples which are to be explicitly excluded from the extension of an intensional view.

In the previous sections we already discussed the relation between a set of attributes $Attr$ and an n-tuple and how a query $query$ can be used as an intensional way to define the set of tuples belonging to a view V. In what follows, we highlight the three other components of an intensional view, namely the set of parent views $Parents$ and the sets $Incl$ and $Excl$ which serve to explicitly document deviations from the intension of an intensional view.

Parent views of an intensional view

The model of intensional views provides support for declaring a set of parent views $Parents$ for a given view V. The rationale behind appointing a number of parent views to an intensional view is two-fold. First, parent views provide a kind of scoping mechanism for restricting the number of software entities which an intensional view is applicable to. Second, the use of parent views makes it possible to make intensional views more reusable: if an intensional view encodes an abstract description of a concern which is applicable to numerous software systems, this view can be reused in a different context by altering the set of parent views of the intensional view.

As can be seen in the domain constraints of our definition of an intensional view, we require this set $Parents$ to be a subset of all intensional views $\mathcal{V}$. According to our well-formedness constraints, a intensional view cannot be part of its own parent views, as this would result in a circular definition when defining the semantics of an intensional view later on.

Explicit deviations from an intension

The $Incl$ and $Excl$ components of an intensional view serve as a means to document explicit deviations from the intension of a view. Although the majority of software entities belonging to an intensional view exactly match the intension, the intension of the view is in some cases either too general or too specific. As a result, the extension of the view does not contain the expected set of tuples.

- If the intension is too specific, there are certain tuples which should belong to the extension of the view but which are not calculated by evaluating the intension. These can be explicitly included in the extension of the view by specifying them in the $Incl$ set;

- If the intension is too generic, evaluating the intension of the view results in a number of tuples which should not be part of the extension of the view. A developer can explicitly exclude them from the extension of an intensional view by specifying such tuples in the $Excl$ set.

We illustrate these deviations from an intension by the following example. Suppose the class `CreditCard` from our example system implements the following two methods:

```
public Integer getTotalThisMonth()
{
  total := ... //calculate the total amount used this month
  return total;
}

public Int retrieveIDnumber()
{
  return IDnumber;
}
```

The first method, `getTotalThisMonth`, computes the total amount of purchases already made with the credit card this month and returns this amount. If we execute the intension of the `Accessors` intensional view, a tuple representing this method will be part of the extension of the view, since the method follows the naming convention that all methods with as prefix "get-" are an accessor method. However, this method is not an accessor method since it does not return the value of a field and should thus not be included in the extension of the `Accessors` view. To document this exceptional case, a developer can add it explicitly to the $Excl$ set.

The second method, `retrieveIDnumber` presents an example of where the intension of the `Accessors` view is too specific. This method is conceptually an accessor method, since it returns the value of the `IDnumber` field. However, the developer who implemented this method did not respect the coding convention and used the prefix "retrieve-" instead of "get-". As such, it is not captured by the intension of the view and thus not part of the extension of the view. To explicitly include this deviation in the intensional view, a developer can add a tuple representing the `retrieveIDnumber` method to the $Incl$ set.

As specified in the domain constraints of our definition of an intensional view, both the $Incl$ set as well as the $Excl$ set are a subset of all possible tuples ($\mathcal{T}_{Attr}$). We also impose a number of additional well-formedness constraints. We require that the entities which are part of the $Incl$ set, and which are thus included in the extension of a view are not already calculated by the intension of the view. We enforce this by requiring that the intersection of $Incl$ with the tuples captured by the intension is empty. The function $eval(V)$ which we use to calculate the set of tuples captured by the intension is discussed in the next section. A similar well-formedness constraint is imposed on the $Excl$ set: tuples belonging to this set must be part of the set of tuples calculated by the intension.

At first sight, these well-formedness constraints might seem redundant. The extension of an intensional view is a set of tuples, thus including a specific tuple that already was present in this set does not alter the elements of the set. Likewise, excluding a tuple that was not present in the extension to begin with does not pose any problems. However, we include these well-formedness constraints since they allow us to identify an interesting evolution conflict. Suppose we have an intensional view V that is defined on a system. In this intensional view, the tuple t_1 is documented as a belonging to the $Incl$ set. If now upon evolution, the tuple t_1 becomes part of the set of tuples calculated by the $query$ of the intensional view V, it is interesting to know that changes to the system resulted in that a tuple that used to be a deviation from the general rule is now captured by that general rule.

Example of an intensional view

We conclude our section describing the definition of an intensional view by means of an example. We can now give a definition of the `Accessors` intensional view, as we have described above, in the notation we introduced in this section.

Example 2.

Accessors $= (Attr, query, Parents, Incl, Excl)$ *where*
Attr $= \{$*'class'*, *'method'*, *'field'*$\}$,
$query =$ *"the **method** starts with the prefix get-, is implemented in a specific **class**, and returns the value of a specific **field**"*,
$Parents = \{$*Accounts*$\}$,
$Incl = \{($***class:***`CreditCard`, ***method:***`retrieveIDnumber`, ***field:***`IDnumber`$)\}$,
$Excl = \{($***class:***`CreditCard`, ***method:***`getTotalThisMonth`, ***field:***`total`$)\}$

The above notation describes the intensional view *Accessors* with three attributes namely 'class', 'method', and 'field'. The intension $query$ of the intensional view captures the naming convention that the method name of all accessor methods must start with a prefix "get-". Furthermore, we provide a single parent view *Accounts* for the *Accessors* view and document the two deviations to the intension of the view which we already described in the section above.

3.3.4 Semantics of an intensional view

In the section above we defined the structure of an intensional view. In this section we take a look at the semantics of such an intensional view. We define the semantics of an intensional view by means of a function $extension(V)$, which for a given view V computes the set of tuples that belong to the extension of this view. This $extension$ function is not limited to evaluating the intension of an intensional view but also has to take the parent views of an intensional view and the deviations to the intension into account.

Before we define the $extension$ function, we first introduce a function $eval(V)$ which evaluates the intension of an intensional view V in the correct scope (i.e. taking the parent views of V into account). Notice that this $eval$ function is the one we also used in the well-formedness constraints of the $Incl$ and $Excl$ sets in Section 3.3.3.

We illustrate the semantics of parent views by means of an example. Consider the *Accessors* intensional view which we encountered earlier. If there are no parent views specified for this view, it is scoped over the entire system. As such, the extension of *Accessors* is the one specified in Figure 3.5. If however, we are only interested in grouping the accessor methods implemented on classes in the `Account` hierarchy, we select the *Accounts* intensional view as the parent view of the *Accessors* view. Suppose the *Accounts* view contains the following 2-tuples:

{(**class**: `Account`,	**method**: `getOwner`)
(**class**: `Account`,	**method**: `getBalance`)
(**class**: `Account`,	**method**: `setOwner`)
(**class**: `Account`,	**method**: `setBalance`)
(**class**: `Account`,	**method**: `transferAmount`)
(**class**: `Account`,	**method**: `makePersistent`)
(**class**: `SavingsAccount`,	**method**: `getInterestRate`)
(**class**: `SavingsAccount`,	**method**: `setInterestRate`)
(**class**: `SavingsAccount`,	**method**: `calculateInterest`)
(**class**: `SavingsAccount`,	**method**: `makePersistent`)
(**class**: `CheckingAccount`,	**method**: `setCards`)
(**class**: `CheckingAccount`,	**method**: `getCards`)
(**class**: `CheckingAccount`,	**method**: `makePersistent`) }

These tuples group all pairs of classes and methods which belong to a class in the `Account` hierarchy. By considering the *Accounts* view as the parent view of the *Accessors* view, we no longer obtain the set of tuples as shown in Figure 3.5 but rather only select the accessor methods in the `Account` hierarchy:

{(**class**: `Account`,	**method**: `getOwner`,	**field**: `owner`),
(**class**: `Account`,	**method**: `getBalance`,	**field**: `balance`),
(**class**: `SavingsAccount`,	**method**: `getInterestRate`,	**field**: `interestRate`),
(**class**: `CheckingAccount`,	**method**: `getCards`,	**field**: `cards`)}

The tuples in the extensions of the parent views serve as a sort of filter[2] for the tuples in the extension of an intensional view. Much like a select statement in tuple calculus, only tuples are considered to be part of the extension of a view if there exists a tuple in *all* of the parent views such that, for all the attributes the intensional view and its parent views have in common, the tuples in the parent view and the tuples in the extension of the intensional view associate the common attributes with identical values. For instance in our example, the tuples of the *Accessors* view have two attributes in common with the tuples belonging to the *Accounts* view, namely 'class' and 'method'. We thus restrict the tuples belonging to the *Accessors* view to those tuples t for which there exists a tuple t' in the *Accounts* intensional view such that $t.class = t'.class$ and $t.method = t'.method$.

We formalize these semantics by defining the function $eval(V)$ as:

[2]Although parent views can intuitively be interpreted as filters on the tuples of an intensional view, this is not necessarily the way they need to be implemented.

Definition 6.

$$eval(V) = \left\{ \begin{array}{l} \{t|\ t\ \in\ eval_{V_{Attr}}(V_{query}) \\ \quad \wedge\ \forall\, parent\ \in\ V_{Parents} : \\ \qquad (\exists\, t' \in extension(parent) : \\ \qquad\quad (\forall\ a\ \in (V_{Attr} \cap parent_{Attr}) : \\ \qquad\qquad t.a = t'.a))\} \end{array} \right.$$

The result of $eval(V)$ consists of all tuples t which are obtained by evaluating the intension V_{query} of V, but for which in the extension of each of the parent views $V_{Parents}$ there exists a tuple t' such that, for *all* attributes which are shared by the view V and the parent view $parent$, the value associated by both the tuple t as well as the tuple t' is the same. Note that, if the intensional view V does not have any parent views, the result of $eval(V)$ will equal the set of tuples obtained by evaluating the intension V_{query}.

Based on the $eval(V)$ function which returns the set of tuples computed by executing the intension of a view V with respect to the parent views of that view, we can now provide a definition for the $extension(V)$ function.

Definition 7.

$$extension(V) = (eval(V)\ \cup\ V_{Incl}) \setminus V_{Excl}$$

The $extension(V)$ function returns the set of tuples calculated by the $eval(V)$ function while including all the tuples belonging to the $Incl$ set of V. Moreover, all tuples belonging to the $Excl$ set are omitted from the result.

Notice that in the definition of the $eval(V)$ function, in the second case we use this $extension(V)$ function to calculate the set of tuples belonging to the extension of the parent views of an intensional view V. This $extension(V)$ function is used rather than recursively calling the $eval(V)$ function, since it also takes the $Incl$ and $Excl$ sets of the parent views into account. The $eval(V)$ and $extension(V)$ functions are mutually recursive: in order to calculate the extension of an intensional view, the $eval(V)$ function is invoked which on its turn uses the $extension(V)$ function to retrieve the set of tuples belonging to the parent views of the intensional view. This recursive process is repeated until the top-most view in the hierarchy is reached, in which case there are no parent views that need to be taken into account and the set of tuples can be calculated by evaluating the intension of this top-most view.

If we apply the $extension(V)$ function to the *Accessors* intensional view which we defined in the previous section, we get as a result:

extension(*Accessors*)=

{(**class**: `Account`,	**method**: `getOwner`,	**field**: `owner`),
(**class**: `Account`,	**method**: `getBalance`,	**field**: `balance`),
(**class**: `SavingsAccount`,	**method**: `getInterestRate`,	**field**: `interestRate`),
(**class**: `CheckingAccount`,	**method**: `getCards`,	**field**: `cards`)}

Concept: Unary constraint U	
Components: $U = (view, quant, pred, Incl, Excl)$	**Domain constraints:** $\begin{aligned} view \in & \quad \mathcal{V} \\ quant \in & \quad \mathcal{Q}uant_A \\ pred \in & \quad \mathcal{P}red_A \\ Incl, Excl \subseteq & \quad \mathcal{T}_A \\ \text{where} & \quad A = view_{Attr} \end{aligned}$
Well-formedness constraints: $\begin{aligned} Incl, Excl \subseteq & \quad extension(view) \\ \nexists t \in & \quad Incl : pred(t) \\ \forall t \in & \quad Excl : pred(t) \end{aligned}$	**Shortcuts for selector functions:** $\begin{aligned} U_{view} = & \quad view \\ U_{quant} = & \quad quant \\ U_{pred} = & \quad pred \\ U_{Incl} = & \quad Incl \\ U_{Excl} = & \quad Excl \end{aligned}$

Figure 3.7: Definition of a unary constraint U

3.4 Constraints on intensional views

As we have discussed in the beginning of this chapter, our model consists of two types of entities: intensional views and constraints over these intensional views. In this section, we introduce the different kinds of constraints which can be imposed on an intensional view. More concretely, we present unary constraints, binary relations and n-ary relations.

3.4.1 Unary intensional constraints

Definition of a unary constraint

A unary intensional constraint is a constraint imposed on a single intensional view. In our running example we already encountered a number of examples of regularities which can be expressed using a unary intensional constraint. For instance, we mentioned the regularity that all accessor methods must contain the name of the field they are accessing. As an illustration of this regularity, we require for instance that the accessor method for the `cards` variable contains the string "cards" in its name.
More formally, this constraint expresses that:

$$\forall\, t \in extension(Accessors) : methodNameContains(t.method, t.field)$$

where $methodNameContains$ is a predicate verifying whether a method name contains a specific string. The definition of a unary constraint U can be found in Figure 3.7. We define the set of all unary constraints as $\mathcal{C}_1$. A unary constraint U consists of five components:

- $view$: the intensional view which the unary constraint applies to;
- $quant$: a quantifier (e.g. $\forall$, $\exists$, $\nexists$) which selects the extent of the tuples in the view v which the condition needs to apply to;
- $pred$: a unary predicate expressing a condition over a tuple;
- $Incl$: a set of tuples for which we explicitly assert that the condition holds;
- $Excl$: a set of tuples for which we explicitly assert that the condition does not hold.

Quantifier $quant$ The quantifier $quant$ quantifies the range of tuples in the extension of the view which the condition must hold for. For instance, we can require that the constraint holds for all tuples in the intensional view, for none, for exactly one, and so on. Mathematically, we present a quantifier $quant$ as a function taking as input a set of tuples T and a unary predicate $pred$ that expresses a condition over the tuples. The quantifier returns a boolean depending on whether or not the condition holds for the quantified set of tuples.

Definition 8. *We thus get:*

$$\begin{aligned}&\mathcal{Q}uant_A = 2^{\mathcal{T}_A} \times \mathcal{P}red_A \;\rightarrow\; Boolean\\ &\textit{where } A \subseteq \mathcal{A}\end{aligned}$$

Notice that both the set of tuples serving as input to the quantifier and the predicate of the quantifier are restricted to the same set of attributes A.

Although certainly not exhaustive, the list of quantifiers we support in our model and the condition which they must fulfill in order to yield true is:

Definition 9 (Predefined quantifiers)**.**

$$\begin{aligned}\forall(T,p) &\iff \forall\, t \in T : p(t)\\ \exists(T,p) &\iff \exists\, t \in T : p(t)\\ \exists!(T,p) &\iff \exists!\, t \in T : p(t)\\ \nexists(T,p) &\iff \nexists\, t \in T : p(t)\\ few(T,p) &\iff |\{t|t \in T \wedge p(t)\}| \leq 0.25|T|\\ some(T,p) &\iff 0.25|T| \leq |\{t|t \in T \wedge p(t)\}| \leq 0.50|T|\\ many(T,p) &\iff 0.50|T| \leq |\{t|t \in T \wedge p(t)\}| \leq 0.75|T|\\ most(T,p) &\iff |\{t|t \in T \wedge p(t)\}| \geq 0.75|T|\end{aligned}$$

The first four quantifiers are a literal translation of the quantifiers used in classic tuple calculus and their implementation in our model as a function serves to translate their semantics to the model of intensional constraints. The last four quantifiers provide a more “fuzzy” means to quantify over the tuples of a view for which a constraint must hold. The few quantifier holds if the predicate p is true for at most 25% of the set of tuples T; $some$ holds if the predicate is true for between 25% and 50% of the tuples; $many$ holds if the predicate holds for between 50% and 75% of the tuples; $most$ holds if the constraint holds for at least 75% of the tuples.

Strictly speaking, we would have to define all the above quantifiers dependent on the set of attributes A. For example, the *forall* quantifier applicable to a set of tuples with as attributes A would be denoted $\forall_A$. Since the definition of the quantifiers however remains identical, independent of the set of attributes A, we omit this prefix A.

Unary predicate $pred$ The unary predicate $pred$ expresses the condition which must be satisfied by all tuples with as attributes A, quantified by quantifier $quant$. Similar to our definition of intensional views, we do not focus here any concrete language in which the predicate is expressed. Instead, we explain the requirements such a predicate $pred$ must satisfy. We require a predicate $pred$ to be a function taking one tuple with attributes A as an argument and returning a boolean, depending on whether the predicate holds for that tuple:

Definition 10. *We denote the set of all unary predicates over tuples with as attributes A as $\mathcal{P}red_A$:*

$$\mathcal{P}red = \mathcal{T}_A \rightarrow Boolean$$

In Chapter 4 we take a look at actual languages for expressing the predicates used when imposing constraints over intensional views.

Exceptions on unary constraints $Incl$ **and** $Excl$ Similarly to the concept of deviations we introduced in intensional views, it is possible to explicitly assert exceptions on unary constraints. To this end, our model of unary constraints contains the $Incl$ and $Excl$ sets. The $Incl$ set contains a number of tuples the constraints explicitly holds for. $Excl$ is used to specify a set of tuples which the constraint does not hold for.

The $Incl$ and $Excl$ sets impose a number of well-formedness constraints. Both sets must contain tuples which are part of the extension of the intensional view $view$. Moreover, the predicate $pred$ must not hold for any tuple t in $Incl$. If this predicate $pred$ would already hold for the tuple t, then it would be redundant to assert that the constraint holds explicitly for t. Similarly, the predicate $pred$ must hold for all tuples which belong to $Excl$. Were this not the case, it would not make sense to explicitly assert that the relation does not hold for t.

Example Using the above definition of a unary intensional constraint, we can now revisit our example constraint U, expressing that the names of all accessor methods must contain the name of the field they are accessing:

Example 3.

$$\begin{aligned}
&U = (view, quant, pred, Incl, Excl) \textit{ where}\\
&view = \textit{ Accessors},\\
&quant = \forall,\\
&pred: t \rightarrow \textit{methodNameContains(t.method, t.field)},\\
&Incl, Excl = \emptyset
\end{aligned}$$

Semantics of a unary intensional constraint

The semantics we associate with the definition of a unary constraint U consists of a number of functions that verify conformance of the constraint with the tuples which belong to the extension of the intensional view which the constraint is imposed over. More precisely, we specify two functions:

- $consistent(U)$: this predicate verifies whether or not the constraint holds for all of the tuples of the extension of the intensional view, taking the quantifier and the deviations to the constraint into account;
- $discrepancies(U)$: returns the set of tuples for which the constraint does not hold.

The $consistent(U)$ predicate checks whether the relation is upheld. We define this predicate as:

Definition 11. *For a unary constraint* $U \in \mathcal{C}_1$*:*
Let $A = (U_{view})_{Attr}$ *then*

$$\begin{aligned}
consistent(U) \iff &U_{quant}(extension(U_{view}), p)\\
&\textit{where } p: \mathcal{T}_A \rightarrow Boolean: (U_{pred}(t) \ \lor \ t \in U_{Incl}) \ \land \ t \notin U_{Excl}
\end{aligned}$$

This predicate requires a bit of an explanation. The notion of consistency of a unary constraint is based on the fact that quantifiers are functions that take two arguments, namely a set of tuples and a constraint. The quantifier returns a boolean depending on whether the constraint holds for the correct quantity of tuples as specified by the quantifier. We use this definition of quantifiers in the $consistent(U)$ function by applying the quantifier to the tuples in the $extension$ of the intensional view which the constraint is imposed on. The condition p that is verified for the tuples of the extension of the intensional view expresses that for each tuple t either the predicate U_{pred} of the unary constraint must hold or the tuple t must be explicitly declared as an exception which the relation holds for (tuple t must thus be part of U_{Incl}). Moreover, tuple t must not be explicitly excluded from the relation, so it must not be part of U_{Excl}.

While the $consistent(U)$ function can give us an answer whether the unary constraint U is violated or not, it does not provide any information concerning the tuples for which the constraint does *not* hold. In order to retrieve this set of discrepancies, we specify a function $discrepancies(U)$ which, given a unary intensional constraint U returns the set of tuples

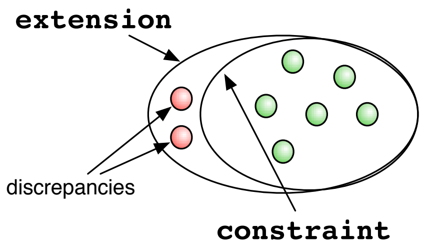


Figure 3.8: Inconsistencies of a unary intensional constraint

which the constraint does not hold for. The intuition behind this function is shown in Figure 3.8. This figure shows a Venn diagram containing two sets. The first set represents the tuples belonging to the extension of the view on which the constraint is imposed; the second set contains the tuples for which the the unary constraint U holds. The intersection of both sets contains all tuples of the extension for which the constraint is satisfied. However, all tuples which belong to the extension of the view, but which are not covered by the constraint, are considered discrepancies.

We define $discrepancies(U)$ as:

Definition 12. *For a unary constraint $U \in \mathcal{C}_1$:*

$$discrepancies(U) = \begin{cases} \{t \in extension(U_{view}) | \\ \quad (U_{pred}(t) \vee t \in U_{Incl}) \wedge t \notin U_{Excl}\} & \text{if } U_{quant} = \nexists \\ \\ \{t \in extension(U_{view}) | \\ \quad (\neg(U_{pred}(t)) \wedge t \notin U_{Incl}) \vee t \in U_{Excl}\} & \text{otherwise} \end{cases}$$

This function discriminates between two cases. The first case occurs when the quantifier used in the constraint is $\nexists$. In such case, the constraint describes a restriction by specifying a condition which must be false for all of the tuples in the intensional view. Thus, the tuples which the condition *does* hold for, are considered to be discrepancies between the constraint and the intensional view. Otherwise, the discrepancies are congruent with the situation as depicted in Figure 3.8. In that case, all tuples which are part of the extension of the view, but which the predicate U_{pred} does not hold for, are considered to be discrepancies between the constraint and the view. For both cases, when calculating this set of discrepancies we also have to take the exceptions to the constraint into account.

Note that identifying the violations of the constraint involves the expert knowledge of the developer. In order to illustrate this, suppose we define a unary constraint with as quantifier $\forall$. In this case it is clear that all tuples for which the predicate of the constraint does not hold are to be considered violations. However, if we specify a constraint with as quantifier $most$, not necessarily all discrepancies are violations of the constraint. Since it is not possible to automatically identify the violations of the constraint in this case, the developer needs to inspect the set of discrepancies.

Concept: Binary intensional relation B	
Components: $B = (view_1, view_2, quant_1, quant_2, pred, Incl, Excl)$	**Domain constraints:** $view_1, view_2 \in \mathcal{V}$; $quant_1 \in \mathcal{Q}uant_{A_1}; quant_2 \in \mathcal{Q}uant_{A_2}$; $pred \in \mathcal{P}red_{A_1 \times A_2}$; $Incl, Excl \in (\mathcal{T}_{A_1} \cup \{\_\}) \times (\mathcal{T}_{A_2} \cup \{\_\})$; where $A_1 = view1_{Attr}; A_2 = view2_{Attr}$
Well-formedness constraints: $Incl, Excl \subseteq (ext(view_1) \cup \{\_\}) \times (ext(view_2) \cup \{\_\})$; $\forall\, (t_1, t_2) \in\ Excl:\ p(t_1, t_2)$ if $t_1, t_2 \neq \_$; $\nexists\, (t_1, t_2) \in\ Incl:\ p(t_1, t_2)$ if $t_1, t_2 \neq \_$	**Shortcuts for selector functions:** $B_{view_1} = view_1; B_{view_2} = view_2$; $B_{quant_1} = quant_1; B_{quant_2} = quant_2$; $B_{pred} = pred$; $B_{Incl} = Incl; B_{Excl} = Excl$

Figure 3.9: Definition of intensional binary relations B

3.4.2 Binary intensional relations

Definition of a binary intensional relation

In Section 3.1 we introduced, in addition to the examples of unary constraints, also an example of a constraint which expresses a binary relation between two intensional views. We described a constraint between the *Mutators* and the *Persistence* views which requires that all mutator methods contain a call to a persistence method. More formally, this design regularity requires the following condition R to be true:

$$\begin{aligned} R = \forall\, t_1 \in \textit{Mutators}:& \\ \exists\, t_2 \in \textit{Persistence}:& \\ containsCall(t_1, t_2)& \end{aligned}$$

where $containsCall(t_1, t_2)$ is a binary predicate which checks whether the value of the method attribute of a tuple t_1 contains a call to the value of the method attribute of tuple t_2.

In order to express regularities like the one above in our model of intensional views, we introduce binary intensional relations as defined in Figure 3.9. The set of all binary intensional relations is denoted as $\mathcal{C}_2$ Our definition of a binary intensional view B consists of seven components:

- $view_1, view_2$: the two intensional views between which the relation is expressed;
- $quant_1, quant_2$: two quantifiers;
- $pred$: a binary predicate that takes two tuples as input: one from the domain of $\mathcal{T}_{view1_{Attr}}$ and one from the domain of $\mathcal{T}_{view2_{Attr}}$. The predicate is thus dependent on the attributes of both intensional views $view_1$ and $view_2$;
- $Incl, Excl$: two sets to explicitly document exceptions to the binary relation.

Binary predicate $pred$ Similar to unary constraints, binary intensional relations contain a predicate which must be satisfied by the set of quantified tuples. For binary relations, this predicate expresses a relation between two tuples. A binary predicate is a function which takes two n-tuples as input and results in a boolean value. We thus define the set of binary predicates $\mathcal{P}red$ as:

Definition 13. *Let* $A_1 = view1_{Attr}$; $A_2 = v2_{Attr}$

$$\mathcal{P}red_{A_1 \times A_2} = \mathcal{T}_{A_1} \times \mathcal{T}_{A_2} \rightarrow \ Boolean$$

Exceptions on binary intensional relations Similar to unary constraints, it is possible to explicitly document pairs of tuples for which the relation holds or not. Since a binary intensional relation defines a constraint between two sets of tuples, the elements belonging to the $Incl$ and $Excl$ sets must equally be pairs of tuples. Dealing with exceptions on binary relations however imposes an additional requirement. Often, a developer does not want to include/exclude a pair of tuples from the binary relation, but rather wants to express that the relation explicitly holds or fails for a certain tuple belonging to either $view_1$ or $view_2$ of the relation. To this end, we introduce the notion of a wildcard (denoted by the symbol _) which allows for documenting these kinds of exceptions. For instance, in our running example, suppose that the method `setInterestRate` on `SavingsAccount` does not make a call to a persistence method, and would thus fail the binary relation which we mentioned in the beginning of Section 3.4.2. The developer of the banking system knows that this method is an exception which does not need to call a persistence method. In order to explicitly document this, the developer can add the following pair to the $Incl$ set:

((**class:**`SavingsAccount`, **method:**`setInterestRate`, **field:**`interestRate`), _)

Notice that the meaning of _ depends on whether it is used in the $Incl$ or $Excl$:

- $Incl$: $(t_1, \_)$ indicates that for a tuple t_1, belonging to the extension of $view_1$, the binary relation explicitly holds;
- $Excl$: $(t_1, \_)$ indicates that, although the relation holds for t_1 belonging to $view_1$, all pairs of tuples with t_1 in first position must be excluded from the result of the relation.

The $Incl$ and $Excl$ of binary intensional relations are also governed by a number of well-formedness constraints. First of all, both sets must be subsets of the cross-product of the extensions of the two intensional views over which the binary relation is defined, augmented with the wildcard _. Moreover, for pairs of tuples which are part of the $Excl$ set, with the exception of those which contain a wildcard, the binary predicate $pred$ must hold; for the pairs belonging to $Incl$, with the exception of those containing a wildcard, the predicate $pred$ should not hold.

Remark that if the tuple $(\_,\_)$ is specified in the $Incl$ or $Excl$ set, this means that the binary relation respectively holds or does not hold for all tuples in the cross-product of the extensions of the two intensional views v_1 and v_2.

Using the above definitions, we can express the constraint from our running example, namely that all mutator methods must contain a call to a method that implements persistence. We implement this constraint using the following binary intensional relation B:

Example 4.

$B = (view_1, view_2, quant_1, quant_2, pred, Incl, Excl)$ *where*
$view_1 =$ *Mutators*,
$view_2 =$ *Persistence*,
$quant_1 = \forall$,
$quant_2 = \exists$,
$pred = containsCall$ *with*
    $containsCall : (t_1, t_2) \rightarrow t_1.method$ *contains call to* $t_2.method$
$Incl = \{((\textbf{\textit{class:}}\texttt{SavingsAccount}, \textbf{\textit{method:}}\texttt{setInterestRate}, \textbf{\textit{field:}}\texttt{interestRate}), \_)\}$,
$Excl = \emptyset$

This example expresses the relation discussed above, including the explicit specification of the `setInterestRate` method as an exception to the relation.

Semantics of binary intensional relations

As was the case with unary intensional constraints, the semantics of a binary intensional relation align with a set of functions which allow for the verification of the validity of the relation and the identification of the tuples for which the relation does not hold. We thus also present a predicate $consistent(B)$ which verifies whether or not the relation is satisfied and a function $discrepancies(B)$ which returns the set of tuples of the intensional views which the relation is imposed over, for which the relation does not hold.

We start by explaining the $consistent(B)$ function. Recall that we introduced a wildcard mechanism for documenting explicit exceptions to a binary relation. As such, when verifying conformance of a binary intensional relation B, we cannot merely take the exceptions into account by checking whether a certain pair of tuples (t_1, t_2) is a member of either the $Incl$ or $Excl$ set. To solve this problem, we introduce a function $match$. This function takes as input a pair of tuples (t_1, t_2) together with a set of pairs of tuples S (these are the elements of $Incl$

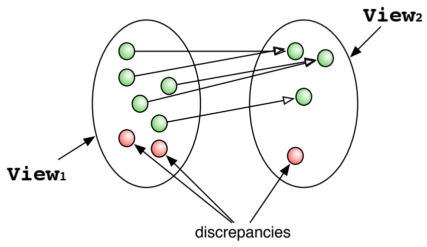


Figure 3.10: Inconsistencies of a binary intensional relation

or $Excl$) and checks whether S contains (t_1, t_2) or if (t_1, t_2) is matched by a pair of tuples belonging to S that contain a wildcard.

Definition 14.

$$match : (\mathcal{T}_{A_1} \times \mathcal{T}_{A_2}) \times 2^{(\mathcal{T}_{A_1} \cup \{\_\}) \times (\mathcal{T}_{A_2} \cup \{\_\})} \rightarrow Boolean :$$
$$(t_1, t_2) \times S \rightarrow (t_1, t_2) \in S \vee (t_1, \_) \in S \vee (\_, t_2) \in S \vee (\_, \_) \in S$$

$match$ thus checks whether (t_1, t_2) is part of the set S or whether there exist a tuple in S containing wildcards such that t_1 and/or t_2 match a wildcard.

Using the $match$ function we described above, we can define the predicate $consistent(U)$ as:

Definition 15. *For a binary relation $B \in \mathcal{C}_2$:*
Let $A_1 = (B_{view_1})_{Attr}$; $A_2 = (B_{view_2})_{Attr}$

$$consistent(B) \iff B_{quant_1}(extension(B_{view_1}), B_{quant_2}(extension(B_{view_2}), p))$$
$$\textit{with } p : (\mathcal{T}_{A_1} \times \mathcal{T}_{A_2}) \rightarrow Boolean :$$
$$(t_1, t_2) \rightarrow (B_{pred}(t_1, t_2) \vee match((t_1, t_2), B_{Incl}) \wedge \neg(match((t_1, t_2)), B_{Excl}))$$

This predicate uses a similar mechanism as the $consistent(U)$ predicate which we defined to verify conformance of a unary constraint. However, the consistent predicate for binary relations nests the two quantifiers: the second quantifier is used as the constraint which the tuples selected by the first quantifier must adhere to. The second quantifier makes use of the condition p which holds if a pair of tuples (t_1, t_2) either satisfies the binary predicate B_{pred} of the binary relation or is matched by a pair of tuples from the B_{Incl} set. Moreover, p requires that the pair of tuples (t_1, t_2) is not matched by any pair belonging to B_{Excl}. We repeat again that we use the $match$ function in order to deal with exceptions to the binary intensional relation which make use of the wildcard mechanism.

Similarly to unary constraints, we can also compute the set of pairs of tuples which are the discrepancies between the the binary intensional relation B and the entities in the intensional views $view_1$ and $view_2$. An illustration of what we consider to be discrepancies against a binary relation is shown in Figure 3.10. This figure displays two Venn diagrams each representing the extension of an intensional view. The arrows between the elements of the Venn

diagrams represent pairs of tuples which the binary relation holds for. We specify the set of possible inconsistencies as those tuples belonging to the cross-product of the extensions of both views for which the relation does not hold.
More formally, we can compute this set of pairs using the $discrepancies(B)$ function:

Definition 16. *For a binary relation $B \in \mathcal{C}_2$:*

$$discrepancies(B) = \begin{cases} \{(t_1, t_2) \\ \quad \in (extension(B_{view_1}) \times extension(B_{view_2}))| \\ \quad\quad (B_{pred}(t_1, t_2) \vee match((t_1, t_2), B_{Incl})) \\ \quad\quad \wedge \neg match((t_1, t_2), B_{Excl})\} & \textit{if } B_{quant_i} = \nexists \\ \\ \{(t_1, t_2) \\ \quad \in (extension(B_{view_1}) \times extension(B_{view_2}))| \\ \quad\quad (\neg(B_{pred}(t_1, t_2)) \wedge \neg match((t_1, t_2), B_{Incl})) \\ \quad\quad \vee match((t_1, t_2), B_{Excl})\} & \textit{otherwise} \end{cases}$$

The above definition makes, similarly to the function which calculates the inconsistencies of unary constraints, a distinction between whether the binary relation uses a $\nexists$ quantifier or not, and based on this information returns the set of pairs of tuples which the relation is not satisfied for. If the binary relation implements a restriction, i.e. if it contains a $\nexists$ quantifier, then all tuples for which the predicate of the relation holds (also taking exceptions into account) can be considered to be discrepancies between the relation and the intensional views on which the relation is imposed. Conversely, for all other combinations of quantifiers, the pairs of tuples which result in that the relation fails are contained in the pairs of tuples for which the predicate of the relation does not hold.

In practice, a developer is not interested in knowing all the pairs of tuples which the relation does not hold for, but rather wants to know the tuples that are discrepancies for either intensional view B_{view_1} or B_{view_2}. In order to retrieve this information, we present two functions named $discrepancies_{view_1}(B)$ and $discrepancies_{view_2}(B)$ which respectively return the tuples belonging to B_{view_1} or B_{view_2} which the relation does not hold for.

Definition 17.

$$discrepancies_{view_1}(B) = \Pi_1(discrepancies(B))$$
$$discrepancies_{view_2}(B) = \Pi_2(discrepancies(B))$$

Where Π_i is a function which takes as input a set of pairs and returns the projection of the i-th element of each pair. For example, if we apply Π_1 to the set of pairs $\{(a, b), (b, c), (c, d), (c, e)\}$ the result is the set $\{a, b, c\}$.

3.4.3 n-ary intensional relations

We generalize the notion of unary constraints and binary relations into n-ary relations, where n is the number of intensional views over which a constraint is imposed.

Concept: N-ary intensional relation N	
Components: $N = (view_1, view_2, \ldots, view_n,$ $quant_1, quant_2, \ldots, quant_n,$ $pred, Incl, Excl)$	**Domain constraints:** $view_i \in \mathcal{V}$ $quant_i \in \mathcal{Q}uant_{A_i}$ $pred \in \mathcal{P}red_{A_1 \times \ldots \times A_n}$ $Incl, Excl \subseteq (\mathcal{T}_{A_1} \cup \{\_\}) \times \ldots \times (\mathcal{T}_{A_n} \cup \{\_\})$ where $A_i = view_{i_{Attr}}$ with $1 \leq i \leq n$
Well-formedness constraints: $Incl, Excl \subseteq$ $(ext(view_1) \cup \{\_\}) \times \ldots \times (ext(view_n) \cup \{\_\})$ $\forall\, (t_1, \ldots, t_n) \in Excl : p(t_1, \ldots, t_n)$ if $t_i \neq \_$ $\nexists\, (t_1, \ldots, t_n) \in Incl : p(t_1, \ldots, t_n)$ if $t_i \neq \_$	**Shortcuts for selector functions:** $N_{view_i} =$ $view_i$ with $1 \leq i \leq n$ $N_{quant_i} =$ $quant_i$ with $1 \leq i \leq n$ $N_{pred} =$ $pred$ $N_{Incl} =$ $Incl$ $N_{Excl} =$ $Excl$

Figure 3.11: Definition of n-ary intensional relation N

Definition of n-ary intensional relations

The definition of an n-ary intensional relation can be found in Figure 3.11 An n-ary relation consists of n intensional views, n quantifiers, an n-ary predicate $pred$ and $Incl$ and $Excl$ sets. Domain constraints and well-formedness constraint similar to those of binary intensional relations apply to n-ary relations. We denote the set of all N-ary intensional relations as $\mathcal{C}_n$

Semantics of n-ary intensional relations

Similar to binary intensional views, we define the semantics of n-ary intensional relations using a function $consistent(N)$ which verifies whether a n-ary relation is upheld and a function $discrepancies(N)$ which calculates the set of n-tuples of tuples for which the relation does not hold. Both these functions are a generalization of the functions for checking the consistency of binary intensional relations: instead of verifying whether the relation holds for a pair of tuples, as was the case with binary intensional relations, these functions verify the validity of the relation over n-tuples of tuples.
We define the function $consistent(N)$ as:

Definition 18. *For an n-ary relation $N \in \mathcal{C}_n$:*

Let $A_i = (N_{view_i})_{Attr}$ *with* $1 \leq i \leq n$

$$
\begin{aligned}
& consistent(N) \iff \\
& \qquad N_{quant_1}(extension(N_{view_1}), (N_{quant_2}(extension(N_{view_2})), \\
& \qquad \ldots, \\
& \qquad N_{quant_n}(extension(N_{view_n}), p))) \\
& \textit{where } p : \mathcal{T}_{A_1} \times \ldots \times \mathcal{T}_{A_n} \rightarrow Boolean : \\
& \qquad (t_1, \ldots, t_n) \rightarrow N_{pred}(t_1, \ldots, t_n) \vee \\
& \qquad\qquad\qquad match((t_1, \ldots, t_n), N_{Incl}) \wedge \neg(match((t_1, \ldots, t_n), N_{Excl})
\end{aligned}
$$

Similar to binary intensional relation, we define the set of discrepancies $discrepancies(N)$ as:

Definition 19. *For an n-ary relation* $N \in \mathcal{C}_n$*:*

$$
discrepancies(N) = \begin{cases}
\{(t_1, \ldots, t_n) \in \\
extension(N_{view_1}) \times \ldots \times extension(N_{view_n}) | \\
\quad (N_{pred}(t_1, \ldots, t_n) \vee match((t_1, \ldots, t_n), N_{Incl})) \\
\quad \wedge \neg match((t_1, \ldots, t_n), N_{Excl})\} & \textit{if } N_{quant_i} = \nexists \\
\\
\{(t_1, \ldots, t_n) \in \\
extension(N_{view_1}) \times \ldots \times extension(N_{view_n}) | \\
\quad (\neg(N_{pred}(t_1, \ldots, t_n)) \wedge \neg match((t_1, \ldots, t_n), N_{Incl})) \\
\quad \vee match((t_1, \ldots, t_n), N_{Excl})\} & \textit{otherwise}
\end{cases}
$$

Finally, the set of discrepancies of the i-th intensional view of N can be computed with:

Definition 20.

$$discrepancies_{V_i}(N) = \Pi_i(discrepancies(N))$$

3.5 Alternative views

3.5.1 Definition

The sections above described the core model of intensional views. In this section, we take a look at **alternative views**, an extension to our model which introduces a dedicated construct for expressing an often occurring type of constraint. The general idea behind alternative views is that, for a single intensional view, multiple intensions exist which describe the exact same set of tuples. For instance, we defined the *Accessors* view in our running example using an intension which retrieves all methods starting with the prefix "get-". Moreover, we stated that all of these methods must be implemented using an idiom: all accessor methods consist of a single statement which returns the value of an instance variable. Both conventions independently give a description of the concept of an accessor method. For the *Accessors* intensional view to be consistent, both these descriptions should yield the same set of tuples.

Concept: Intensional View V	
Components:	**Domain constraints:**
$V = (Parents, default, Alt)$	$Parents \subseteq \mathcal{V}$ $default \in \mathcal{A}lt$ $Alt \subseteq \mathcal{A}lt$
Well-formedness constraints:	**Shortcuts for selector functions:**
$V \notin Parents$ $\forall\, alt \in Alt : alt_{Attr} = default_{Attr}$	$V_{Parents} = Parents$ $V_{default} = default$ $V_{Alt} = Alt$

Figure 3.12: Definition of an intensional view V with alternative views

We slightly need to alter our definition of an intensional view to implement the notion of alternative views in our model. This altered definition of an intensional view V can be found in Figure 3.12 In this definition, the attributes $Attr$, the intension $query$ and the sets of deviations from this intension ($Incl$ and $Excl$) are no longer part of an intensional view. Instead, they are placed in a new kind of entity called an alternative view (the set of all alternative views is denoted by $\mathcal{A}lt$). The intensional view consists of the set of parent views $Parents$, a default alternative view $default$ and the other alternative views Alt of the intensional view. Similar to our original definition of an intensional view, the view V cannot be part of the parent views. What's more, we require that all alternative views Alt have the same attributes as the default alternative view $default$.

The definition of an alternative view Alt can be found in Figure 3.13. An alternative view Alt groups a set of attributes $Attr$, an intension $query$ and the sets $Incl$ and $Excl$ for specifying exceptions to the intension. Similar well-formedness constraints are valid for $Incl$ and $Excl$ as the ones we discussed in Section 3.3.3.

3.5.2 Semantics of alternative views

These changes in our model also have an impact on the semantics of intensional views. In Section 3.3.4 we described the semantics of an intensional view by means of a function $extension(V)$ which calculates the set of tuples belonging to that view. In this section, we redefine this function in order to take alternative views into account.

We first introduce a function $eval(alt, V)$ which takes as input an alternative view alt and an intensional view V. The function returns the set of tuples ($\mathcal{T}_{alt_{Attr}}$) that are calculated by the intension of the alternative view.

Concept: Alternative view Alt	
Components:	**Domain constraints:**
$Alt = (Attr, query, Incl, Excl)$	$Attr \in \mathcal{A}$ $query \in \mathcal{Q}_A$ $Incl, Excl \subseteq \mathcal{T}_A$
Well-formedness constraints:	**Shortcuts for selector functions:**
$Incl \cap eval(Alt, V)) = \emptyset$ $Excl \subseteq eval(Alt, V)$	$Alt_{Attr} = Attr$ $Alt_{query} = query$ $Alt_{Incl} = Incl$ $Alt_{Excl} = Excl$

Figure 3.13: Definition of an alternative view Alt

Definition 21.

$$eval(alt, V) = \begin{cases} \{t \mid t \in eval_{alt_{Attr}}(alt_{query}) \\ \quad \wedge \forall\, parent \in V_{Parents} : \\ \quad\; (\exists\, t' \in extension(parent) : \\ \quad\quad (\forall\, a \in (V_{Attr} \cap parent_{Attr}) : \\ \quad\quad\; t.a = t'.a))\} \end{cases}$$

This function is very similar to the $eval(V)$ function from Section 3.3.4. Using this function, we can define a function $extension(alt, V)$ which calculates the extension of an alternative view alt belonging to an intensional view V, taking the deviations from the intension of the alternative into account:

Definition 22.

$$extension(alt, V) = (eval(alt, V) \cup alt_{Incl}) \setminus alt_{Excl}$$

We revise the $extension(V)$ function in order to calculate the extension of an intensional view. This function takes as input an intensional view V with alternative views and returns the set of tuples belonging to this intensional view.

Definition 23.

$$extension(V) = \begin{cases} extension(V_{default}, V) & \text{if } consistent(V) \\ \textbf{\textit{undefined}} & \text{otherwise} \end{cases}$$

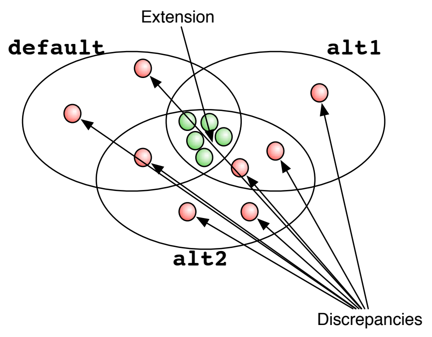


Figure 3.14: Inconsistencies of an intensional view V according to *extensional consistency*

We say that the extension of a view V is the extension of its default alternative, on the condition that the view is consist, i.e. if all alternative views of the view yield the same extension. If the intensional view is *not* consistent, the extension is undefined. Notice that this is different from our original definition of an intensional view – without alternative views – in which the extension of a view could always be calculated.

Finally, we define $consistent(V)$, a predicate which expresses when a view is considered to be *extensionally consistent*, meaning that all alternatives result in the same extension.

Definition 24.

$$consistent(V) \iff \forall\, alt \in V_{Alt} : extension(alt, V) = extension(V_{default}, V)$$

Similar to constraints over intensional views, it is not really interesting for a developer to only know whether an intensional view V is extensionally consistent or not. In order to resolve any inconsistencies, the developer must have access to the set of source code entities which violate the extensional consistency of the intensional view. In order to calculate this set of tuples, we define a function $discrepancies(V)$ which, given an alternative view alt and an intensional view V, returns the set of tuples which violate the extensional consistency of the view. Figure 3.14 illustrates this set of inconsistencies. Suppose we have a view V with three alternatives $default$, alt_1 and alt_2. If the intensional V is extensionally consistent, the extension of all alternatives is the same, thus:

$$extension(default, V) = extension(alt_1, V) = extension(alt_2)$$

In other words, all tuples belonging to the intensional view are part of the intersection of the three alternatives. However, if the intensional view V is not extensionally consistent, any tuples belonging to the extension of one of the alternative views, but not to the intersection of all alternatives, is considered to be a discrepancy.

We define the function $discrepancies(V)$ as:

Definition 25.

$$discrepancies(V) \; = \bigcup_{alt_i \in V_{Alt}} extension(alt_i) \setminus \bigcap_{alt_i \in V_{Alt}} extension(alt_i)$$

The discrepancies of an intensional view with alternatives consist of all tuples which are part of the union of the extensions of all alternative views, but which are not part of the intersection of the extensions. This resembles the situation as depicted in Figure 3.14. We are interested in finding all tuples which are part of *some* of the alternatives, but which do not belong to *all* of those alternatives.

3.5.3 Expressing alternative views using unary intensional constraints

As we mentioned earlier, alternative views are simply a layer of syntactic sugar we put on top of the model of intensional views. While they ease the expression of certain kinds of constraints, conceptually they do not add to the model of intensional views and constraints. To illustrate this, we show in this section how we can simulate the semantics of alternative views using regular intensional views and unary constraints.

Assume we have an intensional view V with n alternative views. For reasons of simplicity, we assume V does not have any parent views. We thus get:

$$\begin{aligned}
&V = (\emptyset, default, Alt)\\
&default = alt_1\\
&Alt = \{alt_2, alt_3, \ldots, alt_n\}\\
&alt_i = (Attr, query_i, Incl_i, Excl_i) \text{ for } 1 \leq \; i \; \leq n
\end{aligned}$$

Where alt_1 is the default alternative view and alt_i is the definition of the i-th alternative (for i between 2 and n). For all i, $query_i$ is the i-th intension; $Attr$ denotes the set of attributes that is shared by all alternative views.

We can achieve the same semantics as the intensional view above by constructing a regular intensional view V' and n unary intensional constraints U_i:

$$\begin{aligned}
&V' = (Attr, query, \emptyset, \emptyset, \emptyset)\\
&query : t \;\rightarrow\; query_1(t) \;\vee\; query_2(t) \;\vee\; \ldots \;\vee\; query_n(t)\\
&\forall\, i \;\in\; \{1..n\} : U_i = (V', \forall, pred_i, Incl_i, Excl_i)\\
&\forall\, i \;\in\; \{1..n\} : pred_i : t \;\rightarrow\; query_1(t) \;\wedge\; query_i(t)
\end{aligned}$$

The intensional view V' consists of an intension $query$, and has empty $Incl$ and $Excl$ sets. $query$ is defined as the logical disjunction of all intensions $query_i$ of the different alternative views of the intensional view V. Moreover, for each alternative there exists a unary constraint U_i which has as an intension the conjunction of the default intension (the intension of the first alternative) with the intension of the i-th alternative; the set of deviations $Incl_i$ and $Excl_i$ of U_i correspond to those of the alternative view Alt_i.

If we construct an intensional view V with n alternative views and a regular intensional view V' with n unary constraints as above, then both intensional views share the same semantics. More formally, the semantic functions we defined over regular intensional views and intensional views with alternatives correspond:

1. $extension(V) = extension(V') \Leftarrow consistent(V)$.
2. $consistent(V) \iff \forall\, i \in \{1..n\} : consistent(U_i)$
3. $discrepancies(V) = \bigcup_{i \in \{1..n\}} discrepancies(U_i)$

In other words:

1. If intensional view V is consistent then both intensional view V as well as V' yield the same extension;
2. The intensional view V with n alternative views is consistent if and only if the n unary constraints imposed over intensional view V' are consistent;
3. The discrepancies of the intensional view V are the same as the union of the discrepancies of the n unary constraints imposed on intensional view V'.

For a formal treatise of this subject and a proof of the above properties, see appendix A.

3.5.4 Unary intensional constraints versus alternative views

As we mentioned above, unary intensional constraints can be used to simulate behavior similar to the usage of alternative views. While both concepts are equivalent at the level of our formal model, from a user's point of view, they are employed in different situations.

Alternative views are used to define a constraint which is both a *necessary* and *sufficient* condition. In a sense, alternative views express different ways to compute the same set of tuples. For instance, in our example of the *Accessors* view, we require that all accessor methods start with the prefix "get-" and also that they must consist of a single statement which returns the value of a field. Due to the property of *extensional consistency* we require that both these alternative views for the *Accessors* intensional view yield the same set of tuples. Tuples which do not belong to the extension of all alternative views are considered possible inconsistencies. We thus require that **all** methods which start with the prefix "get-" are also implemented by a single statement returning the value of a field, and vice versa.

Unary intensional constraints define a *necessary* condition which must be upheld by all the tuples of an intensional view. For example, earlier on we introduced the unary constraint which expresses that all accessor methods must contain the name of the field they are accessing in their name. The main reason why we opted to implement this constraint using a unary intensional relation and not as an alternative view of the *Accessors* view is that this constraint is not *sufficient*. While it must be true for all accessor methods that their name contains the field they are accessing, it is not required that all methods that contain the name of a field are also accessor methods.

3.6 Discussion

Intensional versus extensional In Section 3.3.2 we already discussed that we opted for an intensional description of the set of tuples making up an intensional view rather than an extensional enumeration of all the tuples. Since our goal is to provide a model for documenting, and especially evolving regularities in source code, the choice for an intensional description of a view seems evident. Due to the changing nature of the source code of a system, the set of tuples belonging to the extension of an intensional view also changes over time. The use of an intensional specification in order to define intensional views can thus be considered an enabling factor in automatically supporting the evolution of the extension of intensional views.

Each time new software entities are added to the system or existing ones are modified or removed, this can have an impact on the extensions of the intensional views which are defined on the system. Using an extensional description of an intensional view, all such changes in the source code of the system would have to be explicitly documented by manually adapting the set of tuples belonging to the extension a view. However, the intensional description of a view renders it possible – assuming that the intension of the view classifies the correct set of tuples – to automatically update the set of tuples in order to reflect the changes in the system. In other words, as long as the intension of an intensional view classifies the correct set of tuples, changes to the source code of the system will be automatically captured in the extension of the intensional view.

Combination of intensional and extensional specification Our classification mechanism is not purely intensional. Although we do not allow a developer to specify an intensional view by enumerating the tuples belonging to its extension, we do offer support for extensionally fine-tuning that extension. More specifically, we provide a hybrid classification model in that we allow for the extensional documentation of explicit deviations to the intension of an intensional view. When specifying the set of tuples belonging to the extension of an intensional view, the general case, which entails the majority of the tuples belonging to the extension is captured by an intensional description. This intensional description is complemented with an extensional mechanism to document deviations to the intension: a developer can enumerate tuples which should be explicitly included in the extension of the view or excluded from that extension.

Overlap between different intensional views An intensional view consists of a set of tuples which associate a number of attributes to a software entity (i.e. an element of the universe $\mathcal{U}$ of software entities). This does not keep one single software entity from appearing in multiple intensional views. Due to the nature of software, this is even a desirable feature of our model. In practice, one software entity (class, method, . . .) is often involved in the implementation of multiple concerns or concepts in the system. Since such a concept in the source code is aligned with an intensional view, the same software entity thus can be part of the tuples of multiple views.

Crosscuttingness of intensional views Due to the "tyranny of the dominant decomposition" [TOHS99b], certain concerns in the implementation of a piece of software are misaligned with the modularization of the software. Typical examples of such concerns are *logging*, *tracing*, *transaction management*, etc. Such so-called crosscutting concerns are scattered throughout the code: their implementation is not localized in a single module but can rather spread out over multiple modules. Moreover, their implementation is not cleanly separated from the base concerns which they interact with. Instead, the code implementing crosscutting concerns is tangled with the base functionality of the software. As a result, crosscutting concerns place an additional burden on software developers in terms of evolvability and maintainability of a piece of software.

While new modularization techniques such as aspect orientation [KLM$^+$97] try to capture such crosscutting concerns in a separate module, our model of intensional views complements such techniques by documenting and verifying the software regularities which underly these crosscutting concerns. Our model is inherently suitable for dealing with concerns which crosscut the hierarchical decomposition of a piece of software. Since the queries used to define an intensional view are naturally crosscutting, and a single source-code entity can appear in the extension of multiple views, the views can serve as a means of documentation of the crosscutting concerns and can be used by developers during maintenance tasks to identify the source-code entities related to a particular crosscutting concern. In chapter 7 we take a more in-depth look at the relation between intensional views, crosscutting concerns and aspect-oriented programming and discuss how our approach can aid in alleviating the fragile pointcut problem.

Interpretation of discrepancies In the sections describing the model of intensional views and constraints over these intensional views, we have focused on providing a number of functions for calculating the discrepancies between the documented constraints and the implementation (the source-code entities). While these functions give an accurate description of the tuples representing source-code entities for which the constraints and the implementation do not conform, we only approached these discrepancies from a mathematical point of view and have not yet attributed an interpretation to them, since this interpretation depends on the expert knowledge of a developer over the software system.

In the next chapter (Chapter 4) we give a number of practical examples of how discrepancies can be interpreted, however, we wish to stress that, since our goal is to *co-design* and *co-evolve* the documentation and implementation, such discrepancies do not always signify that the implementation is erroneous with respect to the documentation. In practice, the structural regularities in a software system also evolve, and inconsistencies between the documentation and the implementation can also be an indicator that the documentation is outdated and should be adapted.

Expressiveness of the model Note that the expressiveness of a particular instantiation of our model of intensional views and constraints depends on the underlying query language and source-code meta model that are used in this instantiation. For example, if the meta model or query language do not provide support for reasoning at the granularity of method bodies, then the expressiveness of our approach is hampered in that e.g. regularities expressing particular implementation patterns in the source code cannot be expressed and verified. Similarly, if we use a query language that only reasons over parse-trees, we will be able to express a large variety of structural regularities but will be limited in expressing regularities about the behaviour of the software. As such it is not possible to make any general claims about the expressiveness of our formal model.

In the next chapter, we present one particular such instantiation of our model, namely the IntensiVE tool suite. Throughout the remainder of this dissertation we will demonstrate that this particular instantiation is sufficiently expressive to document and verify a wide range of structural source-code regularities.

Completeness While devising our model of intensional views, one of our goals was to offer a complete model with respect to the kinds of constraints the model allows to impose on the source code of a system. As such, we did not limit ourselves to unary intensional constraints and binary intensional views, but rather provided a conceptual framework (by means of n-ary relations) for imposing a constraint on any given number of intensional views.

Since we did not specify the actual query language in which the intension of a view is specified and the language used to implement the predicates used in our constraint mechanism, we cannot make any claims regarding completeness of our model with respect to the kinds of entities which can be classified nor to the relationships which can be expressed between these kinds of entities.

Minimality When we first introduced the concepts underlying our model of intensional views in Section 3.1 we discussed that our model consists of intensional views, i.e. intensionally defined classifications, and constraints over these classifications. These two concepts are at the core of our approach and provide a minimal model for the entities which make up our model of intensional views.

We devised a number of variations and extensions to these two core concepts in order to provide additional ease of use for the developer and to support the documentation of structural source code regularities. For instance, we included the concept of parent views as a means to provide a scope for intensional views. Similarly, we incorporated support for explicitly documenting exceptions to the intension of an intensional view. On the level of constraints, we introduced the notion of unary intensional constraints and binary intensional relations, which are instantiations of the concept of n-ary intensional relations. Moreover, we also presented the notion of alternative views as a specific kind of constraint and showed that this type of constraint is equivalent to the use of unary intensional constraints.

3.7 Conclusions

In this chapter we have introduced the model of intensional views and constraints, which lies at the core of our approach for documenting, verifying and supporting co-design and co-evolution of structural source-code regularities throughout the implementation of a software system. We have approached our model from a conceptual point-of-view and discussed the concepts of intensional views and constraints on these intensional views using a specification of the concepts in a formalism inspired by relation tuple calculus. This formal specification allowed us to introduce the model independent of any implementation details and allowed us to precisely specify the semantics of the different concepts in the model.

In the next chapter, we take a look at IntensiVE, our prototype implementation of the model of intensional views and constraints. While we left some concepts abstract in this chapter, such as for instance the query language employed to specify the intension of an intensional view, in the next chapter, we give a couple of examples of such concrete query languages.

Chapter 4

IntensiVE

In this chapter we present the technical contribution of this dissertation, namely the Intensional Views Environment, or IntensiVE for short. IntensiVE is our prototype tool suite implementing the model of intensional views. We have developed IntensiVE as an extension to the Cincom VisualWorks Smalltalk environment [Cin07]. IntensiVE consists of a number of tools to support the documentation, verification and evolution of structural source-code regularities in the source code of object-oriented software systems.

We set forward four goals for this chapter:

1. **Section 4.2** When discussing our formal model of intensional views and constraints in Chapter 3 we deliberately abstracted from the underlying software model on which the intensional views are imposed, as well as from the languages used to specify the intension or the predicate of respectively an intensional view or an intensional constraint. Instead we declared a number of properties which such an intension or a predicate must adhere to. The first goal of this chapter is to demonstrate concrete instantiations of these abstract concepts from the formal model. We present how both the intension of an intensional view can be specified as well as the predicate of an intensional constraint, using the Smalltalk language and the SOUL logic language. While the former language allows us to use Smalltalk itself as an underlying model over which intensional views are defined, the latter provides support to easily reason about Smalltalk as well as Java programs in terms of an underlying software model;

2. **Sections 4.3 to 4.5** The second goal of this chapter is to sketch an overview of the IntensiVE tool suite. It is this prototype tool suite which we use throughout the next chapters as a means to perform the experimental validation of our work;

3. **Section 4.6** Furthermore, we discuss a number of methodological issues which arise from our approach and tool suite;

4. **Section 4.7** In Chapter 2, we provided an overview of some of the related work from the field of software classifications as well as approaches for supporting structural source-code regularities throughout the development process. We finish this chapter by comparing this related work to the approach we propose in this dissertation.

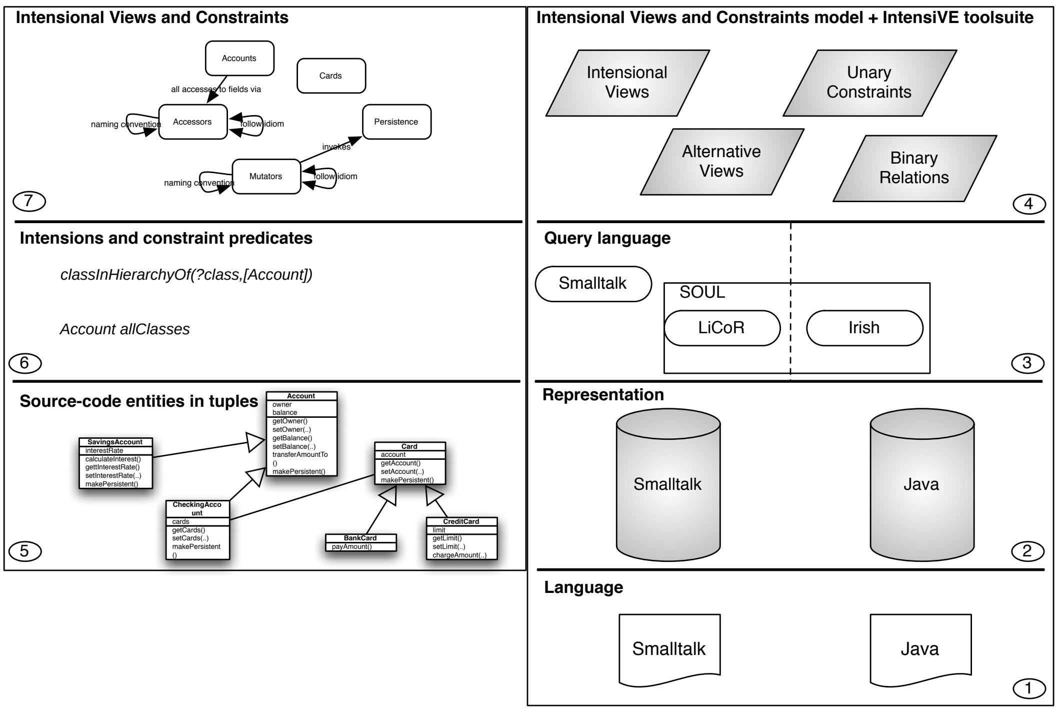


Figure 4.1: Overview of IntensiVE

4.1 Overview of IntensiVE

Environment

In line with the tool-oriented philosophy of Smalltalk, we opted for developing our implementation of the model of intensional views as a tool suite which integrates seamlessly with the surrounding Smalltalk integrated development environment. IntensiVE is implemented as a plugin for the StarBrowser [WD04], a framework for VisualWorks Smalltalk which provides capabilities for implementing software engineering tools based on a classification mechanism. The tight integration of both our tool suite as well as the StarBrowser framework with the Smalltalk environment enhances the capabilities of our tool suite. IntensiVE is not limited to documenting structural source-code regularities and verifying their validity with respect to the source code. A user of our tool suite can directly browse the source-code entities that belong to an intensional view or that violate one of the expressed structural source-code regularities, thus aiding the integration of intensional views into the development process. Another benefit of this tight integration is that the intensional views created by our tool suite are first-class entities within the Smalltalk environment. As such, it is possible to make them directly accessible to other development tools like for instance the unit testing framework of Smalltalk.

Figure 4.1 presents an overview of IntensiVE. The right-hand side of the figure shows the different layers in the implementation of our tool suite. On the left-hand side, the figure illustrates which entities are present in each corresponding layer. The top-most layer contains our actual implementation of the model of intensional views and constraints. This layer makes use of the query languages (second layer) in order for defining the intension of a view or the predicate of a constraint. Such intensions and constraints are expressed in terms of a representation (third layer) of a program written in either the Smalltalk or Java language. In the following sections we discuss each of these layers of IntensiVE in more detail.

Supported languages and underlying software meta-model

In Chapter 3 we did not focus on the underlying software model in terms of which the tuples belonging to the extension of an intensional view are expressed. We rather denoted this software model as the universe $\mathcal{U}$ of source-code entities which, in a tuple, are associated with a certain attribute. Our implementation supports a software meta-model for two programming languages: namely Smalltalk and Java[1] (number 1 in Figure 4.1). For both of these languages, our tool suite uses a software model that contains a representation of a program in either language (number 2). The entities from this representation of a program are the universe $\mathcal{U}$ of source-code artifacts (number 5 in Figure 4.1) that are associated with the attributes of the tuples.

Since our tool suite is implemented in Smalltalk, we make use of the reflective capabilities of this language. We reuse the first-class, fully reified representation of Smalltalk programs which is available as a data structure in the Smalltalk environment as the software meta-model for Smalltalk. This allows us to associate the actual object representing a source-code entity with an attribute of a tuple. As for Java, we use the representation of a Java program obtained by applying the Frost parser. This Java parser, implemented in Smalltalk, represents a Java program by an object-oriented version of its abstract syntax tree. The objects in this syntax tree serve as the source-code entities of the tuples belonging to the extension of an intensional view.

Query languages

In order to query the software model and retrieve the source-code entities which belong to the tuples of the extension of an intensional view, our tool suite also needs access to a query language which can be used to express the intension of a view. In Chapter 3 we defined such a query language as a function $eval$ which takes as input the intension of an intensional view and returns the set of tuples belonging to the extension of the view. IntensiVE supports two instantiations of such query languages (box number 3 of Figure 4.1), which allow a user to define an intension (two examples are shown in box number 6) in terms of the software models discussed above.

- **Smalltalk:** our tool suite supports Smalltalk as a language for expressing intensions in terms of the Smalltalk software model. As we mentioned earlier, Smalltalk programs are represented as first-class entities in the Smalltalk language itself. This first-class

[1]Java 1.2

representation, combined with the *reflective facilities* the language offers, enables reasoning about Smalltalk programs from within that same language;

- **SOUL**: The Smalltalk Open Unification Language (SOUL) [Wuy01] is an implementation of a Prolog-like language on top of Smalltalk. It provides a declarative paradigm for reasoning about object-oriented software and offers symbiotic capabilities with respect to the underlying Smalltalk language. The SOUL language can be used to reason about the Smalltalk as well as the Java software model. To this end it provides two libraries of logic predicates:
  - **LiCoR:** the Library for Code Reasoning (LiCoR) [MMW01] provides reasoning capabilities for the Smalltalk language. It consists of a large number of predicates which ease expressing queries over the fully-reified representation of a Smalltalk program;
  - **Irish:** Irish [FM04] is – like LiCoR – a collection of predicates for the SOUL language. The predicates in this library are tailored for reasoning over the Java language. In particular, these predicates use the representation of Java obtained by using the Frost parser.

In Section 4.2 we discuss these query languages in more detail and demonstrate how they can be employed to express the intension of an intensional view.

Support for the model of intensional views and constraints

Our implementation of intensional views is for the larger part a literal translation of the concepts and semantics of the formal model we presented in Chapter 3. As such, our tool suite provides support for expressing intensional views over both the Smalltalk and the Java software model, using Smalltalk and SOUL as query languages. Moreover, IntensiVE supports, alongside with intensional constraints, the notion of alternative views as a means to impose constraints over intensional views (box number 4 of Figure 4.1). This support for the different concepts of our model can then be used to express a number of intensional views and constraints (box number 7) over the entities in the software model, using any of the supported query languages.

However, our implementation poses a number of restrictions on the general model of intensional views: A first restriction lies in the fact that, while in our formal model an intensional view V can have any number of parent views, our implementation only allows the specification of at most one parent view. This lack of multiple parent views does not impact the expressiveness of an intensional view, since these multiple parents can be simulated by repeating parts of the intensions of the parent views in the intension of the child view. Furthermore the tight integration of intensional views with the surrounding Smalltalk environment makes it possible to access intensional views directly from within a query language (as we will explain in Section 4.2). As a result, intensional views which rely on multiple other views in order to restrict their scope, can directly access these views from within their intension, thus minimizing the code duplication occurring in multiple intensional views.

A second restriction of our implementation concerns the kinds of constraints our tool suite supports. While we introduced the notion of n-ary relations in our formal model, the kinds of constraints expressible in our tool suite is restricted to unary intensional constraints and binary intensional relations. As a result, our tool suite is no longer complete with respect to the constraints which can be imposed over intensional views. Although this renders our tool suite less expressible than the formal model of intensional views, none of the examples of structural source-code regularities we will encounter in this dissertation require the use of ternary or n-ary intensional relations.

Overview of the tool suite

Our tool suite consists of five different sub-tools:

- **Intensional View Editor:** this tool is used to define and manipulate intensional views;
- **Extensional Consistency Inspector:** for a given intensional view, this tool provides detailed feedback on whether the different alternatives of the intensional view are extensionally consistent, i.e. if they all yield the same set of tuples;
- **Intensional Relation Editor:** this tool is used to define and manipulate unary and binary intensional relations;
- **Relation Consistency Inspector:** this tool can be used to report on discrepancies between a certain intensional constraint and the tuples belonging to the extension of the intensional view(s) which the constraint is imposed on;
- **Visualization Tool:** this tool provides a visual representation of the different intensional views and constraints in the system and highlights whether the different views and constraints are consistent or not.

Later on in this chapter, we discuss each of these tools in more detail.

4.2 Supported query languages

A key component of an intensional view V is the intension $query$ which selects the set of tuples that belong to the extension of the intensional view V. While in our formal model, we did not specify any concrete query language in which the intension of a view can be described, we did impose one requirement, namely that for any such query language there must exist an evaluator. The result of evaluating an intension $query$ must be a set of tuples with attributes $Attr$ such that in each tuple, every attribute is associated with a source-code entity from the underlying software meta-model.

In this section, we take a look at two such languages which are supported by the IntensiVE tool suite. These two languages demonstrate how two vastly different paradigms can be used for expressing the intension of an intensional view. In Section 4.2.1, we take a look at the object-oriented programming language Smalltalk, and exemplify how this language, and in particular its reflective capabilities, can be utilized to express the intension of a view.

The second language – SOUL – which we discuss in Section 4.2.2 offers an instance of the declarative paradigm in order to reason about object-oriented programs. This language, and especially the LiCoR and Irish predicate libraries, enables us to define intensions using a declarative paradigm.

Recall from our formal model that similar language facilities are needed in order to express the predicate of an intensional constraint. We specified that such a predicate must take as input a number of tuples (depending on the arity of the intensional constraint) and return a boolean. To implement such predicates, we also use – similar to defining the intension of a view – the Smalltalk and SOUL languages. In Section 4.4.1 we will come back to this topic and discuss how we integrated SOUL and Smalltalk into our framework such that both languages can be used to express the predicate of a constraint.

4.2.1 Smalltalk

The first language for specifying the intension of an intensional view we discuss is Smalltalk. While Smalltalk is the language in which we implemented our tool suite, it also possesses a number of characteristics which render it a versatile tool for expressing the intension of an intensional view. First, Smalltalk programs are represented as data in the Smalltalk language itself. As such, a fully-reified representation of a Smalltalk program is accessible from within the Smalltalk language. Second, the reflective capabilities of Smalltalk ease the retrieval of information from this first-class representation of a Smalltalk program. Using the meta-object protocol of Smalltalk, a user can query a Smalltalk program and extract the required information. We do not discuss the Smalltalk meta-object protocol here, but rather refer to [GR89] for an in-depth treatise of the subject. In what follows, we demonstrate the use of Smalltalk as a query language by showing a number of examples of intensions from the actual Smalltalk implementation of the Banking System (our running example from Chapter 3).

Mechanics of using Smalltalk as a query language

While the reflective capabilities of Smalltalk can be used to express the intension of an intensional view, this does not guarantee that the result of such a Smalltalk meta-program is a valid extension, i.e. a set of tuples that associate a source-code entity with each of the attributes of an intensional view. In order to construct such extensions and tuples, we provide a small framework which is used to construct an intension using Smalltalk. This framework consists of the following classes:

- **Extension**: instances of the `Extension` class contain the actual tuples which belong to the extension of an intensional view. Since `Extension` shares a lot of similarities with a regular collection, we provided an interface for this class which is similar to that of the `Collection` class, which is part of the standard Smalltalk collection framework. In addition, `Extension` contains methods which allow a user to add, remove and access tuples, and to iterate over the tuples in an extension;

- **Tuple**: the tuples which belong to the `Extension` are represented by instances of the `Tuple` class. This `Tuple` class implements facilities for associating an attribute

```
1  extension := Extension new.
2  Banking allClasses do:[:class |
3     class selectorsAndMethodsDo:[:selector :method |
4        extension add:
5          (Tuple new attribute:#class  value: class;
6                     attribute:#method value:
7                       (SmalltalkMethod compiledMethod:method))]].
```

Figure 4.2: Example of an intension for the *Banking* intensional view specified in Smalltalk

with a source-code entity and managing these associations. The most important method `Tuple` implements is `attribute:value:`. This method associates a certain value with a given attribute. Furthermore, we provide a method `valueFor:` which retrieves the association value of a particular attribute.

Using this framework, a user can create a valid intension by writing a Smalltalk meta-program which returns an instance of `Extension`.

Examples

To illustrate such an intension expressed in Smalltalk consider the example in Figure 4.2. This Smalltalk program represents the intension of an intensional view *Banking*. The rationale behind the *Banking* intensional view is that it groups all the classes and methods in the implementation of the banking system, thus serving as a parent view to limit the scope of the other intensional views declared over this example system. As such, the *Banking* view has two attributes, namely 'class' and 'method'. The extension of this intensional view should contain a number of tuples, each consisting of the methods of the banking system along with the class which they are implemented in.

The Smalltalk program calculating this extension consists of two loops. The outermost loop iterates over all the classes in the `Banking` namespace (line 2), which contains all the classes belonging to the banking system. The innermost loop iterates over all the selectors and methods of each of these classes (line 3) and adds for each of these methods (line 4) a tuple to the extension collection. Each tuple contains two associations: one association (line 5) which associates the attribute 'class' with `class` and an association (lines 7 and 8) which binds the attribute 'method' to an instance of `SmalltalkMethod`, which is a wrapper around the actual object representing the method.

Another example of an intension specified using Smalltalk is shown in Figure 4.3. This intension for the *Mutators* intensional view is based on the rationale that in Smalltalk mutator methods are mostly implemented following a similar pattern. For example, for a certain instance variable `field`, in Smalltalk the mutator method would be named `field:` and contain an assignment to the instance variable. Based on this idiom, we express the intension of the *Mutators* view using Smalltalk as the query language.

This example contains a more complex illustration of an intension described in Smalltalk. Similar to the intension for the *Banking* system, we start by creating a `Extension` object (line 1), and iterate over all the classes in the `Banking` namespace (line 2) and the selectors

```
1  extension := Extension new.
2  Banking allClasses do:[:class |
3   class selectorsAndMethodsDo:[:selector :method |
4     class instVarNames do:[:field |
5        ((method writesField:(class instVarIndexFor: field))
6          & (field asString,':' = selector asString))
7              ifTrue:[extension add:
8                 (Tuple new
9                    attribute:#class value: class;
10                   attribute:#method value:
11                       (SmalltalkMethod compiledMethod: method);
12                   attribute:#field value:field asSymbol)]]]].
```

Figure 4.3: Example intension in Smalltalk for the *Mutators* intensional view

and methods implemented by those classes (line 3). Lines 4 to 6 iterate over all the field names of the class and verify if the method performs a write operation to the field at the index corresponding with the field. What's more, it is also verified that the name of the field matches the naming convention we illustrated above by comparing the method name with a concatenation of the field name and a colon (':'). If such a write occurs and if the name of the method follows the naming scheme, a tuple is added to the extension which groups the class, method and field (lines 7–12).

Intensional views as first-class entities in Smalltalk

Due to the open implementation of the Smalltalk language and the fact that the intensional views defined over a system are first-class entities within Smalltalk, it is possible to extend the Smalltalk query language capabilities with a mechanism which makes it possible to transparently access an intensional view from within the intension of another view. This symbiosis with the Smalltalk language makes it possible to define intensional views in terms of other views. Furthermore, we also implement a number of constructs which allow the extraction of the values of specific attributes from an intensional view (similar to a `select` statement from SQL), together with a number of set-theoretical operations such as intersection, union and difference. Using these constructs, we offer the developer a basic language for combining and composing intensional views from within Smalltalk.

This makes it possible to express composite intensional views using the Smalltalk language, or to provide any other means of combination of a number of intensional views. In order to obtain this functionality, we extended our tool suite and its integration with the Smalltalk language as follows:

- We implemented a lookup mechanism such that intensional views can be accessed directly from within Smalltalk code. This is done by reusing the semantics associated with the VisualWorks Smalltalk namespace mechanism. A namespace in VisualWorks is a kind of modularization mechanism which groups a number of classes. For instance, we already used this mechanism when we specified the Smalltalk intension of

```
1 (((Views.Mutators for:#class and:#field)
2    difference:
3      (Views.Accessors for:#class and: #field)) for:#class)
4        intersection:(Views.Accounts for:#class)
```

Figure 4.4: Example of an intension in Smalltalk using intensional views as first-class entities

the *Banking* and *Mutators* intensional views. Using this namespace mechanism, a user can directly quantify a certain class in a certain namespace by using a dot operator. E.g. to refer to the class `Card` in the `Banking` namespace the expression `Banking.Card` can be used.

We overloaded this namespace mechanism by creating a "virtual" namespace `Views`. For this virtual namespace, the lookup mechanism provided by the dot operator will return the intensional view corresponding with the identifier used on the right hand side of the dot. For example, `Views.Accessors` will return the *Accessors* intensional view as a first-class entity;

- The implementation of intensional views provides a number of methods which enable basic tuple set operations on intensional views such as for instance projection (selecting a subset of the attributes of the tuples in the extension of an intensional view), union, intersection and difference.

An example of an intension which uses this mechanism is demonstrated in Figure 4.4. The result of this extension is the set of all tuples with a single attribute 'class' bound to a class in the `Account` hierarchy which contains a field for which there exists a mutator method but not an accessor method.

If we take a more in-depth look at this intension, we see that it uses the virtual namespace mechanism in order to retrieve the intensional views *Mutators*, *Accessors* and *Accounts*. Line 1 retrieves the *Mutators* intensional view and projects the tuples belonging to the extension of that view to the attributes 'class' and 'field'. The result is thus a set of tuples with as attributes 'class' and 'field'. Similarly, in line 3 the same information is extracted from the tuples belonging to the *Accessors* intensional view. In line 2, the difference between these two sets is calculated, resulting in a set of tuples representing all fields and the class in which they are defined for which there exists a mutator method but not an accessor method. At the end of line 3, the associations of the 'class' attribute are extracted from this set. Finally, in line 4, the intersection of these classes is calculated with respect to the associations for 'class' in the *Accounts* intensional view .

4.2.2 Smalltalk Open Unification Language (SOUL)

The second query language IntensiVE supports is SOUL. In Chapter 2 we already briefly discussed this language. SOUL is an implementation of a Prolog-like declarative language on top of Smalltalk. This declarative paradigm, as has been demonstrated in [Wuy01, MKPW06, WM06, MMW01], combined with SOUL's symbiotic capabilities, makes it an ideal candidate

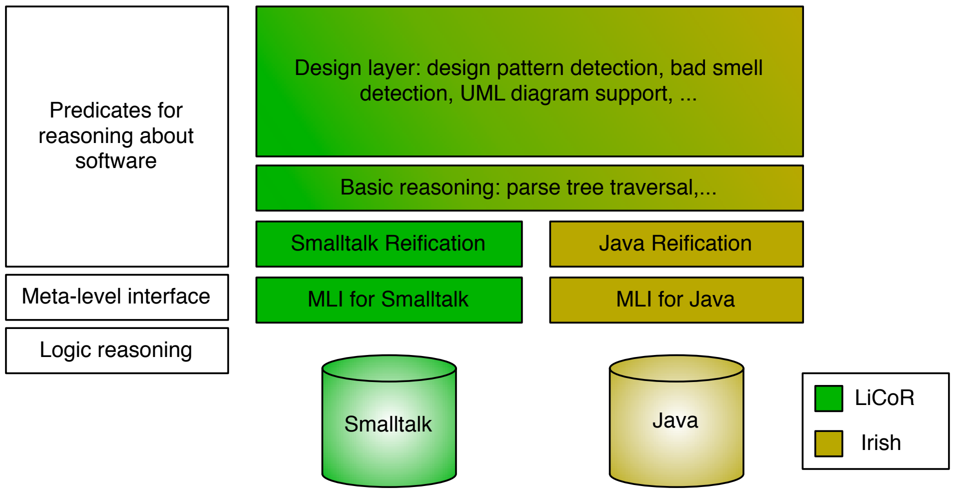

Predicates for reasoning about software
Meta-level interface
Logic reasoning
Design layer: design pattern detection, bad smell detection, UML diagram support, ...
Basic reasoning: parse tree traversal,...
Smalltalk Reification
Java Reification
MLI for Smalltalk
MLI for Java
Smalltalk
Java
LiCoR
Irish

Figure 4.5: Architecture of SOUL: LiCoR and Irish

for reasoning about source code. One of the key features of SOUL is that it supports symbiosis with the underlying Smalltalk environment. This renders it possible to reason directly about Smalltalk objects from within SOUL. Moreover, this symbiosis allows a SOUL query to invoke Smalltalk code during evaluation.

Although SOUL heavily relies on this symbiosis with Smalltalk, it was developed with the intent to be independent of the language about which it reasons. To obtain this independence, the SOUL language only offers a logic kernel together with a number of basic logic primitives. In order to reason about object-oriented programs, a separate library of logic predicates is necessary which uses SOUL and the symbiosis with Smalltalk. This architecture of SOUL is depicted in Figure 4.5. Two such libraries for predicates exist, namely the Library for Code Reasoning (LiCoR) by Mens, Wuyts et al. [MMW01] which reasons over Smalltalk code and is depicted in green in Figure 4.5, and Irish [FM04] (in yellow), developed by Johan Fabry, which provides similar functionality for Java programs.

Each of these libraries provides a meta-level interface (MLI). This MLI is used by the predicates in the library in order to communicate with the software model which is being queried. As such, the MLI implements facilities for retrieving source-code entities from the software model and verifying basic dependencies between such source-code entities. The MLI thus serves as the interface between the logic language and the software model. LiCoR's MLI makes use of the Smalltalk meta-object protocol in order to directly access the entities in a Smalltalk program; the MLI of Irish reasons over the abstract syntax tree of a Java program as obtained by applying the Frost Java parser.

On top of this MLI, each library provides a set of *language-dependent* predicates which present a logic reification of the software entities. These predicates, such as `class`, `method`, `methodInClass` and so on, use the symbiotic capabilities of Smalltalk in order to communicate with the MLI and retrieve the actual objects representing the Smalltalk program or Java AST nodes. On top of these reification layers, a number of layers of predicates are im-

```
classInNamespace(?class,[Banking]),
methodInClass(?method,?class)
```

Figure 4.6: Example of a SOUL intension for the *Banking* intensional view

plemented which are designed to be as *independent* as possible from the the underlying model of software entities about which one is reasoning with SOUL (indicated in Figure 4.5 by the gradient between green and yellow). The layered architecture of the SOUL predicate libraries prescribes that a predicate can only use predicates in the same or an underlying layer. Due to this layering mechanism, predicates in the upper layers of the predicate libraries do not depend on specific Smalltalk or Java constructs and implement functionality which can be reused when reasoning over either language. As such, only predicates in the reification layer are allowed to communicate with the MLI.

Mechanics of using SOUL as a query language

In IntensiVE, it is possible to use SOUL as the query language for expressing an intension. Depending on whether one is declaring intensional views on a Java or a Smalltalk system, respectively the Irish or LiCoR library can be used. When evaluating a SOUL query, all solutions are returned which satisfy the query. Each such solution consists of a set of bindings for the variables used in the logic program such that the values for these variables are a solution for the query.

It is straightforward to translate these semantics of a logic program into those of the intension of an intensional view. An intension expressed using SOUL corresponds to a logic SOUL query. Upon evaluation, the solutions of the query are considered to be the extension of the intensional view. To this end, each solution of the query aligns with a tuple in the extension of the view. The bindings of the attributes are obtained by extracting the corresponding variables from the result of the SOUL query. A SOUL query is thus considered to be a valid intension for an intensional view if all attributes of the intensional view are used as a variable in the SOUL query.

Examples

A first example of an intension expressed in the SOUL language can be found in Figure 4.6. This intension is the SOUL variant of the intension for the *Banking* intensional view, which we presented in Figure 4.2. The SOUL program in the figure consists of only two conditions, thus being more succinct than the Smalltalk version[2]. The first condition restricts the bindings of the variable ‘class’ (variables in SOUL are indicated by a question mark) to the classes belonging to the `Banking` namespace. The second condition will result in bindings for the variable ‘method’ such that ‘method’ is bound to a method implemented in class ‘class’. By evaluating this intension as a query, a set of tuples will be calculated such that for each method in the banking system there exists a tuple consisting of an attribute ‘method’

[2]This succinctness is mostly the result of the different abstractions LiCoR offers to reason about source code.

```
1 statementsOfMethod(statements(?statements),?method),
2 instanceVariableInClass(?field,?class),
3 member(?statement, ?statements),
4 equals(?statement, assign(variable(?field),?value))
```

Figure 4.7: Example of a SOUL intension for the *Mutators* intensional view

```
1 isFieldInClass(?field,?class),
2 nameOfField(?fieldname,?field),
3 methodInClass(?method,?class),
4 methodWithIdentifier(?method,?name),
5 [('get', ?fieldname asString) match: ?name asString]
```

Figure 4.8: Example of a SOUL intension for the *Accessors* intensional view

which is bound to the method and an attribute 'class' which is bound to the class on which the method is implemented.

The SOUL variant of the intension of the *Mutators* intensional view can be found in Figure 4.7. Similarly to the Smalltalk version (Figure 4.3), the intent behind this intension is to capture all the methods which write to a field. The SOUL version of this intension consists of 4 lines. Line 1 selects all the statements 'statements' implemented by a method 'method'. Notice that, since this intensional view has the *Banking* intensional view as its parent, the 'method' variable will only be bound to methods from the banking system. In line 2 all fields 'field' are selected which belong to a class 'class' in the banking system. Finally, lines 3 and 4 impose the restriction that at least one of the statements in the parse tree of the 'method' performs an assignment to a field.

We mentioned above that the symbiosis between Smalltalk and SOUL makes it possible to execute Smalltalk code during the evaluation of a SOUL query. An example of an intension which uses this mechanism is shown in Figure 4.8. This intension captures the entities belonging to the *Accessors* intensional view. We defined this view in Chapter 3 informally as all methods 'method' implemented by a class 'class' which start with the prefix "get-" and which access a field 'field'.

Upon evaluation, the SOUL program in Figure 4.8 will retrieve all fields in the system, along with their class (line 1). From this field, the name of the field is extracted (line 2) and bound to the variable 'fieldname'. Lines 3–4 retrieve all the methods in the class along with the identifier of that method. Finally, line 5 shows the use of a Smalltalk block: a logic condition in the SOUL program which is resolved by executing the Smalltalk code between the square brackets. In our example, this fragment of Smalltalk code performs some pattern matching: it expresses that the identifier of the method should match the concatenation of the string "get-" together with the name of a field in the class in which the method is implemented. Notice that in these Smalltalk blocks the value of a logic variable can be used. Since the *Accessors* intensional view has three attributes, namely 'class', 'method' and 'field', the values for these variables will be extracted from the result of the SOUL query and will make up the tuples belonging to the extension of the intensional view.

```
Mutators(?class,?,?field),
not(Accessors(?class,?,?field)),
Accounts(?class,?)
```

Figure 4.9: Example of using intensional views as first-class entities in SOUL

```
Accessors(?class,?method,[#age])

Accessors([Card],?method,[#account])
```

Figure 4.10: Example of using unification in combination with intensional views as first-class entities

Accessing intensional views as first-class entities in SOUL

Similar to using Smalltalk as a query language for specifying the intension of an intensional view, it is possible to use other intensional views from within the intension of a view when using SOUL as a query language. In order to achieve this first-class citizenship of intensional views in the logic language, we unified the concept of an intensional view with that of a logic predicate. For instance, if we have an intensional view V that has as attributes $a_1, \ldots, a_n$, this intensional view is reified within SOUL by a predicate V with arity n. If we consider for example the intensional view *Accessors* with as attributes 'class', 'method' and 'field', the corresponding logic predicate is:

$$Accessors(?class, ?method, ?field)$$

Upon evaluation of this predicate from within a SOUL program, each solution of this condition corresponds to a tuple belonging to the extension of the *Accessors* view.

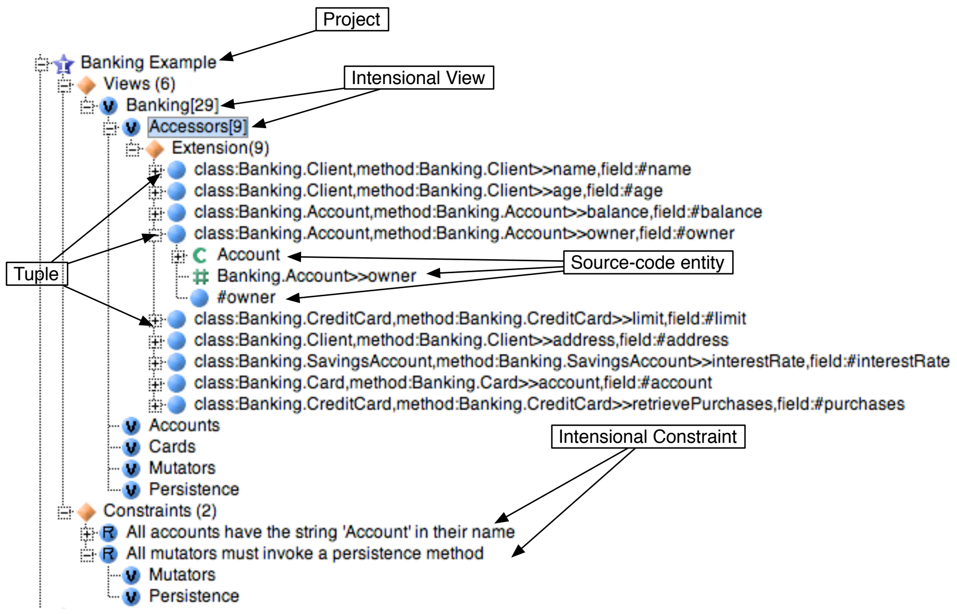
An example of an intension which uses this mechanism is shown in Figure 4.9. This intension is the SOUL version of the Smalltalk intension shown in Figure 4.4. It retrieves all classes which implement a persistence method and which contain a field for which there exists a mutator method but no accessor method. In this intension, we refer to three other intensional views, namely *Mutators*, *Accessors* and *Accounts*. The first line of this intension retrieves the combination of all classes and fields for which there exists a mutator method. Recall that the `?` indicates an anonymous variable in SOUL, similar to _ in Prolog. In the second line, the bindings of 'class' and 'field' get restricted to those for which there does not exist an accessor method. Line 3 further restricts these classes to those which belong to the implementation of accounts.

This symbiosis between intensional views and SOUL does not restrict the potential of the logic language. Our integration of intensional views with the logic language does not come at the cost of limiting the expressiveness of SOUL. Instead, full unification features, backtracking, and so on remain applicable. Such use of unification is illustrated in Figure 4.10. This figure shows two examples of a logic condition which uses the *Accessors* intensional view. Due to the unification, the integration of intensional views as a logic predicate in SOUL can be used to retrieve all accessor methods of a field `age` (first example) or all accessors for the field `account` belonging to the class `Card` (second example).

4.3 Tool-support for Intensional Views

In this section, we present the different tools which belong to the IntensiVE tool suite. We demonstrate how our different tools support the creation and manipulation of intensional views and constraints, and how they integrate with the surrounding Smalltalk environment by means of a concrete implementation in Smalltalk of the banking system. As such, we provide a number of examples of how structural source-code regularities in the banking system are documented using IntensiVE. Moreover, we also show how, using our tool suite, inconsistencies between these documented structural source-code regularities and the actual source code of the banking system can be discovered.

4.3.1 Tree-representation of Intensional Views



```
Banking Example
  Views (6)
    Banking[29]
      Accessors[9]
        Extension(9)
          class:Banking.Client,method:Banking.Client>>name,field:#name
          class:Banking.Client,method:Banking.Client>>age,field:#age
          class:Banking.Account,method:Banking.Account>>balance,field:#balance
          class:Banking.Account,method:Banking.Account>>owner,field:#owner
            Account
            Banking.Account>>owner
            #owner
          class:Banking.CreditCard,method:Banking.CreditCard>>limit,field:#limit
          class:Banking.Client,method:Banking.Client>>address,field:#address
          class:Banking.SavingsAccount,method:Banking.SavingsAccount>>interestRate,field:#interestRate
          class:Banking.Card,method:Banking.Card>>account,field:#account
          class:Banking.CreditCard,method:Banking.CreditCard>>retrievePurchases,field:#purchases
      Accounts
      Cards
      Mutators
      Persistence
  Constraints (2)
    All accounts have the string 'Account' in their name
    All mutators must invoke a persistence method
      Mutators
      Persistence
```

Figure 4.11: The intensional views and constraints defined on the Banking system.

Intensional views (and constraints) imposed over a system are represented in IntensiVE in a tree-like manner. For the banking system, this tree-representation is shown in Figure 4.11. The tree representation consists of five kinds of nodes:

- **Projects**: projects are the top-level nodes which group a number of intensional views and constraints. For example, in Figure 4.11 we see one such project, namely *Banking Example* which groups all the intensional views and constraints imposed over our banking system;

- **Intensional Views**: the child nodes of a project are the intensional views which belong to that project. Notice that one intensional view (in our example the *Banking* intensional view) can have other intensional views as its children. This nesting relationship between intensional views is used to indicate the parent view of an intensional view. In Figure 4.11 we thus see that the *Accessors*, *Mutators*, *Accounts*, *Cards* and *Persistence* intensional views have the *Banking* intensional view as their parent view. Using a drag & drop mechanism, a developer can alter the parent view of an intensional view, duplicate intensional views and move intensional views from one project to another;

- **Tuples**: tuples represent the entities belonging to the extension of an intensional view. In Figure 4.11, the tuples belonging to the *Accessors* intensional view are shown. Note that this set of tuples is not stored in the intensional view but is calculated using the intension of the view;

- **Source-code entities**: these are the actual entities in the source-code of a system which are associated with an attribute of a tuple. In the tree-representation, they are depicted as the children of a tuple. In the figure, we see the source-code entities belonging to one of the tuples of the *Accessors* view. More specifically, we see the `Account` class, the method `owner` implemented by this class and the field `owner`;

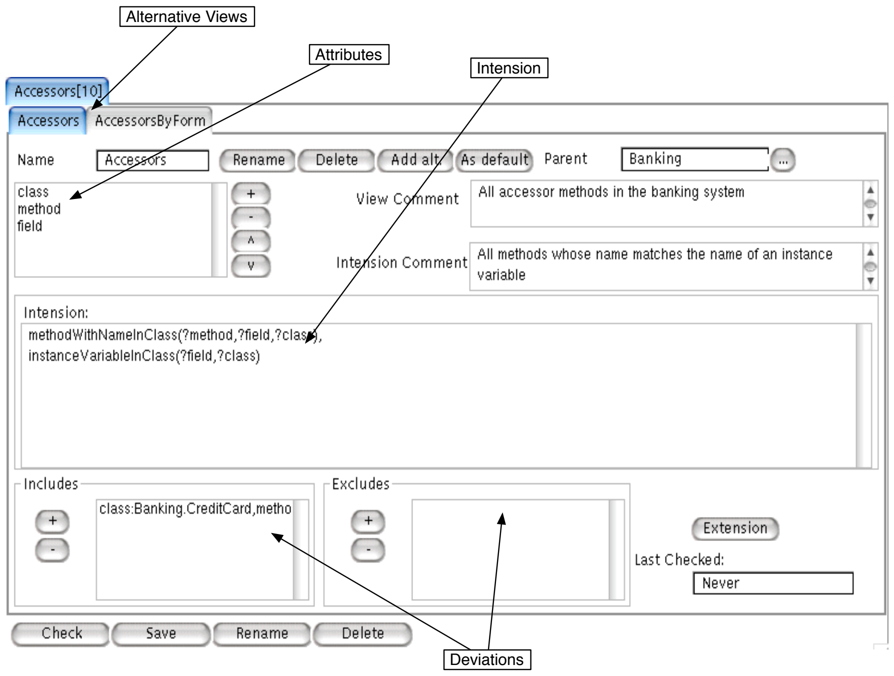
- **Intensional Constraints**: The other children nodes of a Project are the constraints which are imposed over the intensional views. For instance, in the figure we see two such constraints, namely "All accounts must have string 'Account' in their name" and "All mutators must invoke a persistence method". The child nodes of an intensional constraint are the intensional views which the constraint is imposed on.

In addition to presenting a structured overview of the different intensional views and constraints specified over a software system, this tree-representation also serves as the main point of access to the tools offered by our tool suite. Since this tree-representation is integrated with the surrounding IDE, it is possible to directly browse the source code of any of the source-code entities associated with an attribute of a tuple. Moreover, the tools for defining and manipulating intensional views and constraints are also accessible from within this tree: by clicking on an intensional view or constraint, the corresponding editor for the view or constraint will be opened.

4.3.2 Intensional View Editor

An intensional view is manipulated using the *Intensional View Editor*. This sub-tool – displaying the *Accessors* intensional view – is shown in Figure 4.12. The tool provides facilities for defining the attributes of an intensional view (in the case of the *Accessors* view these are 'class', 'method' and 'field'), its intension, comments which detail the purpose of the view and which provide an explanation of the intension, and the sets of tuples which are explicitly included in or excluded from the extension of an intensional view. The different alternative views of an intensional view are indicated by a tabbing mechanism.

In Chapter 3, we already mentioned that the *Accessors* intensional view has two alternative views. The first alternative view is expressed in terms of the naming convention which



Name: Accessors
Parent: Banking

class
method
field

View Comment: All accessor methods in the banking system

Intension Comment: All methods whose name matches the name of an instance variable

Intension:

```
methodWithNameInClass(?method,?field,?class),
instanceVariableInClass(?field,?class)
```

Includes: class:Banking.CreditCard,metho

Excludes:

Last Checked: Never

Figure 4.12: The Intensional View Editor

states that all accessor methods start with the prefix “get-”. While this naming convention is widely adopted in Java programs, Smalltalk developers generally use another convention when implementing an accessor method, namely they require that the selector of the accessor method matches the name of the field it is accessing. Since we implemented our running example in Smalltalk, we document this naming convention instead of the Java variant. The intension of the first alternative (which is shown in Figure 4.12), expressed using SOUL as a query language (with the LiCoR library) is:

```
methodWithNameInClass(?method,?field,?class),
instanceVariableInClass(?field,?class)
```

This intension is a straightforward translation of the naming convention. The first logic condition selects all methods, together with their name (which is bound to the logic variable ‘field’) and their class. The second condition verifies whether there exists a field in the class with as name the value bound to the logic variable ‘field’. Note that the parent view of the *Accessors* view is the *Banking* intensional view (whose intension can be found in Figure 4.2). As such, evaluating the above intension yields all tuples from the banking system which respect this naming convention.

In our implementation of the banking system, there exists one exception to this intension: the accessor method for the `purchases` field on the `CreditCard` class is implemented by the method `retrievePurchases`. While it does not follow the coding convention which characterizes the alternative view, we do want to include it in the extension of the intensional view. As such, we explicitly document this exception by adding the tuple to the *Includes* set:

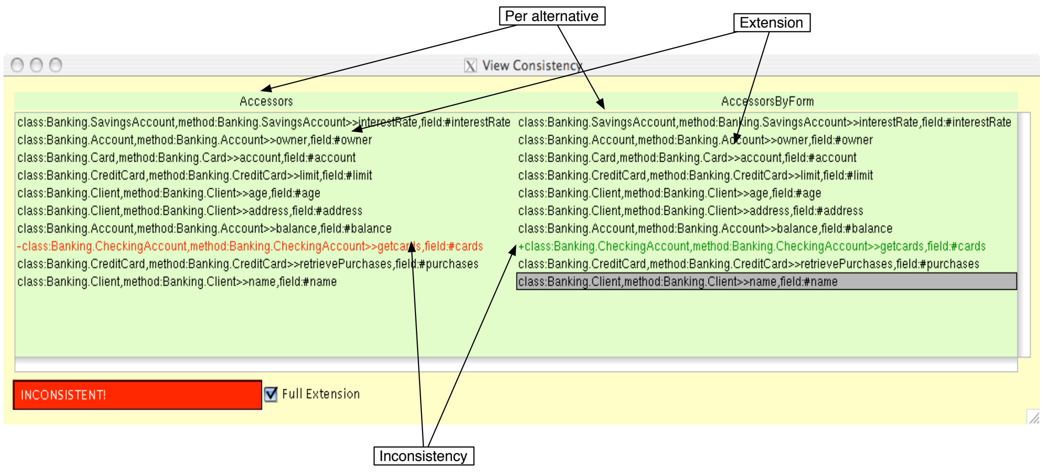
$$(\textbf{class} : \texttt{CreditCard}, \textbf{method} : \texttt{retrievePurchases}, \textbf{field} : \texttt{purchases})$$

The second alternative view (behind the second tab in Figure 4.12, but not actually shown in the figure) is based on the idiomatic implementation of accessor methods. Such accessor methods are typically characterized by being implemented as a single statement returning the value of a field. We express this intension using the following SOUL intension:

```
statementsOfMethod(statements(<?statement>),?method),
instanceVariableInClass(?field,?class),
equals(?statement, return(variable(?field)))
```

This intension consists of three conditions. The first condition restricts the bindings of the 'method' variable to those whose statement-list consists of a single statement. This is achieved by the `statementsOfMethod` predicate. This predicate binds the statements belonging to a method to a functor `statements` which has one argument, namely the list of actual statements. By specifying that this list is a singleton (lists in SOUL are delimited by $<>$), only methods are considered which consist of a single statement. In the second condition, all fields 'field' of a class 'class' are selected. The third condition links the first and second condition, by requiring that the single statement of the method is a return statement which returns the value of the instance variable bound to the SOUL variable 'field'.

4.3.3 Checking extensional consistency of a view



Accessors

```
class:Banking.SavingsAccount,method:Banking.SavingsAccount>>interestRate,field:#interestRate
class:Banking.Account,method:Banking.Account>>owner,field:#owner
class:Banking.Card,method:Banking.Card>>account,field:#account
class:Banking.CreditCard,method:Banking.CreditCard>>limit,field:#limit
class:Banking.Client,method:Banking.Client>>age,field:#age
class:Banking.Client,method:Banking.Client>>address,field:#address
class:Banking.Account,method:Banking.Account>>balance,field:#balance
-class:Banking.CheckingAccount,method:Banking.CheckingAccount>>getcards,field:#cards
class:Banking.CreditCard,method:Banking.CreditCard>>retrievePurchases,field:#purchases
class:Banking.Client,method:Banking.Client>>name,field:#name
```

AccessorsByForm

```
class:Banking.SavingsAccount,method:Banking.SavingsAccount>>interestRate,field:#interestRate
class:Banking.Account,method:Banking.Account>>owner,field:#owner
class:Banking.Card,method:Banking.Card>>account,field:#account
class:Banking.CreditCard,method:Banking.CreditCard>>limit,field:#limit
class:Banking.Client,method:Banking.Client>>age,field:#age
class:Banking.Client,method:Banking.Client>>address,field:#address
class:Banking.Account,method:Banking.Account>>balance,field:#balance
+class:Banking.CheckingAccount,method:Banking.CheckingAccount>>getcards,field:#cards
class:Banking.CreditCard,method:Banking.CreditCard>>retrievePurchases,field:#purchases
class:Banking.Client,method:Banking.Client>>name,field:#name
```

Figure 4.13: The Extensional Consistency Inspector

When multiple alternative views are defined for a single intensional view, it becomes possible to verify the extensional consistency of the intensional view, as we have explained in Chapter 3. Extensional consistency implies that all alternative views of a given intensional view yield the same set of tuples. In our tool suite we provide the *Extensional Consistency*

Inspector: a sub-tool that allows for verification of this type of constraint for a given intensional view. Next to informing a user whether the intensional view is extensionally consistent or not, this tool reports on the inconsistencies between the different alternatives of the intensional view.

Figure 4.13 shows this tool opened on the *Accessors* intensional view. Recall from the previous section that this view has two alternatives: one alternative expressing the naming convention that the selector of an accessor method must match the name of the field it is accessing; the other alternative expressed in function of the idiom that an accessor method consists of a single statement returning the value of a field.

The inspector shows a column for each alternative view of the intensional view. The first column corresponds to the alternative based on the naming convention; the second column represents the idiom-based alternative. For each alternative, the different rows in the column show the tuples belonging to that alternative. Tuples present in the extension of all alternatives are indicated in black. Discrepancies between the different alternatives are indicated in red and green. If a tuple is part of the extension of some of the alternatives, but not of all, it is indicated in green (and prefixed with a '+' sign) in the columns of the alternatives to which it belongs; if it is absent from the extension of a certain alternative view, it is indicated in red (and prefixed with a '-').

When verifying the extensional consistency of the *Accessors* intensional view, we see that there is one tuple which is marked as a possible inconsistency between the two alternative views of the *Accessors* view.

More specifically, the tuple

$$(\textbf{class} : \texttt{CheckingAccount}, \textbf{method} : \texttt{getcards}, \textbf{field} : \texttt{cards})$$

is marked as part of the second alternative (green), but it is not part (red) of the first alternative of the *Accessors* view. Thus, while the accessor method represented by this tuple is implemented as a single statement which returns the value of `cards` field, it does not respect the naming convention which dictates that the selector of the accessor method must correspond to the name of the field. In order to provide easy access to the source-code entities which violate the extensional consistency of an intensional view, the extensional consistency inspector provides facilities for browsing the source-code related to the inconsistencies (by right-clicking on an inconsistency).

4.4 Tool-support for Intensional Constraints

In this section we provide an overview of how our tool suite provides support for imposing constraints on the intensional views defined on a software system. We demonstrate how unary intensional constraints and binary intensional relations are defined using our tool suite, how the predicate expressing the actual constraint is specified and how our tool can provide feedback concerning the discrepancies between the documented constraint and the source code of the system.

4.4.1 Specifying the predicate of a constraint

Similar to the intension of an intensional view, both the Smalltalk as well as the SOUL query language can be used in order to express the predicate of an intensional constraint. To illustrate how this mechanism works in practice, we take a detailed look at the predicate of the binary intensional relation: "all mutator methods must invoke a persistence method". This binary relation is expressed in terms of two intensional views, namely the *Mutators* and *Persistence* view. Mathematically, this relation can be expressed as:

$$\begin{aligned}&\forall\, t_1 \in \textit{Mutators} :\\ &\quad \exists\, t_2 \in \textit{Persistence} :\\ &\quad\quad calls(t_1.method, t_2.method)\end{aligned}$$

In Chapter 3, we stated that the predicate of a binary intensional relation must be a function that takes as input two arguments, namely two tuples, and returns a boolean expressing whether the constraint holds or not for a given pair of tuples. For the above relation, this predicate is a function p such that:

$$p(t_1, t_2) = calls(t_1.method, t_2.method)$$

We can express this binary predicate in SOUL as follows:

```
methodCallsMethod(?mutator.method, ?persistence.method)
```

This predicate uses the `methodCallsMethod` predicate of LiCoR which, given two methods, returns true if the body of the first method contains a call to the second method. Notice that the name of the actual tuples which are inputted into the predicate can be chosen by a developer. For instance, the tuples belonging to the *Mutators* view are designated `mutator`; those belonging to the extension of the *Persistence* view are referred to as `persistence`. In order to access the value of a specific attribute of a tuple, a dot operator is used. E.g. `mutator.method` retrieves the value of the 'method' attribute from a given tuple.

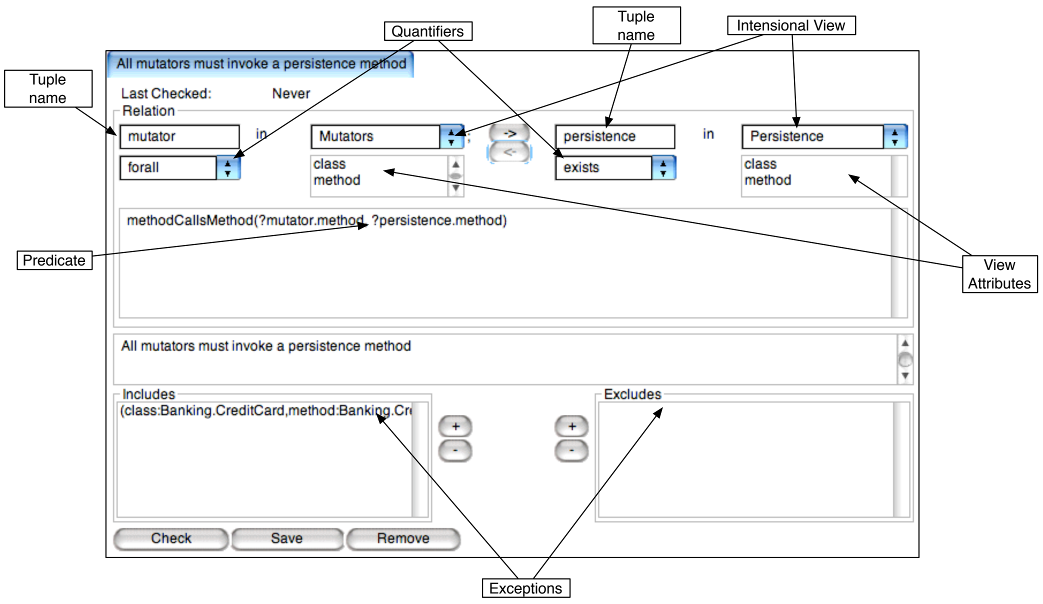
Similarly, the same binary predicate can be expressed in Smalltalk by the following Smalltalk program:

```
(mutator valueFor:#method) compiledMethod
   sendsSelector:
      (persistence valueFor:#method) compiledMethod selector
```

The above Smalltalk program receives two tuples from the IntensiVE environment, namely `mutator` and `persistence`, and returns a boolean depending on whether the mutator method contains a call to the persistence method. To this end, the program retrieves the value for the 'method' attribute from the `mutator` tuple, extracts its compiled method and verifies whether this compiled method contains a message send of the selector corresponding with the selector of the persistence method.

4.4.2 Intensional Relation Editor

The *Intensional Relation Editor* is the main tool in our tool suite for defining and manipulating constraints over intensional views. Both unary intensional constraints, as well as binary



methodCallsMethod(?mutator.method, ?persistence.method)

All mutators must invoke a persistence method

(class:Banking.CreditCard,method:Banking.Cr

Figure 4.14: The Intensional Relation Editor

intensional relations are defined using this tool. Figure 4.14 shows the *Intensional Relation Editor* opened on the binary intensional relation expressing that “all mutator methods must invoke a persistence method”. This sub-tool of our tool suite allows for defining a constraint over intensional views by specifying:

- **Intensional Views**: depending on whether one is defining a unary constraint or a binary relation, respectively one or two intensional views are provided;
- **Quantifiers**: one or two quantifiers, depending on the number of intensional views involved in the constraint;
- **Exceptions**: a user can explicitly document that for certain tuples or pairs of tuples the relation does or does not hold;
- **Predicate**: the expression which is verified for the tuples belonging to the extension of the intensional views involved in the relation;
- **Tuple names**: the names which will be used to indicate the tuples in the predicate.

For instance, for the binary intensional relation which is shown in Figure 4.14, we specified two intensional views, namely the *Mutators* and *Persistence* views with as quantifiers respectively $\forall$ and $\exists$. We documented one exception to the relation, namely:

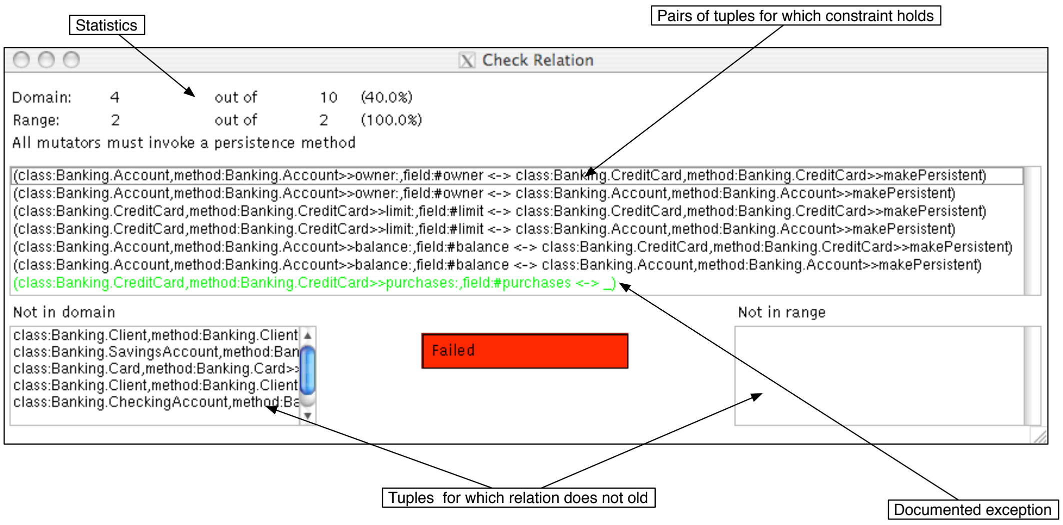
$$((\textbf{class} : \texttt{BankCard}, \textbf{method} : \texttt{purchases}, \textbf{field} : \texttt{purchases}), \_)$$

This exception makes use of the wildcard mechanism (‘_’) and states that for the tuple describing the mutator method of the `purchases` field, implemented by the `BankCard` class, the relation explicitly holds. As the predicate for this binary intensional relation, we used the SOUL variant which we explained in the previous section.

Unary intensional constraints are declared in a similar way as the binary intensional relation we demonstrated in this section. This can be done by selecting ‘None’ as the second intensional view in the intensional relation editor. In such a case, the relation editor requires a unary predicate instead of a binary predicate; exceptions to the relation consist of single tuples rather than of pairs of tuples.

Although the *Intensional Relation Editor* could straightforwardly be extended such that it can deal with n-ary relations, we did not incorporate such constraints in our tool suite since the need has not arisen from the case studies we conducted.

4.4.3 Verifying validity of an intensional constraint


Statistics
Pairs of tuples for which constraint holds
Check Relation
Failed
Tuples for which relation does not old
Documented exception

```
Domain:    4        out of     10    (40.0%)
Range:     2        out of     2     (100.0%)
All mutators must invoke a persistence method
```

```
(class:Banking.Account,method:Banking.Account>>owner:,field:#owner <-> class:Banking.CreditCard,method:Banking.CreditCard>>makePersistent)
(class:Banking.Account,method:Banking.Account>>owner:,field:#owner <-> class:Banking.Account,method:Banking.Account>>makePersistent)
(class:Banking.CreditCard,method:Banking.CreditCard>>limit:,field:#limit <-> class:Banking.CreditCard,method:Banking.CreditCard>>makePersistent)
(class:Banking.CreditCard,method:Banking.CreditCard>>limit:,field:#limit <-> class:Banking.Account,method:Banking.Account>>makePersistent)
(class:Banking.Account,method:Banking.Account>>balance:,field:#balance <-> class:Banking.CreditCard,method:Banking.CreditCard>>makePersistent)
(class:Banking.Account,method:Banking.Account>>balance:,field:#balance <-> class:Banking.Account,method:Banking.Account>>makePersistent)
(class:Banking.CreditCard,method:Banking.CreditCard>>purchases:,field:#purchases <-> _)
```

Not in domain

```
class:Banking.Client,method:Banking.Client
class:Banking.SavingsAccount,method:Ban
class:Banking.Card,method:Banking.Card>>
class:Banking.Client,method:Banking.Client
class:Banking.CheckingAccount,method:Ba
```

Not in range

Figure 4.15: The Relation Consistency Inspector

Similar to verifying extensional consistency of the alternatives of an intensional view, our tool suite offers support for checking whether a given constraint imposed on a number of intensional views holds with respect to the source code of a system. To this end, we supply the *Relation Consistency Inspector*. Figure 4.15 shows this sub-tool opened on the binary intensional relation we discussed in the previous section.

First of all, this tool shows the pairs of tuples from the cross-product of the extensions of the *Mutators* and *Persistence* intensional views for which the predicate holds (pane in the center of the figure). Note that explicit exceptions to this constraint are shown in either green (include) or red (exclude). More interestingly, the tool also provides detailed information concerning tuples which the relation does *not* hold for. For a binary intensional relation, the tuples from the first intensional view which do not occur in the relation are shown in the *Not in domain* box; the tuples from the extension of the second view for which the relation does not hold are presented in the *Not in range* box. For instance, in our example relation it is reported that the *Not in domain* box contains the following tuples belonging to the extension *Mutators* intensional view:

(**class**:`Client`,	**method**:`age:`,	**field**:`age`),
(**class**:`SavingsAccount`,	**method**:`interestRate:`,	**field**:`interestRate`),
(**class**:`Card`,	**method**:`account:`,	**field**:`account`),
(**class**:`Client`,	**method**:`address:`,	**field**:`address`),
(**class**:`CheckingAccount`,	**method**:`cards:`,	**field**:`cards`),
(**class**:`Client`,	**method**:`name:`,	**field**:`name`)

Each of these tuples does not respect the structural source code regularity stating that a mutator method must invoke a persistence method and are thus reported by our tool. Note that the *Not in range* box is empty: this indicates that every persistence method in the banking system is possibly invoked.

When opened on a unary intensional constraint, the *Relation Consistency Inspector* will show similar behavior: tuples which the predicate holds are shown in the middle pane of the tool for. Tuples for which the constraint fails are shown in the *Not in domain* box.

4.5 Verifying a collection of intensional views

Until now we have focused on the parts of our tool suite which allow defining intensional views and constraints on these views, and which support the verification of the validity of a single intensional view or constraint with respect to source code. However, as the number of intensional views and constraints increases, it becomes unfeasible to verify them one by one manually. In order to give a higher-level overview of the intensional views and constraints imposed on a system, and their validity with respect to the source code of that system, we provide two tools:

- A visualization tool which presents a graphical overview of the intensional views, constraints and their validity;
- An extension to the Smalltalk unit testing framework which aids in incorporating the verification of intensional views and constraints into the regular testing phase of the development cycle.

4.5.1 Visualization tool

The visualization tool is implemented as an extension to the StarBrowser and uses the HotDraw [Bra92] framework for graphical editors to give a visual representation of the different intensional views declared on a software system and the constraints imposed on those intensional views. Figure 4.16 demonstrates this tool applied to the intensional views we defined on our running example system, namely the implementation of a banking system.

In the visualization of the system, each intensional view is represented by an ellipse. Depending on the color (green or red), the depicted intensional view is respectively extensionally

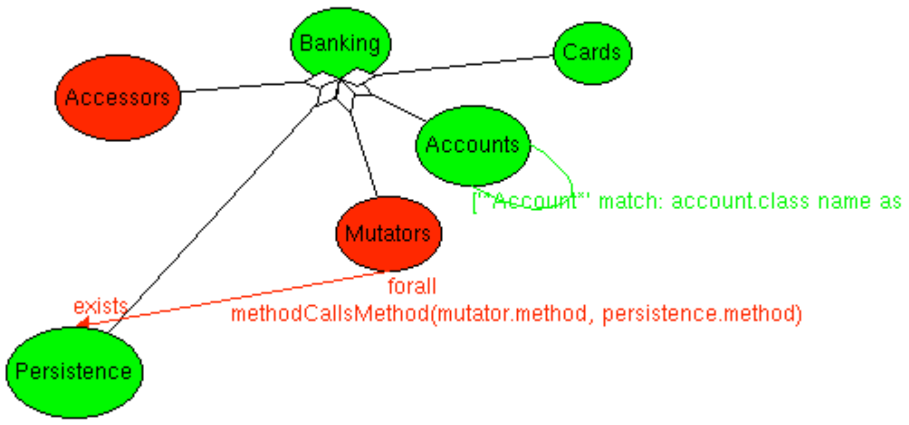

Banking
Cards
Accessors
Accounts
["*Account*" match: account.class name as
Mutators
forall
methodCallsMethod(mutator.method, persistence.method)
exists
Persistence

Figure 4.16: Visualization of the intensional views and constraints defined on the banking system

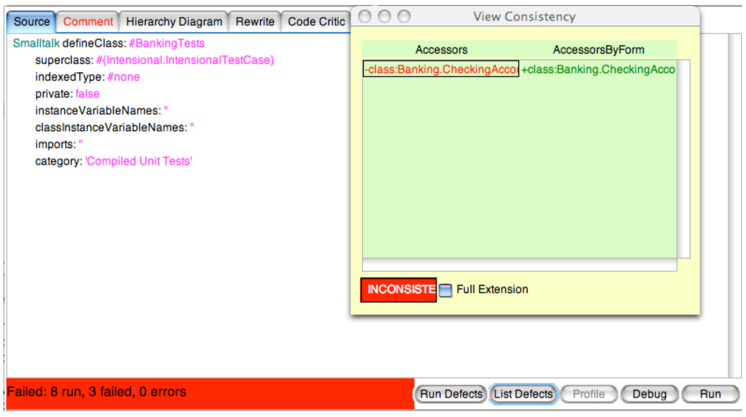
consistent or extensionally inconsistent. For instance, in Figure 4.16, we see that all intensional views except the *Accessors* and *Mutators* view are extensionally consistent. For each intensional view, the parent view is indicated by a line with a diamond on the parent view end. In our example, all intensional views have the *Banking* view as their parent.

Constraints imposed on intensional views are shown as a line between the intensional views involved in the constraint. The color of the line indicates whether the constraint holds or not. In Figure 4.16 we see that the relation between the *Mutators* and *Persistence* intensional view, which we also described in Section 4.4.2, does not hold.

4.5.2 Unit test integration

Another means to verify the consistency of a set of intensional views and intensional constraints is offered by the integration of the IntensiVE tool suite with the Smalltalk unit testing framework. The principle of unit tests [Bec99] stems from eXtreme Programming (XP) in which a "test-driven" philosophy is adopted. Each unit test consists of a small fragment of code which verifies the correct behavior for a single class or method. The general idea behind using unit tests is that, if these small tests are executed frequently, bugs are detected as early as possible during development.

To support unit tests, the Smalltalk environment provides a framework which facilitates defining tests and executing them. A new test case is created by subclassing the `TestCase` class. Separate tests are implemented by creating methods whose selector starts with the prefix "test-". By using assertions, different parts of the behavior of a system can be tested. Moreover, the environment contains a number of enhancements and browsers which run the tests defined on a system and report on failing tests. These test runners then allow a developer



```
Smalltalk defineClass: #BankingTests
	superclass: #{Intensional.IntensionalTestCase}
	indexedType: #none
	private: false
	instanceVariableNames: ''
	classInstanceVariableNames: ''
	imports: ''
	category: 'Compiled Unit Tests'
```

Accessors	AccessorsByForm
-class:Banking.CheckingAcco	+class:Banking.CheckingAcco

```
Failed: 8 run, 3 failed, 0 errors
```

Figure 4.17: Illustration of the integration of IntensiVE with the Unit testing framework.

to access the debugging facilities of Smalltalk in order to correct the errors resulting from the failing tests.

We extended this framework with a new kind of entity, namely *IntensiVE test cases*. An example of this integration is shown in Figure 4.17. An IntensiVE test case is automatically generated from a number of intensional views and the constraints imposed on those views (in our example, it is generated from all the intensional views and constraints we defined on the Banking system). Similar to regular tests cases, an IntensiVE test case can be verified from within the different test runners the Smalltalk environment offers. For each intensional view and constraint, the test case contains a separate test. Executing an IntensiVE test case corresponds to verifying whether the test case's intensional views and constraint are consistent with the implementation. Upon failure of an IntensiVE test case, the *Extensional Consistency Inspector* or *Relation Consistency Inspector* is shown such that a user can remedy the inconsistencies between the structural source-code regularities and the source code of a system.

4.6 Methodology

While the previous sections introduced our tool suite and demonstrated by means of a number of examples how structural source-code regularities can be documented and verified using our tools, in this section we propose a lightweight methodology for using our approach and incorporating it in the development process. Due to the multitude of different kinds of structural source-code regularities, and the diversity of ways these regularities are manifested in the source code of a system, it is outside the scope of this dissertation to provide a detailed overview of how each kind of regularity can be supported by our approach and tool suite.

The methodology we discuss in this section will consist of a number of guidelines for documenting regularities using our approach and for integrating support for these regularities into the development process. We have put this methodology into practice in the case studies which we present in Chapters 5 and 6.

4.6.1 Documenting a structural source-code regularity

We start our methodology by proposing a number of guidelines for documenting a structural source-code regularity using intensional views and constraints. A distinctive feature of our approach is that we explicitly discriminate between the set of source-code entities on which the constraint is imposed on the one hand, and the actual constraint on the other hand.

Creating the intensional view

When documenting a regularity, we start by identifying the set of tuples representing the source-code artifacts which the regularity should apply to. If this set of artifacts is already captured by an intensional view, then the regularity can be documented by imposing an additional constraint on this intensional view. If not, a new intensional view needs to be created that captures the source-code entities. In general, such an intensional view will align with a concept from the problem or solution domain of the system which we wish to document a regularity for. For instance, in the Banking example we encountered certain intensional views which are specific to the domain of banks like *Cards* and *Accounts* but also certain intensional views that align with implementation concepts like *Accessors* and *Persistence*.

Attributes Attributes specified for the intensional view can be considered to be the external interface to the view. They are chosen such that they provide access to the source-code entities involved in the constraint that expresses the regularity. For instance, since we wanted to document a number of regularities governing accessor *methods* in the Banking system, we included an attribute 'method' in the *Accessors* intensional view. This set of attributes can be complemented such that other information related to the concept captured by the intensional view is made accessible. We also added for the *Accessors* intensional view attributes associated with the class on which the accessing method was implemented, as well as the name of the field that was accessed. This way, a tuple can contain contextual information about the concept captured by the view, providing a developer access to the source-code entities related to a certain instance of that concept. However, in order not to clutter the tuples, it is recommended to limit the number of attributes of an intensional view to at most four or five.

Intension When defining an intensional view, the developer also provides an initial intension. Upon execution, this intension must yield the tuples representing the source-code entities which the regularity that is being documented is applicable to. To this end, IntensiVE supports SOUL and Smalltalk as a query language. As we shall illustrate by means of a number of examples later on in this dissertation, the SOUL language seems to be a more suitable candidate to succinctly and declaratively express the intension of an intensional view. However, due to performance reasons, a developer might opt for Smalltalk as the query language whenever the intension tends to be computationally intensive.

As we explain later on in this section, we propose a step-wise process for defining the intensional views and constraints imposed on a system. As a result, if the developer specifies an initial intension that is either too specific or too general, or if some deviations from this initial intension are omitted, this step-wise definition of the documentation will reveal such inconsistencies and aid in refining the definition of the intensional view.

Parent view The parent view of the intensional view presents a means for determining the scope which the intensional view is applicable to. If the concept, or the set of source-code entities, to which the regularity applies is subsumed by an already existing intensional view, then the new intensional view can use this existing intensional view as its parent view. The intension of the newly created view will then serve as a kind of filter, selecting the source-code entities from the extension of the parent view which are of interest to the concept that is documented with the new intensional view. Similarly, if multiple intensional views share some overlapping source-code entities, then these overlapping entities can be captured by a separate parent intensional view with as intension the common part of the intensions of the two views.

The result is a hierarchy of intensional views that can be refactored during the addition or refinement of the set of intensional views on a software system. Intensional views that are situated high in this hierarchy represent more general concepts; the views that are lower in the hierarchy are in general a representation of more specific concepts in the source code of the system. Note that often for a certain software system, a top-level intensional view is created that captures all entities in the software system. All other intensional views defined for this system are directly or indirectly a child of this top-level view. E.g. in the banking system we created a top-level view *Banking system* which the other intensional views on that system are children of.

When to use an alternative view, a unary constraint or a binary relation

While the source-code entities which a regularity is applicable to are documented using an intensional view, the actual regularity is encoded by using either an alternative view, a unary intensional constraint or a binary intensional relation. Although we earlier discussed the applicability of each of these kinds of constraints, we briefly repeat which situation calls for which kind of constraint.

- **Alternative view**: an alternative view is added to the intensional view representing the concept which the regularity is applicable to if the regularity is limited to a single concept in the source code of the system. Moreover, the regularity needs to be both **necessary** and **sufficient**. Not only must the regularity hold for all source-code entities belonging to the intensional view; if a source-code entity adheres to the regularity it must also belong to the extension of the intensional view. If the regularity thus provides an alternative description of a certain concept, it is documented using an alternative view;

- **Unary intensional constraint**: other regularities which are applicable to a single intensional view, but which only provide a **necessary** condition on the entities belonging to the intensional view are documented using a unary intensional constraint. While the regularity must hold for all tuples in the extension of the intensional view, this does not

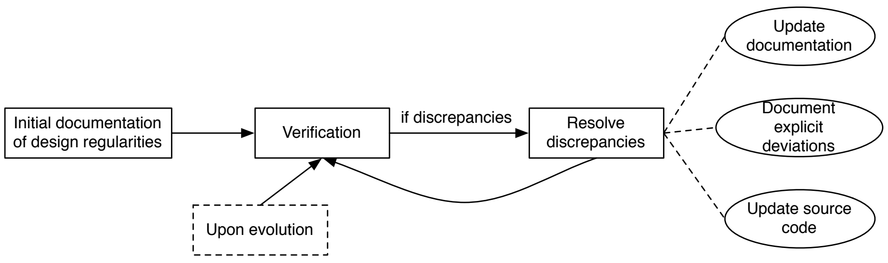


Figure 4.18: A schematic illustration of the process of iteratively refining the documented regularities and the source code.

mean that if an entity respects the regularity, it should be included in the intensional view;

- **Binary intensional relation**: binary intensional relations are used to document regularities that express a dependency between **two** different intensional views.

Similar to the initially defined version of an intensional view, this first version of a constraint on intensional views captures the initial assumption about how the documented regularity manifests itself in the source code.

4.6.2 Co-design of documentation and implementation

After a developer has encoded the initial assumption about how a certain regularity is manifested in the source code (or as we shall discuss in the next section, after the system has evolved) we propose in our methodology the *co-design* of the documented design regularities and the source code in order to synchronize both artifacts. We do not consider the documentation and the source code as two separate entities which are created independently of each other. Rather, our methodology takes the verifiable causal link that our documentation introduces into account. As such, we propose an iterative, simultaneous refinement of the documentation and the source code as a means to synchronize the design regularities and the implementation. This process of co-design illustrates the prescriptive use of the documentation created using intensional views and constraints. The various intensional views and constraints defined over a system serve as a description of the design regularities that govern a system. During the co-design of the regularities and the implementation, both this description as well as the actual source code are refined.

More concretely, we suggest the following process (a schematic overview of which is shown in Figure 4.18):

1. After initial documentation of a regularity is created, the developer verifies the consistency of the involved intensional views and constraints using the *Extensional Consistency Inspector* and the *Relation Consistency Inspector*;

2. If the intensional view/constraint is not consistent, the developer inspects the tuples which are indicated by the tools as discrepancies, along with the source code of the

entities in the tuples. Based on this information, the developer can resolve the discrepancies by (a combination of) the following actions:

- Discrepancies can be caused by the intension of the intensional view, or the predicate (or quantifiers) of the constraint being incorrect or imprecise. In such a case, the active documentation that the developer created for the regularity does not properly represent the regularity in the source code. In order to rectify this situation, the developer needs to update the intensional view or constraint;
- Our tools can report a number of discrepancies about which the developer decides that they are deviations to the intension of the view, or that the constraint is not applicable to them. In order to synchronize the documentation of the regularity and the source code, the developer can explicitly mark them as deviations to the intension or as exceptions to the constraint using the $Incl$ and $Excl$ sets;
- An inspection of the source code of the entities belonging to a discrepancy can also reveal that the discrepancy was not caused by the documentation, but rather that the source code did not properly adhere to the documented regularity. In other words, the discrepancy was the result of an infringement of the regularity. The developer can solve this inconsistency by altering the source code such that it correctly obeys the regularity.

3. After the documentation and/or source code have been updated, the developer repeats the process of verification and resolution until the intensional views and constraints are consistent with respect to the source code.

After this iterative process, in which the definition of the intensional views/constraints and the source code will be refined, the documented design regularity and the actual implementation will be synchronized. In other words, this step-wise refinement results in that the verification of the documentation and the source code no longer yields discrepancies.

4.6.3 Co-evolution of documentation and implementation

Once the developer has created active documentation of the structural source-code regularities that govern a system using intensional views and constraints, it is imperative that upon evolution of the system this documentation and the source code remain mutually consistent. Thus, the documentation and the source code must *co-evolve*. This co-evolution is illustrated in Figure 4.19. Upon changes in either regularities or source code, the support for co-evolution allows keeping both artifacts synchronized.

Depending on the phase in the implementation cycle, this co-evolution of the documented design regularities and the source code illustrates both the descriptive as well as the prescriptive use of the intensional views and constraints. During the earlier phases of the implementation, the different design regularities can still be volatile and subjective to evolution. As a consequence, the intensional views and constraints serve as a kind of descriptive documentation for these regularities. During the later phases of the implementation, we envision a more prescriptive use of the documented regularities. In such a scenario, the documentation serves

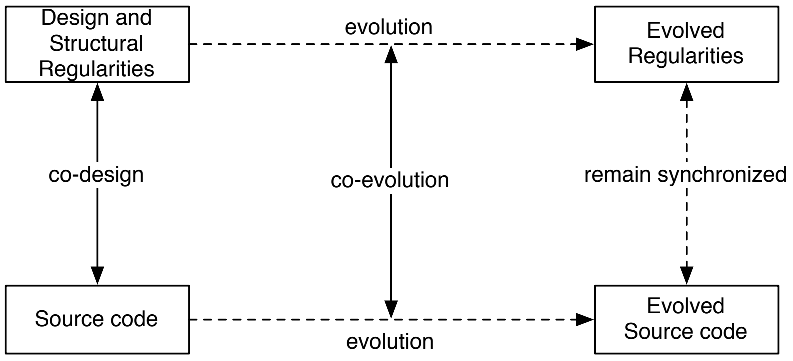


Figure 4.19: A schematic illustration of co-evolving the documented structural source-code regularities with the source code.

as a means to verify whether evolution of the source code does not break any of the structural source-code regularities that govern a system.

Structural regularity testing

As a first means to support this co-evolution, we wish to detect discrepancies between the intensional views/constraints and the source code early on during development. Therefore we propose a "test-often" philosophy that is similar to the unit testing methodology underlying eXtreme Programming [Bec99]. This methodology describes a process in which a number of unit tests are written that each verify that a specific and small part of the system behaves correctly. By incorporating these tests suites in the development environment and by rigorously and consistently executing these tests each time considerable changes have been made to the implementation, the developer can detect bugs in the implementation at an early stage. Similarly, the intensional views and constraints that are defined over a system can be considered to be *structural regularity tests*. By continuously verifying the validity of the documented regularities with the implementation, discrepancies between both these artifacts will be brought to the developer's attention at an early stage during development.

Beck attributes a number of properties to unit tests which are also applicable to our approach. The intensional views and constraints provide a means to document the different regularities that govern the system on a *by need* basis. The verification of the documentation and the resolution of possible discrepancies can be performed when a developer feels the need for this, e.g. when a number of changes have been made to the implementation. What's more, since intensional views and constraints provide documentation for structural source-code regularities, they aid in making the system more *understandable*. First off, this better understanding is aided by that our approach makes the regularities explicit to the developers. Second, by enabling the consistent propagation of structural source-code regularities in the source code, this makes the source code more uniform resulting in improved readability. Furthermore, since the test-driven methodology makes it possible to detect infringements of regularities early on, this aids in strengthening *confidence* about the source code following the different regularities that govern the system.

Conflict resolution and support for co-evolution

Whenever during the implementation phase our tool suite (for instance our tool that integrates with the unit testing framework) identifies a discrepancy between a documented regularity and the source code, the same iterative process as described in Figure 4.18 is applicable in order to resolve the discrepancy. Not only do we advocate co-design to create documentation for structural source-code regularities, it also lies at the heart of supporting *co-evolution* between the source code and the documented regularities. Whether the structural regularities or the source code of the system evolves, the iterative process prescribes a means of updating the documentation of the regularities and/or the source code of the system such that both artifacts are synchronized again. Due to the integration of our tools with the surrounding development environment, this process is aided since both the documentation as well as the implementation can be accessed from within our tool suite.

4.7 Comparison with the state of the art

Now that we have introduced our model of intensional views and constraints (Chapter 3) and our prototype implementation of this model, namely IntensiVE, we situate how our work relates to the state of the art which we discussed in Chapter 2.

4.7.1 Classification mechanisms

The construct of an intensional view is a classification mechanism that groups a set of source-code entities that are conceptually related. Although our approach is unique in that it employs a classification mechanism in order to provide support for the documentation and verification structural source-code regularities, this classification mechanism bears many resemblances to those that we discussed in Section 2.2.

Virtual software classifications The concept of an intensional view is inspired both by the virtual classifications of De Hondt as well as by the classification model employed by Kim Mens in their doctoral dissertations. However, our work can be considered as a generalization of these two classification models. Classifications in our approach do not consist of a set of software artifacts but rather group a set of tuples representing an instance of a certain concept in the source code. As we shall illustrate in Chapter 6 this representation using tuples allows conceptual entities in the source code to be documented more succinctly. Moreover, we extended the virtual classifications both with a scoping mechanism (parent views) as well as with constructs to explicitly cope with exceptions to the query describing the classifications. We also generalized the notion of constraints on classifications such that our model supports relations between an arbitrary number of classifications.

Concern Manipulation Environment Although the goal of the concern manipulation environment, namely supporting the manipulation of concerns throughout the development cycle, differs from our goal of documenting, verifying and co-evolving structural source code regularities, the classification models of both approaches share a number of commonalities. Both approaches propose a classification model in which classifications are first-class entities. Moreover, they both support the definition of classifications by means of a query and provide constructs to impose constraints over the classification. However, due to the difference in focus, the emphasis in our approach lies on the different types of constraints we offer (alternative views, n-ary constraints) which aid in expressing structural source-code regularities and verifying conformance of the regularities with respect to the source code of a system.

Cosmos Cosmos provides a specialized classification mechanism containing numerous concepts such as classes, properties and topics, which from the point of view of supporting structural source-code regularities during the development process seem not to be applicable. In the terminology used by Cosmos, the notion of an intensional view best aligns with the concept of *instance* concerns. Perhaps the most notable difference between both classification mechanisms is that, while our approach offers the definition of classifications using a description (intensional), Cosmos only provides support for enumeration-based classifications (extensional).

Conceptual Models Similar to Cosmos, Conceptual Models provides a classification mechanism which only supports the extensional definition of the classifications. Using Conceptual Models, this is done by manually specifying a set of lines of source code that belong to a module. As a result, both the Cosmos classification mechanism as well as conceptual models seem less suitable for supporting the evolution of structural source-code regularities since, upon evolution of a system, the entities belonging to a classification would always have to be updated manually as opposed to intensionally defined classifications, in which the set of entities belonging to a classification can be re-computed.

Concern Graphs Concern graphs offer a similar classification mechanism as intensional views, in that both techniques make use of a query-based definition of the set of entities belonging to a classification. Furthermore, similar to our methodology, concern graphs advocate an iterative process where the query describing a concern graph is refined in multiple steps. In contrast to our approach however, concern graphs lack the possibility to express constraints on the concern model, rendering this technique less suitable for supporting documentation and verification of structural source-code regularities.

4.7.2 Support for structural source-code regularities

In addition to discussing how our approach relates to the different classification mechanisms we discussed earlier, we also provide a comparison of our model, tool suite and methodology with other approaches that share a similar goal of supporting structural source-code regularities throughout the implementation process. Recall from Chapter 2 that we discriminated between three groups of such approaches: code checkers, meta-programming languages and architectural conformance checkers. In what follows, we dedicate a section to each of these groups and discuss the parallels and differences of intensional views and constraints with these approaches.

Code checkers

In Section 2.3.1 we discussed a large group of approaches that provide dedicated support for the verification of system-wide regularities such as stylistic constraints, commonly made mistakes, code smells, and so on. With the exception of a number of approaches, such as FindBugs and PMD, the set of regularities that can be verified with these approaches is fixed, resulting in these approaches being less suitable for supporting more domain-specific or application-specific regularities.

In contrast, our approach aims at offering a general platform for documenting and verifying structural source-code regularities. Our approach does not focus on any kind of regularity in particular, but rather supports the documentation of various kinds of structural source-code regularities. Furthermore, our approach takes into account that regularities are liable to evolve over time. Since code checkers provide support for regularities which are rather universal in nature and which do not change over time, this need for evolving the documented regularities is not prevalent with such approaches.

However, the generality our approach offers comes at the trade-off that specialized techniques – such as these code checkers – can provide more dedicated support. For example the P^3 system offers, next to facilities for detecting common coding errors, support for automatically correcting such errors. Furthermore, since these approaches focus on verifying specific regularities, the analysis they perform of the source code can be highly optimized resulting in efficient run-time and accuracy.

Meta-programming languages

In Section 2.3.2 of Chapter 2 we discussed a number of approaches that offer a meta-programming language which reasons about a reification of the source code of a system and allows the implementation of checkers for different kinds of structural source-code regularities.

In general, these approaches are complementary to our work. We presented our approach in Chapter 3 independently of the query language used to specify the intension of a view or the predicate of a constraint. Similarly, our model of intensional views and constraints also abstracts about the underlying source-code representation. As such, the reification of the source code about which each of these meta-language approaches reasons can be used as the universe $\mathcal{U}$ of source-code entities over which intensional views are defined. The actual meta-programming language can be adapted such that it is used to express the intension of an intensional view or the predicate of a constraint. By means of a plugin architecture, our tool suite IntensiVE actively supports this integration of other query languages and source-code representation.

We put this integration into practice in the implementation of IntensiVE, where we opted to use one such meta-programming approach, namely SOUL, as the primary means for expressing the intension of a view. SOUL offers the advantage over other meta-programming systems such as e.g. CCEL that it provides a full reification of the source code, including the method bodies. Furthermore, the declarative paradigm, which SOUL is an instantiation of, has been shown to be an expressive means for reasoning about software.

The combination of our approach and tool suite with meta-programming languages aids in leveraging some of the disadvantages of the latter. First of all, intensional views and constraints provide a structured means to documented regularities and thus creating *explicit* documentation of these regularities. Moreover, this documentation is a first-class entity in the surrounding development environment and the verification of this documentation can be integrated into the development process. Second, our methodology and tool suite actively support the co-design and co-evolution of structural source-code regularities with the source code of a system. As such, the structured means of documenting regularities and the integration with the development process that our approach offers is complemented with the extensive querying facilities offered by meta-programming languages.

Architectural conformance checkers

The third group of related approaches we discussed in Chapter 2 are architectural conformance checkers. These approaches provide support for verifying a high-level description of the architecture or the design of a system with the source code. While our approach can also be used to express more low-level regularities such as programming idioms, these architectural conformance checkers show a number of similarities with intensional views. First, intensional views can align with a high-level concept in the source code. The constraints on intensional views can be used to document dependencies or interactions between these high-level concepts. Furthermore, some of these architectural conformance checkers (such as two-tier programming and virtual software classifications) also advocate – similar to our approach – a methodology in which the architectural description is co-evolved with the implementation of the system. In what follows we briefly revisit the architectural conformance checkers we discussed earlier and provide a more detailed comparison with our approach.

Reflexion Models Reflexion models present a technique in which a developer creates a high-level view of the different concepts in a system and the calling relations between these concepts. The approach supports the verification of this high-level view with respect to the source code of the system. Our approach shows a number of similarities with reflexion models. The different concepts which appear in the high-level model used by reflexion models align with the intensional views a developer specifies over a software system; dependencies between such high-level concepts boil down to the use of relations in our approach. Besides that our approach supports a richer set of constraints than calling relations between conceptual units, the main difference with reflexion models lies in the mapping of the high-level model onto the source code of the system. While in our approach this mapping is implicit from the intensional definition of a view, in reflexion models a developer needs to explicitly and manually map a high-level concept to a set of source-code entities.

Tool support for design patterns The approach of Florijn et al. relies on the concept of a *fragment*: a design element along with a number of roles. Using these fragments, a design pattern can be documented. An instance of a design pattern can be documented by manually mapping the different roles of the design pattern description to actual entities from within the source code of a system. By comparing the prototypical description of a design pattern with the actual instantiation, inconsistencies can be detected and an automatic resolution strategy can be proposed.

As we will demonstrate in Chapter 5, our approach lends itself to documenting and verifying different regularities underlying the implementation of design patterns. Although we do not only focus on the interactions between the different participants of a design pattern, but also capture naming conventions, implementation patterns, and so on, our approach shares a number of similarities with that of Florijn. Each of the different roles of the design pattern correspond to an intensional view; dependencies between roles are documented using intensional relations. However, while Florijn's approach requires an explicit, extensional mapping of the different design pattern roles to the entities in the source code, the different views documenting the role of a design pattern capture the source-code entities belonging to that role by means of an intensional description.

Ptidej Similar to the work of Florijn, Ptidej provides support for the identification and verification of design patterns in the source code of a system. In Ptidej, a design pattern is documented by means of a meta-model that expresses how the different roles of the pattern relate. By also modeling the actual architecture of the system using the meta-model, Ptidej is able to detect design patterns, identify infringements against the proper implementation and provide corrective measures.

While Ptidej documents the prototypical structure of a design pattern, and uses this documentation in order to support instantiations of this design pattern, we approach the problem from a different angle. In contrast to Ptidej, we do not document the abstract design pattern. Rather, we document the structural source-code regularities underlying a *particular instantiation* of the design pattern. This results in that our approach requires more effort to document the regularities of a design pattern, since each instantiation of the pattern requires a separate set of intensional views and constraints. Conversely, our approach offers the advantage that it allows us to document and verify instantiation-specific naming conventions, dependencies, and so on.

Two-tier programming Our model of intensional views and the associated methodology can be considered to be another instantiation of two-tier programming. Using our approach, the underlying model of the software on which we define intensional views is the first-order tier. The structural source-code regularities we document using intensional views and relations form a high-level representation of certain concepts in the software and can thus be considered the second-order tier. In our approach, there exists no explicit association mapping. Rather, the intensional views which make up the second-order tier are defined in terms of the entities in the first-order tier, resulting in an implicit mapping. An additional benefit of this implicit mapping is that, upon addition or removal of software entities from the first-order tier, there is no need to update the association mapping manually. What's more, this implicit mapping mechanism renders it possible to identify inconsistencies automatically between the first-order and second-order tier.

4.8 Discussion

Extensibility of the tool suite

Although our tool suite at the moment only supports the software model of Smalltalk and Java, on which intensional views and intensional constraints can be imposed using Smalltalk and SOUL as a query language, it has been developed with extensibility in mind. From its inception, one of the main motivations behind our implementation is to provide a prototype research environment which allows easy experimentation with different underlying software models and query languages. In order to ease this experimentation, our implementation has been conceived as a framework that can support many other software models and query languages.

Choice of Smalltalk as implementation language

Notwithstanding the popularity of the Eclipse platform and the Java language for implementing research-oriented software engineering tools, we opted for our implementation to use Smalltalk and the Cincom VisualWorks environment. In particular, the openness of the language and development environment, the reflective capabilities offered by the Smalltalk meta-object protocol and the dynamic typing of the Smalltalk language render it an ideal candidate for prototyping IntensiVE.

- **Dynamic typing** In addition to speeding up the rapid prototyping of the IntensiVE tool suite, the use of a dynamically typed language offers a number of significant advantages. Foremost, the combination of dynamic typing with the "everything is an object" philosophy underlying Smalltalk eases the integration of different software models into the IntensiVE tools. Without having to take static typing information into account, it is possible to associate any kind of Smalltalk object to an attribute of a tuple. This makes it possible to readily incorporate other software models into IntensiVE.

- **Reflective capabilities** In addition to making it possible to use Smalltalk as a query language for IntensiVE, the reflective capabilities of Smalltalk also eased implementing the IntensiVE framework. A first situation in which this reflection proved to be useful was for configuring the different customizations of the framework. For instance, the part of the tool suite that initializes the different evaluators for the supported query languages uses the first-class representation of classes and methods in order to retrieve all applicable evaluators, configure those evaluators and integrate them into the user interface of IntensiVE.

  A second situation in which we rely on the reflective capabilities of Smalltalk is in the implementation of the persistence mechanism of IntensiVE. Although the image based development of VisualWorks is suitable for managing and manipulating an object representation of the intensional views defined over a system, in order to distribute a set of intensional views, a more persistent representation of these views is necessary. This representation is obtained by representing a volatile intensional view or constraint as a Smalltalk program: intensional views and constraints can be compiled into a method that, upon execution, yields the original view or constraint. This representation of data as a program has as an advantage that the standard distribution and versioning facilities of Smalltalk are applicable.

- **Open implementation** Smalltalk offers an open implementation of both the development environment as well as the language and standard libraries. The openness of the environment rendered it possible to provide a tight integration of our tool suite with the IDE. This is for instance demonstrated by the fact that our tool suite is integrated with the Smalltalk unit testing framework and that from within our tools it is possible to easily browse the source code related to the documented structural source-code regularities. Moreover, the open implementation of the Smalltalk language makes it possible to extend the language itself. In IntensiVE, we have demonstrated this feature by extending the Smalltalk namespace mechanism such that it can be used to access intensional views as first-class language entities. Finally, the fact that the implementation of the entire class library is available and extensible from within Smalltalk itself, makes it possible to extend this class library straightforwardly. For instance, we were able to extend the standard `Collection` framework with the same set-theoretical operations that the `Extension` class from the IntensiVE framework implements. This allowed us to, internally in the implementation of IntensiVE, transparently use extensions and collections.

SOUL versus Smalltalk

In this chapter, we have illustrated how both the reflective capabilities of Smalltalk as well as the SOUL logic language can be used as a query language in IntensiVE. In the remainder of this dissertation, the majority of the intensions we define will be formulated in the latter language, SOUL. Although Smalltalk is a versatile tool to reason about Smalltalk programs, in practice the use of a declarative paradigm appears more suitable in order to express the intension of an intensional view. This suspicion is strengthened by the observations of [Wuy01, WM06, MMW01] and by some of our earlier experiments, presented in [MKPW06]. Especially the symbiosis which SOUL offers between the logic paradigm and the underlying Smalltalk language render it possible to succinctly express the intension of an intensional view. This is illustrated by the intensions shown in Figures 4.2 and 4.3. These intensions for the *Banking* and *Mutators* intensional views, expressed in Smalltalk, are considerably more verbose than their SOUL variants, which can be found in Figures 4.6 and 4.7. However, in some cases, the use of Smalltalk over SOUL as a query language can be preferred. Especially in situations where the unification and backtracking properties of SOUL are not needed, the use of Smalltalk as a query language can result in a considerable performance increase.

Impact of the model on the implementation and vice versa

The model of intensional views and the associated tool suite IntensiVE have been developed in an iterative way. We did not start by implementing the complete version of the model as it is presented in Chapter 3. Instead, we iterated numerous times over both the model and the implementation, in which decisions concerning the model often led to changes in the implementation. Conversely, over time the model of intensional views evolved in order to cope with practical restrictions we encountered while performing experiments with the tool suite.

For instance, in earlier versions of the model of intensional views, the extension of an intensional view did not consist of a set of tuples but – more simply – out of a set of source-code entities. In order to document a concept such as the accessor methods in our banking system, in the older version of intensional views, we would create an intensional view *Accessor methods* which groups all accessor methods. Similarly, we would create an intensional view *Banking methods*, which classifies all the methods belonging to the banking system. This latter view would then be the parent view of the *Accessor methods* intensional view. As we would also be interested in the classes concerning to the accounts, we would also create an intensional view *Account classes* which has the *Banking classes* view as its parent. The fact that a single intensional view only contained specific source-code entities instead of a more structured representation of a concept in the source code of a system, resulted in a proliferation of intensional views. Moreover, this also resulted in an excessively large number of relations. E.g. we would have a relation expressing that all *Banking methods* must be implemented by a class in the *Banking classes* intensional view. This practical problem gave rise to an overhaul of the model of intensional views, in which we use tuples as a means to represent a certain concept in the source code of a system.

Similarly, the notion of deviations to the intension of an intensional view, and the exceptions on constraints were introduced in our model, after a number of practical experiments pointed out that in practice, the documented structural regularity would not quite fit with the situation in the source code and that it should be possible to mark certain source-code entities explicitly as exceptions to the general rule. Likewise, the observation that the intensions of multiple intensional views over the same system would contain duplicated code, led to the conception of parent views.

4.9 Conclusion

In this chapter we introduced IntensiVE, our prototype tool suite that is a practical realization of the model of intensional views and constraints which we presented in Chapter 3. Whereas our model makes abstraction of the query language used to define intensions and predicates, our implementation supports the definition of intensional views and constraints over both Java and Smalltalk programs using Smalltalk and SOUL as a query language. Furthermore, we presented our lightweight methodology which provides a set of practical guidelines for documenting structural source-code regularities using IntensiVE, and for integrating the documented regularities into the development process. Also in this chapter, we provided a comparison of our approach with the state of the art.

The IntensiVE tool suite was developed with the goal of validating our approach for documenting and verifying structural source-code regularities and maintaining the causal link between these regularities and the source code. As such, we will use our prototype tool in Chapters 5 and 6, where we present the experiments that serve as a validation of the work we put forward in this dissertation.

Chapter 5

Documenting and verifying regularities underlying design patterns

In order to validate the approach we advance in this dissertation, we demonstrate how the model of intensional views and the associated tool suite can be used to document, verify and evolve a wide range of structural source-code regularities present in the source code of a software system. This validation consists of two parts:

- In this chapter we demonstrate that our approach is expressive and versatile enough to create verifiable documentation for various kinds of structural source-code regularities. As such, we report on an experiment in which we documented different source-code regularities underlying the implementation of object-oriented design patterns [GHJV95];

- In Chapter 6, we report on an experiment in which we applied our approach and methodology to create verifiable documentation of the regularities governing three different systems. In particular, we documented the regularities underlying our own tool suite IntensiVE, the extensible wiki system SmallWiki [DRW05, Ren03] and DelfSTof [MT05, TM04], a framework for performing formal concept analysis on source code. The goal of this experiment is to illustrate how our approach and methodology support the co-design and co-evolution of the documented regularities and the source code of each of those three systems.

5.1 Design Patterns

Design patterns are well-known, reusable solutions to commonly occurring problems in the design of a piece of software [GHJV95]. Design patterns are not limited to sketching a technical solution for a specific design problem but also encompass a description of the context of the pattern, its applicability and the impact it has on the design of the software. At the technical level, a design pattern's description contains a number of participants which specify the different source-code entities involved in the pattern and how these participants collaborate. As such, a design pattern is not an actual part of the design of a piece of software but rather a template for a proven solution. A particular instantiation of a design pattern will associate a source-code entity (e.g. class, method) or a set of entities (e.g. class hierarchy) with each of the participants of the pattern; the collaborations between the participants describe how the different source-code entities which implement such participants interact.

The use of design patterns gives rise to a number of structural source-code regularities. A first kind of regularity corresponds to the requirements which arise from the collaborations between the participants of the design pattern. Second, in the implementation of a design pattern a developer often uses stylistic regularities such as naming conventions in order to convey knowledge about the different participants of the pattern. For example, a developer can identify the concrete visitors in the implementation of a Visitor design pattern by using the string "Visitor" in the class name. Moreover, also implementation regularities such as language idioms, inheritance constraints, and so on are introduced to regulate the implementation of the different participants of a pattern.

While a certain structural source-code regularity is often specific to a particular instantiation of a design pattern, other instances of the same design pattern in most cases use a similar regularity. For instance, one possible naming convention underlying the implementation of a Visitor design pattern can be that the *visit* methods all start with the *prefix* "visit-". While this naming convention is specific to the single instance of the pattern, other instances often use a different but similar convention by e.g. consistently using a *suffix* "-visit" or any other naming convention, when implementing a visit method.

Due to this wealth of structural source-code regularities, design patterns form an ideal case study to demonstrate the range of regularities which can be expressed and verified using intensional views. As such, in what follows we discuss how, for a number of design patterns, we can document the structural source-code regularities underlying instances of those patterns using intensional views. We do this based on one particular instantiation of each design pattern, taken from the implementation of the Smalltalk Open Unification Language (SOUL), the standard libraries of VisualWorks Smalltalk or from the implementation of our own tool suite, IntensiVE.

5.2 Experimental set-up

In the following section we discuss nine design patterns from [GHJV95] and discuss a manifold of structural source-code regularities that underly these design patterns. It is not our goal to give a complete overview of all structural source-code regularities that arise from the use of design patterns, but rather to demonstrate different kinds of regularities that occur when applying design patterns and how these regularities are supported by our approach and tool suite.

For each of the different design patterns, our discussion is structured as follows:

- We start each section by giving a short overview of the design pattern's rationale, applicability and mechanics;

Design Pattern	Section
Visitor	5.3.1
Abstract Factory	5.3.2
Facade	5.3.3
Adapter	5.3.4
Chain of Responsibility	5.3.5
Observer	5.3.6
Template Method	5.3.7
Proxy	5.3.8
Builder	5.3.9

Table 5.1: Overview of the design patterns discussed in this experiment

- We discuss a number of structural source-code regularities which stem from the use of the design pattern. For each regularity, we mention the kind of regularity to which it belongs, based on the classification we presented in Section 2.1.3. While some of these regularities express a naming convention or another kind of stylistic constraint used to improve comprehension and readability of the source code, other regularities express invariants about how the different participants of a design pattern should interact, and yet others regulate the usage of the entities making up an instantiation of the design pattern. Note that this set of regularities is not final: we by no means claim that the set of regularities we discuss encompasses all regularities which underly a specific design pattern. Nor do these regularities encompass all possible kinds of regularities that can be expressed with intensional views and constraints;
- One concrete instantiation of the design pattern is discussed, taken from either the implementation of SOUL, the implementation of IntensiVE or from any of the standard frameworks and libraries included with VisualWorks Smalltalk;
- For this concrete instantiation, we discuss how each of the regularities manifests itself in the implementation and how we can codify this instance of the regularity using intensional views and intensional constraints.

5.3 Documenting design patterns

5.3.1 Visitor

Design pattern

Summary The Visitor design pattern presents a means to implement a set of operations which can be performed on a certain object structure. The goal of this pattern is to detach these operations from the actual classes which implement the object structure, such that it is possible to alter the set of operations with relative ease. An overview of this design pattern is shown in Figure 5.1. The participants of the Visitor pattern are divided into two class hierarchies: the hierarchy of the `Element` class which implements the elements belonging to

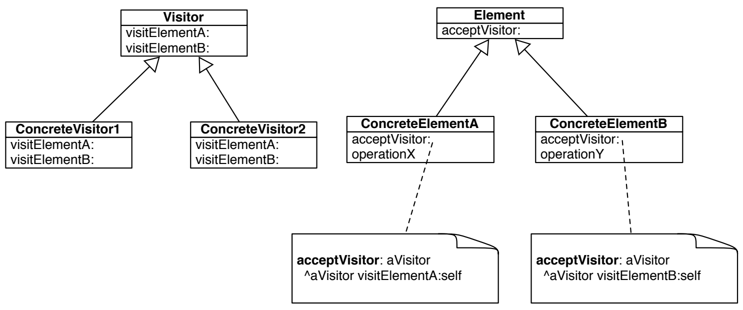


Figure 5.1: Class diagram of the Visitor design pattern

the object structure and the hierarchy of the `Visitor` classes that each represent a specific operation on the object structure. Each of the `Elements` implements an `accept` method which takes as input a Visitor object and calls, via a double-dispatch protocol, a corresponding `visit` method. This visit method, implemented on a `Visitor` class, represents the operation for the specific Element it visits.

Regularities The implementation of an instantiation of the Visitor design pattern gives rise to a number of structural source-code regularities:

1. (stylistic) The accept methods implemented in the `Element` hierarchy are characterized by an intention-revealing naming convention. E.g. they are consistently named `accept` or `acceptVisitor`, or any other similar naming scheme;

2. (stylistic) The single argument of each accept method is consistently named (e.g. in Smalltalk, the argument of all accept methods is named `visitor` or `aVisitor`);

3. (idiom) The accept methods are implemented by a double-dispatch protocol. This idiomatic implementation consists of a single statement which contains a call to the single argument of the accept method with a self reference as the argument of the call;

4. (stylistic) The classes in the `Visitor` hierarchy all follow a similar naming scheme. E.g. they contain the string "Visitor" in their name;

5. (stylistic) The visit method, implemented by a `Visitor` follows the naming convention that it uses both an intention-revealing name (e.g. contain "visit" in the method name) as well as that the name of the visit method contains the `Element` it is visiting (e.g. a visit method visiting `ElementA` would be named `visitElementA`);

6. (requirement) All accept methods must contain an invocation of a visit method.

While the majority of these regularities present a stylistic regularity which aids in using a proper naming scheme when implementing an instantiation of the Visitor design pattern, regularity 3 demonstrates the use of an idiomatic implementation. Moreover, regularity 6 illustrates a design requirement which must be respected in order for the instantiation of the pattern to be able to function correctly.

Instantiation

The instantiation of the Visitor design pattern, based on which we documented the aforementioned structural source-code regularities can be found in the implementation of the SOUL logic language. In SOUL, an object structure is used to represent the abstract syntax tree of a logic program. Due to the number of different operations which are applicable to this abstract syntax tree, the developers of SOUL have opted to use the Visitor pattern in order to implement these operations. For instance, the retrieval of the logic variables in a SOUL program, the process of renaming certain entities and the lexical addressing scheme of SOUL have been implemented by means of a visitor over the abstract syntax tree.

Documenting the regularities

Before we start documenting the above regularities, we assume that there exists an intensional view named *SOUL* with as attributes 'class' and 'method' which groups all the classes and their methods of the implementation of the SOUL language.

The first three regularities are all related to the concept of an accept method. As such, we document these regularities by creating an intensional view *Accept methods* with attributes 'class' and 'method' and as parent view the *SOUL* view. The first two regularities each express a different definition of the concept of an accept method: (1) all methods named `accept:`, (2) all methods which have as the name of their first argument `aVisitor` are – in the SOUL implementation of the Visitor pattern – considered to be an accept method. We thus document these regularities by declaring two alternative views for the *Accept methods* intensional view. These two alternatives are characterized by the following intensions (expressed using SOUL as the query language):

1. `methodWithName(?method,[#accept:])`

2. `argumentsOfMethod(<variable(aVisitor)>,?method)`

Each of these alternatives aligns with one of the first two structural source-code regularities. The first intension captures all methods named `accept:` thus encoding regularity (1); the second intension selects all methods of which the name of the unique argument is `aVisitor`, which is the concretization of regularity (2) in the implementation of the Visitor pattern in SOUL. By encoding these two regularities using alternative views for the *Accept methods* view we ensure that, upon verifying extensional consistency of the different alternatives, violations to these regularities will be detected, as long as the accept method respects at least one of the two regularities. Only in the case that an accept method respects none of these regularities, will our tool be unable to report it as an infringement. However, in such a situation it becomes quite unlikely that the method is indeed an accept method. To summarize, the *Accept methods* intensional view thus expresses the constraint that all methods named `accept:` must have a single argument named `aVisitor` and vice versa.

The other regularity that is applicable to accept methods, namely regularity (3) which states that all accept methods should be implemented using a double-dispatch protocol is encoded by means of the following unary intensional constraint:

$\forall$ *accept* $\in$ *Accept methods* :

```
statementsOfMethod(statements(<?statement>),?accept.method)),
argumentsOfMethod(<?argument>,?accept.method),
equals(?statement,
 return(send(?argument,?mess,<variable(self)>)))
```

This unary constraint expresses that the implementation of all accept methods must consist of a single statement, that performs a double dispatch to the unique argument of the accept method. Notice that, in line with our methodology, we did not implement this regularity as an additional alternative view for the *Accept methods* view but rather as a separate unary constraint. While all accept methods should be implemented using a double-dispatch, the opposite is not true. As such, the double-dispatch is a necessary requirement for an accept method but not a sufficient requirement.

Regularity (4) expresses a naming convention that must be upheld by the elements in the `Visitor` hierarchy. In SOUL, these visitors are implemented as subclasses of the `SimpleTermVisitor` class. Regularity (4) manifests itself in the SOUL implementation by requiring that all Visitor classes end with the suffix “-Visitor”. In order to capture this stylistic constraint, we first create an intensional view *Visitors* with attributes ‘class’ and ‘method’ and as parent view *SOUL*. For this intensional view, we specify a single intension:

```
classInHierarchyOf(?class,[SimpleTermVisitor])
```

This intension restricts all bindings of tuples from the *SOUL* intensional view to those for which the class is bound to any class in the hierarchy of `SimpleTermVisitor`.

Based on this intensional view, we document regularity (4) using the following unary intensional constraint:

$\forall$ *visitor* $\in$ *Visitors* :

```
'*Visitor' match:  (visitor valueFor:  #class) name
```

The predicate of the above constraint provides an illustration of the use of Smalltalk as a query language. This predicate verifies whether the binding of the ‘class’ attribute of a tuple from the extension of the *Visitors* intensional view is suffixed with the string “-Visitor”. We implement this predicate by matching the method name of a visit method with the regular expression `*Visitor`. Verifying consistency of the above constraint results in the set of all subclasses of `SimpleTermVisitor` which do not respect regularity (4).

In order to encode regularity (5), which states that all visit methods are characterized both by a naming convention which reveals their intent as well as that they should contain the name of the element they are visiting in their method name, we define an intensional view *Visit methods*. This view, which has the *Visitors* view as a parent, has a single attribute ‘method’ and consists of two alternative views with as intension:

1. `['*Visit:' match: ?method selector asString]`

2.
```
classInHierarchyOf(?element,[AbstractTerm]),
 ['*', ?element name asString asLowercase,'*'
      match: ?method selector asString]
```

The first alternative encodes the first part of regularity (5) which states that all visit methods are characterized by a naming convention. In the case of the Visitor in SOUL, this naming convention expresses that all visit methods should end with the suffix "-Visit". The second alternative codifies that the name of the method should contain the name of an element which is visited. Verifying consistency of these alternative views will report on any tuples consisting out of a visit method and its implementing class which either does not end with the suffix "-Visit" or which does not contain the name of a visited element in its method name.

Finally, in order to express regularity (6), we use a binary intensional relation. This regularity, that expresses that all accept methods must contain a call to a visit method, is defined in terms of the *Accept methods* and *Visit methods* intensional views. More concretely, we define this relation as:

$\forall\, accept \in$ *Accept methods* :
 $\exists\, visit \in$ *Visit methods* :
  `methodCallsMethod(?accept.method, ?visit.method)`

The relation holds if for all accept methods, there exists a visit method such that the method body of the accept method contains a call to the visit method. If an accept method does not contain such a call, it is reported to be a possible inconsistency between the structural source-code regularity and the instantiation of the Visitor design pattern in the implementation of SOUL[1]. Such accept methods in which the call to the visit method is missing can be an indication of an error in the implementation of the accept method.

5.3.2 Abstract Factory

Design pattern

Summary The goal of the Abstract Factory design pattern is to provide an interface for constructing a family of different objects without having to refer to their actual implementing classes. As such, it is possible to interchange the family of objects without having to adapt the client code. This design pattern is illustrated in Figure 5.2. In the figure, we have two hierarchies of products namely `AbstractProductA` and `AbstractProductB`. For both of these hierarchies, there exist two groups of subclasses which each form a family of objects. For instance, `AbstractProductA` has two subclasses: `ProductA1`(which belongs to a first family of objects that is indicated by a light shade of grey) and `ProductA2` (which belongs to another family indicated by the darker shade of grey). For each of these families of objects, a separate factory exists which constructs instances of the objects in the family.

[1]Notice that we only verify whether the accept method contains a call to the visit method. While this allows us to identify accept methods where this call is missing, it does not give us any guarantees that the visit method will actually be called at run-time.

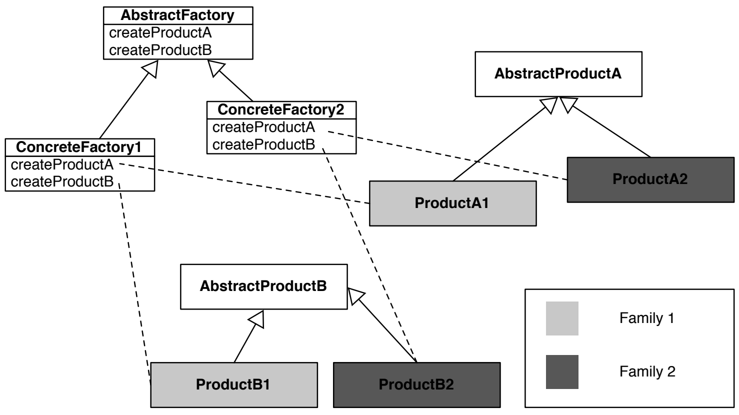


Figure 5.2: Class diagram of the Abstract Factory design pattern

E.g. the class `ConcreteFactory1` that creates products of the first family of objects. If the client needs to change the family of objects that is used, this can be achieved by using another factory.

Regularities For the Factory design pattern, we consider the following four regularities:

1. (stylistic) A Factory should have an intention-revealing name (e.g. contain the string "Factory" in its class name);

2. (stylistic) The methods of a Factory that constructs a Product are characterized by a naming scheme that indicates that it is a factory method (they are prefixed with for example "construct-", "create-" or "make"). Moreover, this method should contain the name of the product it produces;

3. (idiom) The factory method is implemented as a single statement returning a product;

4. (usage) Only a Factory is allowed to create instances of a product.

Instantiation

As was the case with the instantiation we used to illustrate the Visitor design pattern, the example of the Abstract Factory pattern comes from the implementation of SOUL. In SOUL, the parser constructs the elements of the abstract parse tree representation by using an abstract factory.

Documented regularities

Regularity (1) manifests itself in the SOUL implementation by requiring that all Factory classes contain the string "Factory" in their class name. This regularity is encoded using our approach by creating an intensional view *Factories* with as attributes 'class' and 'method'.

This intensional view has the *SOUL* view – which we described above – as its parent view. It consists of a single alternative view with as intension:

```
classBelow(?class,[Factory])
```

The intension selects all the (direct or indirect) subclasses of the `Factory` class from the implementation of SOUL. Regularity (1) is encoded by the unary intensional constraint:

$\forall\, factory \in$ *Factories* :
`'*Factory*' match:  (factory valueFor:#class) name asString`

This regularity is quite similar to regularity (4) of the Visitor design pattern, which we discussed above. Upon verification of this regularity, our tool suite reports on any subclasses of `Factory` which do not contain the string "Factory" in their class name.

Regularities (2) and (3) can be implemented using the concept of alternative views. In the implementation of the Factory design pattern in SOUL, these factory methods are characterized by being implemented in the protocol `terms`[2] as well as the naming scheme which dictates that the name of a factory method consists of the the prefix "make-" along with the name of the product it creates. Moreover, all factory methods in SOUL are implemented by the same idiom: they consist of a single statement which returns the product class which is being constructed[3]. In order for us to express these regularities, we create an intensional view *Factory methods* which groups all methods that construct a product. This intensional view has as attributes 'class' and 'method' and is a child of the *Factory* view we defined above. The two alternative views we specify for the *Factory methods* view have as intension:

1.
   ```
   methodInProtocol(?method, [#terms]),
   Products(?product),
   ['make', [?product name] match: ?method selector asString]
   ```

2.
   ```
   statementsOfMethod(statements(<?statement>), ?method),
   Products(?product),
   equals(?statement, return(variable([?product name])))
   ```

The first intension expresses the concretization of regularity (2) in SOUL: it collects all the methods in the implementation which are classified in the protocol `terms` and the name of which consists of the prefix "make-" and the name of a product. Note that in this intension, the second condition uses the fact that intensional views are first-class entities in the SOUL query language by referring directly to the *Products* intensional view. While we do not specify *Products* here, it is defined as all the subclasses of the `AbstractTerm` class. The second alternative documents regularity (3): upon evaluation of this intension, all the methods in SOUL will be selected which consist of a single statement that returns a product. Since, due to extensional consistency, these two intensions must yield the same set of entities, all

[2]In Smalltalk, methods are annotated with a protocol

[3]The instantiation of the Factory pattern in SOUL does not returns instances of Products, but rather the classes that represent a Product.

factory methods which do respect the naming convention, but which do not adhere to the implementation idiom and vice versa can be detected.

In order to document regularity (4), which encodes that only Factories are allowed to instantiate products, we first need to define an intensional view which groups all the clients of the products of the abstract factory in SOUL. This intensional view, named *ProductClients*, has two attributes ‘class’ and ‘method’ and as a parent view the *SOUL* view. It consists of a single alternative view with as intension:

```
methodReferencesClass(?method,?product),
Products(?product,?)
```

As such, it will collect all the methods which contain a direct reference to a product. Notice that this intension is defined in terms of the *Products* view. Since in the implementation of SOUL only the abstract factory is allowed to reference the actual products, we express this regularity by means of the following binary intensional relation:

$$\begin{array}{l}\forall\, referent \in \textit{ProductClients} :\\ \quad \exists\, factorymethod \in \textit{Factory methods} :\\ \qquad \texttt{referent = factorymethod}\end{array}$$

This intensional relation verifies that all clients of a product are in fact a factory method. If a method external to the implementation of the Factory references a product, it is identified as being a possible error. We could also have implemented this regularity using the following intensional relation:

$$\begin{array}{l}\nexists\, referent \in \textit{ProductClients} :\\ \quad \forall\, factorymethod \in \textit{Factory methods} :\\ \qquad \texttt{referent} \sim= \texttt{factorymethod}\end{array}$$

Notice that both intensional relations are equivalent and upon verification, result in the same set of discrepancies.

5.3.3 Façade

Design pattern

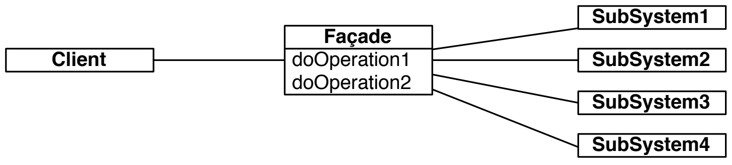


Figure 5.3: Class diagram of the Façade design pattern

Summary The Façade design pattern (illustrated in Figure 5.3) is used to implement a simple, common interface for a group of more complex subsystems. This way, the client code of the subsystems becomes less coupled to those subsystems, since they do not have to communicate with these systems but use a single point-of-entry of communication, namely the Façade. Furthermore, the Façade hides the complexity of the subsystems from the clients use the services offered by the subsystems. Moreover, the use of the Façade pattern makes it easier to adapt or interchange the subsystems which are being used.

Regularities While a number of structural source-code regularities underly the Façade design pattern, we only discuss one such regularity here. In particular, we will document the structural regularity which states that all clients of the Façade are prohibited from directly using the subsystems for which the Façade provides an interface. If this usage regularity is violated, a client of the Façade is improperly applying the design pattern, resulting in a loss of the advantages the Façade provides.

Instantiation

The concrete instantiation of the Façade design pattern we use in order to demonstrate how we can document the above structural source-code regularity is taken from the implementation of the compiler infrastructure of VisualWorks Smalltalk. The class `Compiler` is a Façade providing an interface for the scanner, the parser, the actual Smalltalk compiler, the decompiler and the code generator of VisualWorks Smalltalk.

Documented regularities

In order to document the regularity underlying the Façade design pattern, we create two intensional views, namely *Façade clients* and *Façade subsystems*. The first intensional view (with one attribute 'class') groups all the classes in the Smalltalk image which use the `Compiler` Façade. This view consists of a single alternative view with as intension:

```
methodInClass(?method,?class),
methodReferencesClass(?method,[Compiler])
```

This intension captures all classes which implement a method that refers to the Façade class `Compiler`.

The *Façade subsystems* intensional view classifies all the classes which the Façade provides an interface for. This view has an attribute 'class' and consists of one alternative view with as intension:

```
packageWithName(?package,['System-Compiler-Public Access']),
classInPackage(?class,?package),
not(equals(?class,[Compiler]))
```

Upon execution, this intension captures all the classes in the package 'System-Compiler-Public Access', which contains all classes involved in this instantiation of the Façade design pattern. Since the class `Compiler` (which is the Façade) is also part of this package, but is not a subsystem of the Façade, we included the last condition of the intension, namely `not(equals(?class, [Compiler]))`. As a result, the `Compiler` does not get included in the extension of the *Façade subsystems* intensional view. Another option would have been to document the `Compiler` class as an explicit deviation to the intension by adding the tuple (**class** : `Compiler`) to the excludes list of the *Façade subsystems* view. We opted however for the solution of incorporating the exclusion of the `Compiler` class in the intension of the *Façade subsystems* since we consider it to be an integral part of that intension.

Based on the above two intensional views, we can implement the structural source-code regularity which expresses that clients of the Façade are not allowed to directly use the subsystems of the Façade by means of the following binary intensional relation:

$\forall\, client \in$ *Façade clients* :
  $\nexists\, subsystem \in$ *Façade subsystems* :
    `classReferencesClass(?client.class, ?subsystem.class)`

The predicate of this binary intensional relation verifies whether the client of the Façade directly references a class which implements a subsystem of the Façade. The result of verifying the above constraint using the *Relation Consistency Inspector* tool of IntensiVE is that this tool will report any clients of the Façade for which the predicate of the relation is true, i.e. clients which do refer to a subsystem of the Façade.

5.3.4 Adapter

Design pattern

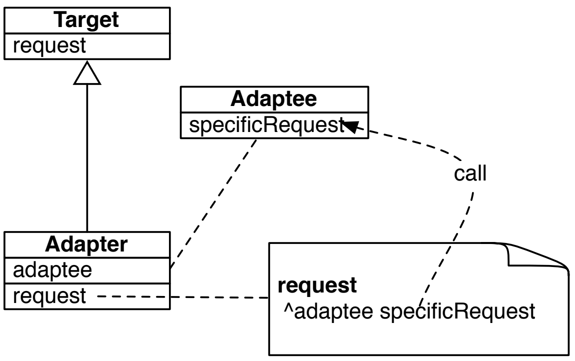


Figure 5.4: Class diagram of the Adapter design pattern

Summary The Adapter design pattern provides a reusable solution to the problem which arises when one wishes to reuse an existing class with an interface that does not match the needed one. The different participants of this design pattern are shown in Figure 5.4. The class we wish to reuse is called the `Adaptee`. Rather than using this class with an incompatible interface directly in the implementation of the software, the `Adapter` provides a wrapper around the `Adaptee`. The `Adapter` will implement the interface that is expected by the software system (in the figure the method `request`). The implementation of this interface will then delegate to the `Adaptee` by invoking the correct method (in the figure the method `specificRequest`) of the `Adaptee`.

Regularities For the Adapter design pattern, we discuss two structural source-code regularities:

1. (inheritance) The Adapter must implement the appropriate interface which is expected by the application;

2. (requirement) The Adapter must invoke operations on the Adaptee in order to carry out a request.

Instantiation

In order to demonstrate the above regularities in practice, we consider an instantiation of the Adapter design pattern from within the implementation of IntensiVE. The `Extension` class, which is part of the framework we implemented for constructing an intension of an intensional view using Smalltalk as a query language, serves as an Adapter for the object representing a set of tuples. As we mentioned in Chapter 4, this `Extension` class offers an interface which is similar to the one of the `Collection` class. On top of these standard `Collection` operations, it also implements a number of set-theoretic operations such as union, intersection, projection, and so on which are not part of the interface of the Smalltalk collection library. As such, the `Extension` class provides a wrapper around the set of tuples, implementing the standard collection operations as well as the set theoretic operations.

Documented regularities

The concretization of regularity (1) in the instantiation of the Adapter pattern we discussed above expresses that the `Extension` class should provide an interface which is compatible with that of the `Collection` class from the standard Smalltalk collection library. We implement this regularity by constructing an intensional view *Adapter* with an attribute ‘method’. The parent view of the *Adapter* intensional view is the view *IntensiVE implementation*, which groups all classes and methods in the implementation of IntensiVE. For the *Adapter* view we specify two alternative views with as intensions:

1.
   ```
   equals(?class,[Extension]),
   methodInProtocol(?method,[#'collection compatibillity'])
   ```

2.
   ```
   methodWithNameInClass(?method,?selector,[Extension]),
   classUnderstands([Collection],?selector)
   ```

The first intension selects all the methods implemented by the `Extension` class which are classified in the protocol ‘collection compatibility’. The second intension selects all methods implemented by `Extension` whose selector is also understood by the `Collection` class. Due to the property of extensional consistency, both these intensions should yield the same set of tuples. As such, any methods of the collection interface which are not implemented by the `Extension` class, or which are not correctly classified in the 'collection compatibility' are reported as discrepancies to the user of our tool suite.

The second regularity of the Adapter pattern manifests itself in our instantiation in that all methods of the `Extension` class which implement a part of the interface of the `Collection` class, must forward a call to the wrapped set of tuples. This implementation convention is documented using a unary intensional constraint:

$\forall$ *adapter* $\in$ *Adapters* :

```
statementsOfMethod(statements( <
  return(send(
    send(variable(self), tuples, <> ),
     ?message,
     ?args)) > ),?adapter.method)
```

This unary constraint verifies whether all methods of the Adapter consist of a single statement which returns the value of the delegation of a message to the variable `tuples`. It is this instance variable `tuples` of the `Extension` class which contains a reference to the wrapped set of tuples.

5.3.5 Chain of Responsibility

Design pattern

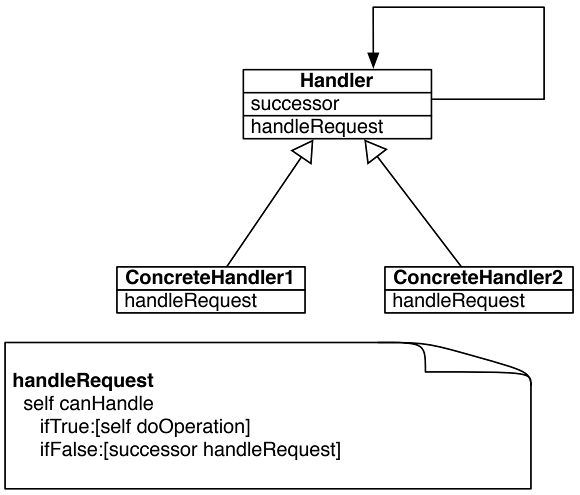


Figure 5.5: Class diagram of the Chain of Responsibility design pattern

Summary The Chain of Responsibility (CoR) design pattern provides a solution template for the problem of a group of objects which can all handle a certain kind of request, but for which it is not a priori known which object will handle it precisely. The class diagram of the CoR pattern is shown in Figure 5.5. The design pattern consists of an abstract class `Handler` with an instance variable `successor` and a number of concrete handlers which can handle the request. To this end, these concrete handlers implement a method `handleRequest`. In order to handle a specific request, a chain of handler objects is created. The first handler in the chain receives the request and, if possible, handles the request. If the first handler cannot handle it, it will forward it to the next handler (which is stored in the `successor` instance variable) and so on, until the end of the chain is reached or a handler has processed the request. A code excerpt implementing this mechanism is shown in Figure 5.5.

Regularities For the Chain of Responsibility design pattern, we consider the following three structural source-code regularities:

1. (inheritance) All handlers must override the method which handles the request;
2. (requirement) All methods which handle a concrete request must also include code which forwards the request to the next handler in the chain. Only these methods may refer to the next handler in the chain;
3. (usage) Clients of the handlers must access them via the handler chain; Direct access to any of the concrete handlers is prohibited.

Instantiation

As an instantiation of the CoR design pattern, we consider an experimental sub-tool of IntensiVE which, given a set of intensional views, automatically searches possible relations that hold between the intensional views (this tool was not included in our discussion of IntensiVE in Chapter 4). Due to the large number of relations which can be proposed by this tool, we implemented a filtering mechanism which performs some post-pruning on the results reported by this tool. We constructed a number of filters which take as input a binary intensional relation and verify whether or not it is an interesting result. These filters are implemented by means of the CoR pattern: if according to a filter a certain relation is obsolete, the filter will prune it. If not, the filter will pass the relation to the next filter in the chain, and so on. And the end of the chain, there is a special filter which collects all non-pruned results.

Documented regularities

We document regularity (1) by creating an intensional view *Handlers* with as parent view *IntensiVE implementation*, as attributes ‘class’ and ‘method’ and two alternative intensions:

1.
   ```
   classInHierarchyOf(?class,[AbstractFilter]),
   methodInClass(?method,?class)
   ```

2.
   ```
   methodInClass(?method,?class),
   methodWithNameInClass(?,[#filterElement:],?class)
   ```

The first alternative selects all the classes implemented in the hierarchy of the `AbstractFilter` class (this is the abstract Handler in the instantiation of the CoR), together with their methods. The second alternative captures all classes which implement a method named `filterElement:`, together with the methods implemented by those classes. As such, these two alternatives express the constraint that all classes in the hierarchy of `AbstractFilter` must implement a method `filterElement:`. Conversely, all classes which implement `filterElement:` but which are not part of the `AbstractFilter` hierarchy will also be reported as discrepancies between the intensional view and the implementation. While it might seem that this constraint is more strict than regularity (1) we discussed above, it is based on the observation that, if a class implements a method `filterElement:` but is not part of the `AbstractFilter` hierarchy, it is likely that this is an error in the source code.

In order for us to codify regularity (2), we create an intensional view *Handling methods* with as attribute 'method' and as parent view the *Handlers* view. For this intensional view we create two alternative views:

1. `methodWithNameInClass(?method,[#filterElement:],?class)`
2. `methodSendsSelector(?method,[#next])`

The first alternative will capture all the methods with as name `filterElement:`; the second alternative yields all methods which send the selector `next`. Since both intensions must result in the same set of tuples, checking extensional consistency of these alternative views will report on all handling methods which do not call the successor in the chain of handlers as well as all methods which refer to the next handler in the chain, but which are not a handling method and which should thus not be allowed to refer to this next handler.

Finally, we implement regularity (3), which states that no individual handler should be accessed outside the chain, by means of a unary intensional constraint:

$\nexists\, handler \in \mathit{Handlers}:$
```
method(?method),
not(Handlers(?,?method)),
methodReferencesClass(?method, ?handler.class)
```

This constraint verifies that for none of the handlers, there exists a method in the system (outside of the methods belonging to the *Handlers* view itself) which references a handler. We allow methods in the implementation of the Handlers to access the handlers directly, since some of these methods perform the construction of the handler-chain.

5.3.6 Observer

Design pattern

Summary The Observer design pattern is applicable in situations in which a number of entities depend on the state of a particular object. Each time the state of the object changes,

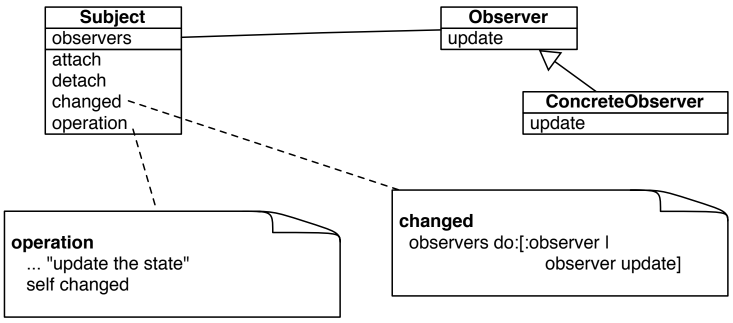


Figure 5.6: Class diagram of the Observer design pattern

all dependent entities must be notified of this change. The class diagram of this design pattern is shown in Figure 5.6. The object which other objects depend on is called the `Subject`. This subject implements a number of facilities in order to manage objects which observe its state: the `attach` and `detach` methods can be used to respectively register or unregister an object as an Observer of the Subject. When the state of the subject changes, the method in the subject responsible for the change invokes the `changed` method on the subject (an example of such a method `operation` is shown in the figure). This `changed` method (whose prototypical implementation is also shown in Figure 5.6) will then notify all the Observers by invoking an `update` method on them.

Regularities While there exist a number of regularities such as e.g. naming conventions which underly the implementation of an instantiation of the Observer design pattern, most of these regularities can be documented in a similar way as the regularities we discussed above. We therefore focus on another structural source-code regularity which illustrates an interesting use of our approach and tool suite. In particular, we are going to document the regularity that all state changing methods must notify that the state of the subject has changed.

Instantiation

The particular instantiation of the Observer design pattern which we will document is taken from the implementation of IntensiVE. Throughout the implementation of our tool suite, we use the Observer design pattern to notify the user interface components of changes in the underlying intensional views and constraints that are shown in the interface.

The implementation of the Observer pattern in IntensiVE makes use of the built-in model-view-controller framework of Smalltalk: this framework adds capabilities for adding and removing Observers to any Smalltalk object. It also provides a set of methods (`changed`, `changed: ...`) which, when invoked, inform the Observers of the object of a state change. As such, whenever a method of the implementation of IntensiVE changes the state of an object, it must invoke one of `changed`,`changed:`,... methods in order to notify the object of the change. Since we use the MVC framework of Smalltalk, the observers of the object will then be notified.

Documented regularities

We document the above regularity by creating two intensional views: an intensional view *Change notification* which groups all methods that perform the notification to the Observers and a view named *State changes* that captures all methods that change the state of an object in the implementation of the model of intensional views. The *Change notification* view has a single attribute 'method' and one alternative which is defined as:

```
methodInClass(?method,[Object]),
methodInProtocol(?method,[#'changing'])
```

This intension (written down in SOUL) captures all methods of the class `Object` which are classified in the `changing` protocol. These are all the methods like e.g. `changed` implementing the notification of the Observers in Smalltalk's MVC framework.

The *State changes* intensional view has as attributes 'class' and 'method' and consists of a single alternative view. This view has the *IntensiVE implementation* view as its parent view. In order to define the intension of this alternative, we use the Smalltalk language. Due to the large number of state changing methods, using Smalltalk as the language for specifying the intension instead of SOUL allows us to more efficiently[4] retrieve the set of tuples belonging to the *State changes* view.

```
1  extension := Intensional.Extension new.
2  mutators := Views.Mutators for:#method.
3  (Store.Registry allPackages select:[:package|
4   package name = 'Intensional Views Model']) first definedClasses
5    do:[:class |
6      class selectorsAndMethodsDo:[:selector :method|
7          mutators do:[:mutator |
8            (method
9             sendsSelector:((mutator valueFor: #method)
10              selector))
11               ifTrue:[extension add:
12                 (Tuple new attribute:#class  value:class;
13                            attribute:#method value:
14                               (SmalltalkMethod
15                                 compiledMethod:method)]]]].
```

Without going into detail, we briefly discuss the workings of this intension. The intension selects all mutator methods (line 2) in the implementation of IntensiVE by referring to the *Mutators* intensional view. It then iterates over all classes in the 'Intensional Views Model' package (lines 3–4), which contains the implementation of the intensional views model. Next, the intension iterates over all these classes and their methods (lines 5 and 6) and verifies for each method implemented on class `class`, whether it invokes any of the mutator methods (lines 8–10). If a mutator method is called, a tuple representing the method which calls the mutator is added to the extension (lines 11–15). Note that this intension in Smalltalk corresponds to the following SOUL query:

[4]On an Apple Mac Mini with an Intel Core Duo 1.66Ghz processor and 1Gb of RAM , the Smalltalk equivalent ran for 5 seconds as opposed to the SOUL query which took 20 seconds.

```
Mutators(?mutator),
packageWithName(?package,['Intensional Views Model]),
classInPackage(?class,?package),
methodInClass(?method,?class),
methodCallsMethod(?method,?mutator)
```

While this SOUL equivalent of the intension is much more succinct and easier to comprehend, we opted for the Smalltalk version due to its better efficiency. Notice that in order to capture the state changing methods, we trusted that only methods using a mutator method alter the state of an object. While this is not guaranteed by Smalltalk itself (a method can directly access the instance variable in its class), we are allowed to make this assumption. We documented a separate structural source-code regularity expressing the explicit obligation that all accesses to instance variables must be performed via a mutator method. As such, we enforce that all direct accesses to an instance variable can only occur from within a mutator method.

In order to document the regularity that all state changing methods must invoke a method that notifies the observers, all we need to do is implement the following binary intensional relation:

$\forall\, statechange \in$ *State changes* :
  $\exists\, changing \in$ *Change notification* :
   `methodCallsMethod(?statechange.method, ?changing.method)`

This relation verifies that all methods which perform a state change contain an invocation of a method which notifies the observer of this change.

5.3.7 Template Method

Design pattern

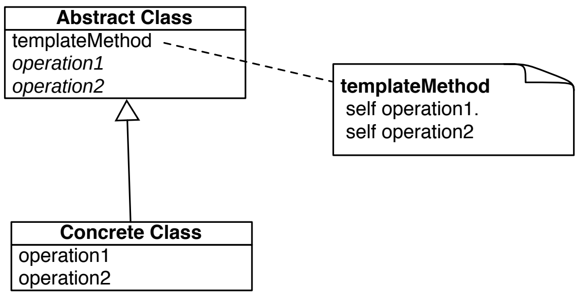


Figure 5.7: Class diagram of the Template Method design pattern

Summary The Template Method design pattern is illustrated in Figure 5.7. This design pattern enables the implementation of an algorithm while deferring some of the algorithm's steps to a subclass. This concept is demonstrated in the figure. The algorithm is implemented by the method `templateMethod` in the `AbstractClass`. In the implementation of the algorithm, two operations `operation1` and `operation2` are invoked which are left abstract in `AbstractClass`. A concrete algorithm (represented by the class `ConcreteClass`) implements `operation1` and `operation2`. As such, the invariant part of the algorithm is implemented by the `AbstractClass` while each `ConcreteClass` implements the variable parts of the algorithm.

Regularities The Template design pattern imposes an interesting kind of structural source-code regularity that we did not yet encounter in any of the previous examples. Namely, in order for the algorithm to be functioning properly, each subclass of `AbstractClass` must implement the correct interface containing the abstract methods which get called from within the `templateMethod`. If this inheritance regularity is violated, the implementation of the algorithms might function improperly.

Instantiation

The instantiation of the Template Method design pattern we use as a guideline to document the above regularity is taken from the implementation of IntensiVE. In IntensiVE, the evaluation of intensions is delegated to a dedicated class. For each query language that is supported by our tool suite, there exists a separate subclass of `AbstractEvaluator` which evaluates the intension. This class also implements a method `verifyIntension:` that verifies whether an intension is correct and, if not, throws an exception. This `verifyIntension:` method is a template method which delegates the verification of both the syntactical correctness of an intension, as well as whether the intension uses the correct attributes of an intension to two methods, namely `correctIntension:` and `correctAttributes:` which are implemented by the concrete evaluators.

Documented regularities

We can document this structural source-code regularity quite elegantly by creating an intensional view *TemplateMethods* with as parent view *IntensiVE implementation*, as attribute ‘class’ and two alternative views with an intension:

1. `classBelow(?class, [AbstractEvaluator])`

2.
   ```
   methodWithNameInClass(?,[#'correctIntension:'],?class),
   methodWithNameInClass(?,[#'correctAttributes:'],?class)
   ```

When evaluating the first intension, this will result in all the classes which are (directly or indirectly) a subclass of the `AbstractEvaluator` class. The result of the second intension is the set of all classes which implement both the method `correctIntension:` as the method `correctAttributes:`. For this intensional view to be extensionally consistent, all the concrete evaluators must implement the `correctIntension:` and `correctAttributes:`. Moreover, a user will also be informed about any classes which do implement these two methods, but which are not a descendant of `AbstractEvaluator`.

This second constraint – which the dual version of the documented regularity – results in that our alternative views express a stronger constraint than the one we described above. However, if a class implements the `correctIntension:` and `correctAttributes:`, but is not part of the `AbstractEvaluator` class hierarchy, this is an indicator of a possible error. That is why in this case, we opted to implement the above constraints by means of alternative views.

5.3.8 Proxy

Design pattern

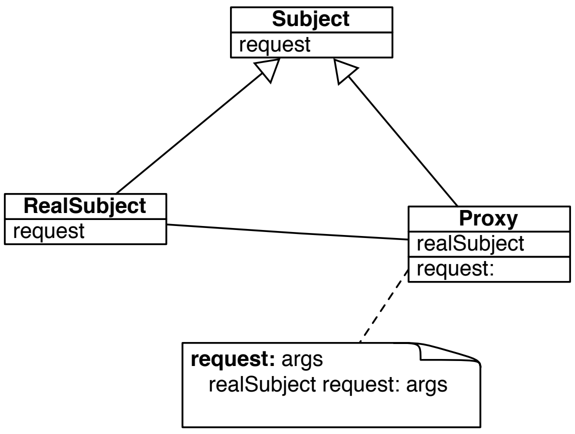


Figure 5.8: Class diagram of the Proxy design pattern

Summary The Proxy design pattern is used when implementing a class which serves as a placeholder for another class. The class diagram of this pattern is shown in Figure 5.8. The `Proxy` class which serves as a placeholder for another class is implemented in the same hierarchy (i.e. `Subject`) of the element it replaces (the `RealSubject` class). The `Proxy` implements the same interface as the subject. However, instead of handling the request itself, the proxy forwards it to the `RealSubject`. This design pattern closely resembles the Adapter pattern which we discussed in Section 5.3.4. The main difference between the Adapter and the Proxy pattern is that the latter pattern is used when the interface of the wrapper is identical to the interface of the wrappee.

Regularities One structural source-code regularity underlying an instantiation of the Proxy design pattern we will discuss here is the idiomatic implementation of the handling of requests on the Proxy class. When a Proxy receives a request which is also implemented by the Subject it is wrapping, it should forward this request to the actual subject.

Instantiation

The instantiation of the Proxy design pattern we document here is taken from the implementation of the *Trippy* object inspector of VisualWorks Smalltalk. In the implementation of Trippy, a Proxy design pattern is used to make it possible to preview any object that represents a visual component from the interface framework of VisualWorks. The `VisualComponentProxy` pretends to be such a `VisualComponent` in order for Trippy to visualize it without having to take the actual visual component which is wrapped by the proxy out of its proper hierarchy. In the implementation, `VisualComponentProxy` thus forwards any requests to the instance of `VisualComponent` which it serves as proxy for.

Documented regularities

We document the above regularity by creating an intensional view *ProxiedMethods* with as attribute 'method' and a single alternate view with as intension:

```
methodWithNameInClass(?method,?selector,[VisualComponentProxy]),
methodWithNameInClass(?,?selector,[VisualComponent])
```

This intension captures all the methods implemented by the `VisualComponentProxy` class for which there exists a method with the same selector in the `VisualComponent` class. It thus returns a set of tuples representing all methods of the proxy which implement the interface of the subject.

In order for the proxy to be implemented correctly, all the methods belonging to the *ProxiedMethods* intensional view should share an idiomatic implementation which forwards the incoming request to the wrapped subject. This is verified by the following unary intensional constraint imposed on the *ProxiedMethods* intensional view:

$\forall\, proxy \in$ *ProxiedMethods* :

```
  methodWithName(?proxy.method, ?message),
  argumentsOfMethod(?arguments, ?proxy.method),
  methodWithSend(?proxy.method, variable(actualComponent),
                 ?message, ?arguments)
```

The above constraint holds if for all methods of the proxy that implement a part of the `VisualComponent` interface, it is true that they forward the requests they receive to the wrapped subject using the same arguments (in case of the Trippy example this subject is stored in the instance variable `actualComponent`). If the `VisualComponentProxy` thus implements a proxied method that does not contain such a call, upon verification of the above constraint, this method will be reported as a possible inconsistency.

5.3.9 Builder

Design pattern

Summary The Builder design pattern is applicable whenever a developer strives to decouple the construction of a complex object structure from its actual representation. This

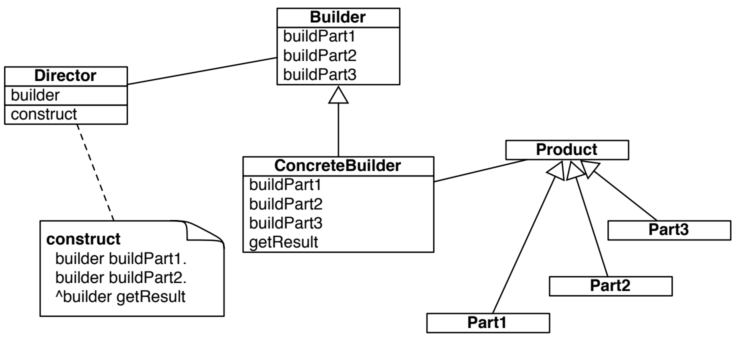


Figure 5.9: Class diagram of the Builder design pattern

makes it possible to reuse the same construction process for a different representation of the object structure. The workings of this design pattern are shown in Figure 5.9. The `Builder` hierarchy implements the infrastructure for the construction of the object structure. A builder contains a method for each of the different products it can manufacture, along with a method (`getResult` in the figure) for returning the entire object structure. An object that guides the construction process (called the `Director`) uses an instance of the builder to construct the actual object structure. For instance, in the figure, the director contains a method `construct` that invokes a number of calls to the builder in order to construct the object. In order to return the constructed object, the `construct` method invokes the `getResult` method of the builder.

Regularities We discuss two structural source-code regularities for the Builder design pattern:

1. (stylistic) All methods responsible for constructing products must follow a similar naming scheme;
2. (requirement) The director should not reference any of the products directly.

Instantiation

The parser infrastructure of VisualWorks Smalltalk uses the Builder design pattern to construct the abstract syntax trees representing a Smalltalk program. The different classes in the hierarchy of the `Parser` class are the directors which make use of a concrete builder. For the Smalltalk parser, this concrete builder is the `ProgramNodeBuilder` class which constructs objects in the hierarchy of `ProgramNode`.

Documented regularities

To create verifiable documentation for the two regularities above, we start by creating an intensional view called `BuilderProducts` which contains all the products created by the

Builder. This intensional view with as attribute 'class' has one alternative view with as intension:

```
classInHierarchyOf(?class,[ProgramNode])
```

As such, it contains all abstract syntax tree nodes, represented by all classes in the hierarchy of `ProgramNode`. Next, we create an intensional view *Production methods* which groups all methods of the concrete builders that construct an abstract syntax node. This intensional view has as attributes 'class' and 'method' and a single alternative view with as intension:

```
1 classInHierarchyOf(?class,[ProgramNodeBuilder]),
2 methodInClass(?method,?class),
3 BuilderProducts(?product),
4 methodWithSend(?method,variable([?product name asSymbol]),new,?)
```

Lines 1 and 2 of the above intension select all classes and methods in the `ProgramNodeBuilder` hierarchy. Line 3 retrieves all products by referring to the *BuilderProducts* intensional view. Finally, line 4 restricts the set of methods from the `ProgramNodeBuilder` hierarchy to those who send a message `new` to a product.

Using the *Production methods* intensional view and a unary intensional constraint we can express regularity (1) as follows:

$\forall\, production \in \textit{Production methods}$:
    `['new*' match: ?production.method asString]`

Upon verification, this unary constraint will return true or false depending on whether all the methods that construct a product following the same naming scheme, i.e. the method name starts with the prefix "new-". We implemented this regularity using a unary constraint rather than by creating an additional alternative view for the *Production methods* intensional view since the constraint that all these methods should start with "new-" is a necessary condition but not a sufficient one: while it must hold that all production methods start with "new-" the opposite is not necessarily true.

We document the second regularity by creating an intensional view *BuilderDirectors* with as attribute 'class' and one alternative view:

```
classInHierarchyOf(?class,[Parser])
```

This intensional view thus states that all classes in the hierarchy of `Parser` are a director in the instantiation of the Builder design pattern. Using this intensional view, we express regularity (2) by the following binary intensional relation:

$\forall\, director \in \textit{BuilderDirectors}$:
  $\nexists\, product \in \textit{Builderproducts}$:
    `methodReferencesClass(?director.method, ?product.class)`

This relation verifies that none of the directors directly reference a builder product class.

5.4 Discussion

Kinds of regularities encountered

The goal of the above experiment was to demonstrate how a wide range of structural source-code regularities can be documented using our model of intensional views and constraints. During this experiment, we studied various specific regularities that are present in instantiations of nine different design patterns and explained how, using our methodology and the IntensiVE tool suite, we created verifiable documentation for these regularities. This documentation entails the following kinds of structural source-code regularities:

- We demonstrated a number of *stylistic* constraints which describe the naming scheme that the different source-code entities making up the participants of a design pattern should follow. As we have shown, such constraints cover a wide range of structural source-code regularities from specifying how different classes or methods should be named, over how the arguments of a method should be named, to even more complex naming schemes. An example of such a more complex naming scheme can be found in the Factory design pattern: we documented that each factory method must consist of the prefix "make-" along with the name of the actual entity it produces;

- Another kind of regularity we encountered in this experiment was the *idiomatic* implementation of some of the participants of a design pattern. These regularities are characterized by that they describe a certain implementation pattern for a participant of a design pattern. We documented these regularities by encapsulating this implementation pattern using multiple alternative views or unary intensional constraints. For instance, one such regularity we encountered is that the accept methods of a Visitor design pattern need to be implemented using a double-dispatch protocol in order to function correctly. We documented this regularity by means of a unary intensional constraint on the *Accept methods* intensional view. Another example of such an idiomatic implementation stems from the instantiation of the Adapter design pattern. In this pattern, we require that all methods implemented by an Adapter follow a similar implementation pattern, namely that in their implementation they perform a delegation to the adaptee;

- A number of the regularities we documented involve the correct *usage* of the design pattern. These regularities prescribe how the different entities belonging to an instantiation of a design pattern can be used by client code. For instance, for the Façade pattern, we documented that a client of the façade is not allowed to directly access the subsystems for which the façade provides an interface. Other examples of such usage regularities that we have encountered in the above experiment are that in the instantiation of the Builder pattern the director is not allowed to create products directly or in the Chain of Responsibility pattern, where external elements are not allowed to directly use any of the elements of the chain;

- We also encountered a number of *inheritance* regularities. These kinds of regularities describe how the different entities in a class hierarchy are structured. One example we

```
Parser>>parseArgsAndTemps: aString notifying: req
 "(for explainer) parse the string and answer
    with an Array of Strings (the arg and temp names)"
  ^self
    initPattern: aString
    notifying: req
    saveComments: false
    return: [:pattern |
      "skip primitive if any"
      self readStandardPragmas: MethodNode new temps: #().
      ((pattern at: 2) , self temporaries)
            collect: [:param | param name asString]]
```

Figure 5.10: The implementation of the `parseArgsAndTemps:notifying:` method in the `Parser` class.

encountered of such an inheritance regularity is that in the instantiation of the Template method design pattern all subclasses of `AbstractEvaluator` must override a method named `verifyIntension:` in order;

- The last kind of structural source-code regularities we encountered are *design requirement* regularities. These regularities express the dependencies between different concepts from the design of a system. For example, in the Visitor design pattern we encountered the design regularity that all accept methods must contain an invocation of a visit method. Another example of such a regularity is that, when implementing an Observer design pattern, each state change should trigger an update of the observers.

Verification of the regularities

While the emphasis of this chapter lies with the documentation of structural source-code regularities, the actual documentation we created for each of the instantiations of the design patterns is verifiable with respect to the implementation of that instantiation. Since the intensional views and constraints we presented in the experiment are a translation of how we *assumed* the different regularities manifest themselves in the implementation of the design patterns, we – in line with our methodology – iteratively refined this initial documentation with respect to the source code. Based on the feedback provided by our tool suite, we then refined the documentation and/or the source code until both artifacts were synchronized.

This iterative refinement of the regularities and the source code has led to the following two observations:

- In some cases, inconsistencies between the documented regularities and the actual source code revealed a number of errors in the implementation of the design pattern. For instance, when verifying the validity of the documentation we created for the instantiation of both the Visitor as well as the Factory design patterns in SOUL, the tool suite reported a number of source-code entities that erroneously deviated from the documented naming conventions. Similarly, verifying our documentation of the Observer design pattern in the implementation of IntensiVE itself identified a couple of locations in the source code which perform a state change but for which the corresponding notification of this state change was – incorrectly – omitted.

- Upon verification of the intensional views and constraints, we also encountered a number of discrepancies which did not result from bugs in the source code, but rather illustrated that in some cases our initial documentation of a regularity did not entirely match with how this regularity was manifested in the source code. For instance, for the instantiation of the Builder design pattern, we imposed the restriction that none of the directors should directly refer to a concrete product. However, when verifying the consistency of this constraint with respect to the source code, we were reported on a single method which did not respect this regularity. The implementation of this method can be found in Figure 5.10. This method, belonging to the default Smalltalk parser, provides a means for the *Explainer* tool to extract the variable names (arguments and temporaries) from a string representing a Smalltalk program. In order to skip any primitives in this string, the `readStandardPragmas:temps:` method is invoked, which expects an instance of `MethodNode` as its first argument. The creation of this instance is not related to the Builder design pattern, and is thus not a violation of the documented regularity. As such, we document this method as an explicit exception to the regularity.

Validity of the regularities

Note that the documentation of the structural source-code regularities does not provide any guarantees that the documented instantiation of the design pattern behaves properly. In our documentation we often provide a conservative approximation of the envisioned regularity. E.g. in the Visitor design pattern we require all accept methods to invoke a visit method. However, since we only analyze the source-code of the system we cannot make any claims whether at run-time, an accept method will actually invoke a visit method. Instead, we approximate this information by requiring that the accept method's body contains a call to a visit method. As such, it is possible that, although our tool suite reports no discrepancies between the documentation and the implementation, at run-time the accept method's invocation of the visit method never gets executed. However, if our tool suite reports that an accept method does not contain a call to a visit method, this can be an indicator that the instantiation of the Visitor design pattern is not implemented correctly.

Applicability of different kinds of constraints

Although we already discussed the applicability of the different kinds of constraints supported by our tool suite and approach earlier, this experiment provides a nice illustration of their use:

- **Alternative views**: provide a means to give *multiple, equivalent descriptions* of a *single intensional view*. For instance, in the instantiation of the Adapter design pattern, we have used alternative views to express that the set of methods in the protocol 'collection compatibility' must be identical to the set of methods that implement a message also understood by the `Collection` class. Due to the property of extensional consistency, which requires that all alternatives of the same intensional view yield the same extension, using alternative views as a constraint over intensional views allows us to detect all source-code entities which belong to at least one of the alternative views, but not necessarily to all of the alternatives. For instance, suppose that we have a concern described by three alternative views, each expressing a different regularity governing the concern. If a source-code entity respects one of these regularities, but not the other two, then it will be flagged as a possible inconsistency by our tool suite. Only in the case that a source-code entity does *not respect any of the regularities* documented by alternative views will it be missed by our approach;

- **Unary intensional constraints**: document a regularity which is applicable to a single intensional view. For instance, we have documented the naming convention that all methods of a Builder that construct a Product should begin with the prefix "new-" by means of a unary constraint. As we have discussed earlier, the difference between a unary constraint and the use of alternative views is that the former kind expresses a condition that is necessary while the latter expresses a necessary and sufficient condition;

- **Binary intensional relations**: are used to express a regularity that involves two intensional views. For instance, the requirement that no director in the instantiation of a Builder pattern should be allowed to directly instantiate a Product is expressed with such a relation.

Reusability of the documentation

It was not our goal to create a reusable set of source-code regularities which is generally applicable whenever a design pattern is instantiated. Instead, we focussed on demonstrating – using design patterns as a case study – how our approach can be applied to document different kinds of regularities that govern the implementation of one specific instantiation of a design pattern.

However, this does not mean that the documentation we created is limited to the instantiation of the design pattern which we applied it to. First of all, the regularities we discussed are often not limited to a single instantiation of the design pattern, but express a constraint which is in general applicable to instantiations of the same design pattern. The usage constraints, design conventions and implementation idioms such as for instance the regularity that all factory methods must return an instance of a product, a proxy must forward calls to the wrapped object and so on, describe invariants which must be upheld by any instantiation of the design pattern for the instantiation to be able to behave properly. While the stylistic constraints we have defined are often specific to a single instantiation of a design pattern, similar conventions often govern other instantiations of that same pattern.

Second, also the actual documentation we created for the structural source-code regularities is to some extent reusable. While the intensional views and constraints over the views which we defined during our experiment document the regularities underlying a single instantiation of a design pattern, they can be considered to be a sort of implementation template for documenting similar regularities underlying other instantiations of the same design pattern.

Although it lies outside the scope of this dissertation, we envision a more reusable means to document the different regularities underlying design patterns using some sort of "template regularities". Such templates consist of a number of parameterized intensional views and constraints that express the different regularities governing a certain design pattern. By instantiating these template regularities, i.e. by specifying which source-code entities are associated with the roles of the design pattern, the set of intensional views and regularities can document a specific instantiation of the design pattern and verify whether this instantiation respects the different structural source-code regularities that govern that design pattern.

5.5 Conclusions

In this chapter we presented an experiment in which we demonstrated how our model of intensional views and constraints and the associated tool suite can be used to document various structural source-code regularities. As a case study, we analyzed the different structural-source code regularities underlying the implementation of object-oriented design patterns and demonstrated how we can translate these regularities, applied to a specific instantiation of the design pattern, into intensional views and constraints over these intensional views.

The goal of this chapter was to provide an initial assessment of whether our approach provides a sufficiently generic and expressive means for documenting structural source-code regularities. In the next chapter, we complement this validation by studying how our model of intensional views and constraints, along with our iterative and test-based methodology can aid in maintaining the causal link between documented regularities and source code when a system evolves.

Chapter 6

Supporting co-design and co-evolution using intensional views and constraints

In this chapter we report on three case studies which serve as a means to assess how our approach and associated methodology support the maintenance of the causal link between the documentation of structural source-code regularities and the source code of a system:

- In Section 6.1 we discuss an experiment in which we documented the structural source-code regularities underlying the implementation of the IntensiVE tool suite. Furthermore, we illustrate how we incorporated the verification of these regularities in the development process of IntensiVE and analyze how our approach and methodology aided in both detecting and resolving evolution conflicts;

- Section 6.2 gives an account of the second experiment we conducted, in which we documented the set of regularities governing the implementation of SmallWiki, a collaborative wiki system implemented in VisualWorks Smalltalk. We started this experiment by encoding the regularities underlying an initial version of SmallWiki. By applying this documentation to subsequent versions of SmallWiki, we studied how the evolution of the system impacted the intensional views and constraints we defined and how, by applying our methodology, we were able to synchronize the documentation and the source code;

- In the third case study we documented DelfSTof (Section 6.3). DelfSTof is an object-oriented framework for performing experiments on source code using formal concept analysis. For this framework, we documented some of the regularities governing the correct instantiation of this framework and applied it to a concrete instantiation of the framework.

6.1 IntensiVE

6.1.1 Overview

In Chapter 4 we introduced IntensiVE, the tool suite we implemented in the context of the work we present in this dissertation. In this first experiment, we use the implementation of IntensiVE as a case study to demonstrate how our approach can support the co-design and co-evolution of documented structural source-code regularities with respect to the implementation. IntensiVE is a modest-sized application that consists of 148 classes and 2118 methods.

We documented a number of regularities underlying the implementation of IntensiVE, using the tool suite itself. We created a total of 41 intensional views which group source-code entities from the implementation of IntensiVE. For 13 of these 41 intensional views, more than one alternative view was specified. Moreover, we imposed 23 constraints over these intensional views: 15 unary intensional constraints and 8 binary intensional relations.

An overview of the intensional views we declared on the source code of IntensiVE and the constraints over these views can be found in Figures 6.1 and 6.2. In these figures, each intensional view is represented as a rounded rectangle; when there are multiple alternative views for one intensional view, this is indicated by stacked rectangles. For instance, for the intensional view *Intension Evaluators* (displayed in Figure 6.1), we defined three alternative views. The parent view of a certain intensional view is indicated by means of a dotted line with a circle at the end of the parent view. E.g. the intensional view *Extension* has the *Core model* view as its parent. If an intensional view is defined in terms of other views, this is indicated by a dotted line with a square at the end. For instance, in Figure 6.1, we see that the intensional view *Core Model* is defined in terms of the *Views Model* and *Constraints Model* views. Constraints over intensional views are visualized by an arrow.

In what follows, we briefly discuss the different structural regularities we documented for IntensiVE. We will not give a detailed description of the actual definition of all intensional views and constraints, but only discuss those views and constraints that illustrate the detection of an interesting evolution conflict. However, the interested reader can find a complete overview of the definition of all intensional views and constraints on IntensiVE in Appendix B.

Intensional views and constraints over the core model of IntensiVE

The core model of IntensiVE consists of the part of the source code that implements the model of intensional views and constraints. As such, it encompasses all the classes and methods which implement the notions of *intensional views*, *unary constraints*, *quantifiers*, *evaluation of intensions*, and so on. The intensional views and constraints documenting the structural source-code regularities in this part of the implementation of IntensiVE are shown in Figure 6.1. We can distinguish between the following regularities:

- **Domain-related regularities:** a number of the intensional views we defined over the core model directly align with a domain concept from the model of intensional views. For example, we have intensional views that group the source-code artifacts that implement the evaluation of the intension of an intensional view (*Intension evaluators*),

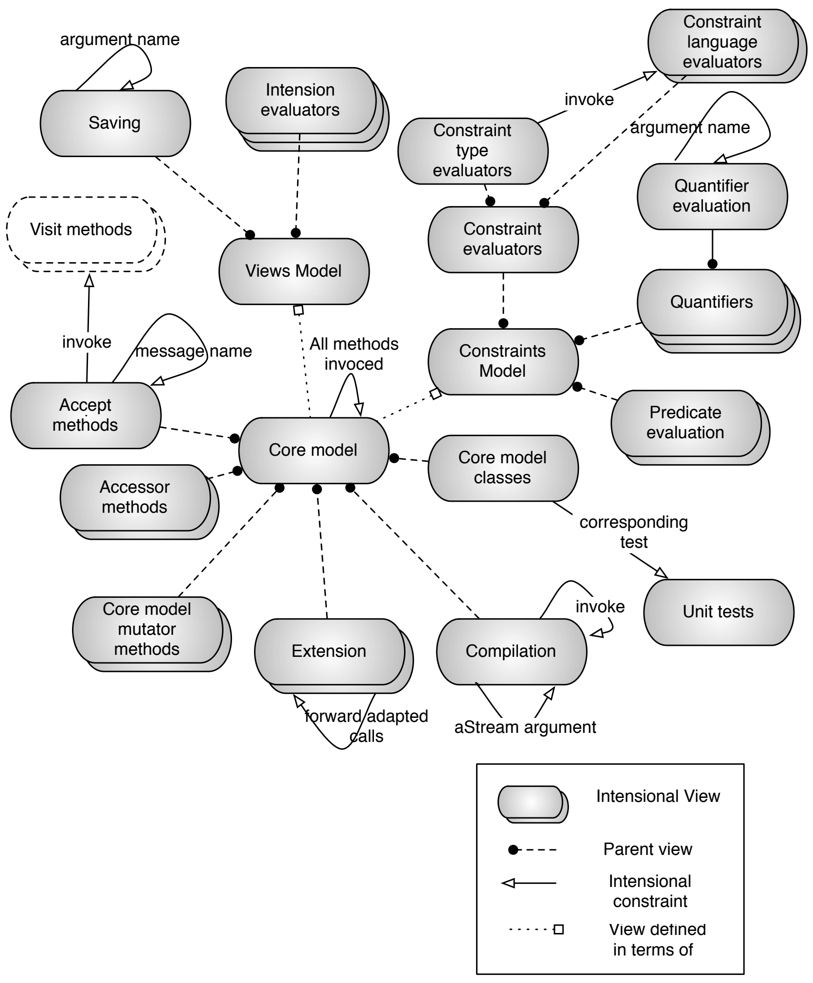


Figure 6.1: Overview of the intensional views and constraints in the IntensiVE core.

the implementation of quantifiers (*Quantifiers* and *Quantifier evaluation*), the saving mechanism of the entities in the core model (*Saving* and *Compilation*), and so on. Note that these intensional views and the constraints imposed on them implement different kinds of regularities:

- A number of stylistic constraints are encoded such as for instance naming conventions for certain class hierarchies, a correct naming scheme for argument names, etc;
- Since IntensiVE is implemented as a framework, some of the regularities prescribe the specialization interface which needs to be implemented for a certain concept (e.g. the different methods a class implementing a quantifier must override).;
- We also documented a number of design constraints that dictate how different

concepts in the framework must interact. For example, we have expressed that all *Constraint type evaluators* must implement the correct invocation pattern for calling a *Constraint language evaluator*. This binary intensional relation regulates how the implementation of the evaluation of constraints must make use of the evaluators for the predicate of a relation.

- **Consistent implementation of idioms:** A number of intensional views document the consistent implementation of certain implementation concepts such as accessor and mutator methods. These views are not limited to expressing whether these idioms are implemented correctly, but also regulate the correct usage of mutators and accessors throughout the implementation of the IntensiVE core model;

- **Documentation of design patterns:** We created some intensional views that document an instantiation of a design pattern in the implementation of IntensiVE. For example, the *Extension* intensional view documents an instantiation of the Adapter design pattern. These intensional views and the constraints imposed on the views document the pattern's structure by expressing the correct naming scheme of the different participants in the design pattern, the idiomatic implementation of some of the participants and dependencies between participants (e.g. all adapted methods must forward a request to the adaptee).

- **Implementation regularities:** In our documentation of regularities underlying the core model of IntensiVE, we expressed the constraint that for each class in the core model, there should exist a corresponding unit test case. While this regularity is not directly related to documenting the implementation of IntensiVE, it documents a regularity of the development process itself.

Other intensional views and constraints over IntensiVE

Next to the intensional views and constraints over the core model of IntensiVE, we also documented a number of structural source-code regularities in the graphical user interface of IntensiVE and the part of the saving mechanism that handles the representation of data as programs. Furthermore, we documented the implementation of the Factory and Observer design patterns in IntensiVE, the implementation of the deduce tool and a number of implementation concepts like private methods which express often-occurring coding conventions. These intensional views and constraints are illustrated in Figure 6.2.

- **Graphical user interface:** We documented a number of regularities underlying the graphical user interface of IntensiVE. In particular, we expressed a number of regularities concerning the implementation of the drag & drop functionality in the editors of the IntensiVE tool suite. Furthermore, we documented an architectural regularity which regulates the correct interaction between the editors and the intensional views/constraints which are being edited. In IntensiVE, the editor is strictly separated from the item that is being edited. As such, the editor only has access to the item that is being edited by means of the mutator methods of the item. A wrapper is placed around

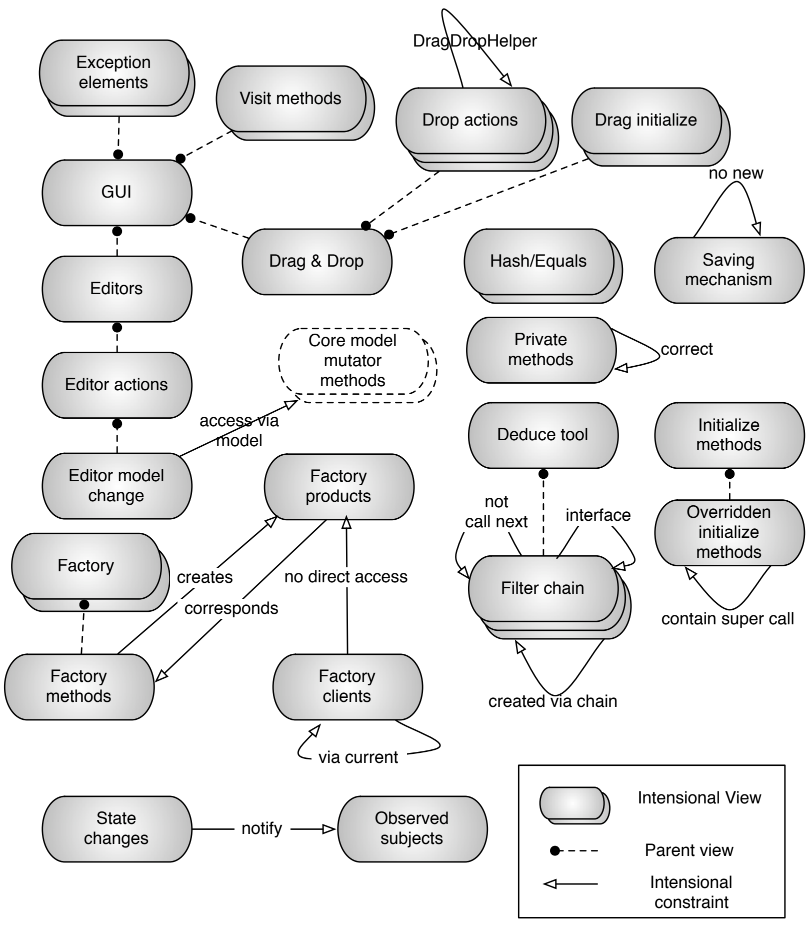


Figure 6.2: Overview of the intensional views and constraints in the user interface and satellite tools of the IntensiVE implementation.

this item in order to provide additional services such as change notification and undo functionality. Consequently, we documented that from within the editor only changes can be made to an item by means of invocations of mutator methods to the wrapped item;

- **Design patterns:** In the implementation of IntensiVE, we used a Factory design pattern to construct instances of entities from the core model as well as the user interface, etc. We documented this instantiation of the Factory design pattern similarly to the Factory design pattern we discussed in Chapter 5. Furthermore, in the GUI part of IntensiVE we used an Observer design pattern in order to notify an editor of changes in the entities which are being edited. This design pattern has also been documented using a set of intensional views and constraints. Similarly, we documented the chain of responsibility from the implementation of the deduce tool, one of the experimental tools we grafted on the implementation of IntensiVE;

- **Implementation regularities:** We also created a number of intensional views on the implementation of IntensiVE for capturing infringements on a number of frequently made errors or coding errors. We created an intensional view specifying that all implementors of the '=' message in IntensiVE should also implement a method `hash`. If not, object comparison in collections might behave incorrectly. Similarly we expressed the regularity that all methods classified in the private protocol can only be invoked from within the class hierarchy in which they are defined[1]. Furthermore, as a precaution that the initialization of a parent class is not lost, we documented a regularity stating that methods which override the `initialize` method must contain a super call.

6.1.2 Supporting co-design and co-evolution

In this section we illustrate how we were able to support co-design and co-evolution of the documented structural source-code regularities with the source code of IntensiVE by applying our approach and methodology. This section is divided into two parts:

- We first discuss a number of examples that demonstrate how the co-design of the documented regularities in IntensiVE and the source code led to the step-wise refinement of the intensional views and constraints as well as to the detection of inconsistencies in the source code of IntensiVE;

- Secondly, we relate our experience of incorporating the verification of the regularities we documented using intensional views and constraints into the development process of IntensiVE. We illustrate how this integration allowed us to detect infringements on documented regularities as early as possible during development. Furthermore, we discuss how changes in the implementation and design of our tool suite impacted the documented regularities and how our methodology aided in synchronizing the documentation and the source code upon evolution.

Co-design of intensional views and source code

The intensional views and constraints we imposed on the implementation of IntensiVE were created during the later phases of the actual development of our tool suite, as soon as the functionality of the tool suite was rich and stable enough to express the intensional views and constraints documenting the tool suite's own regularities. In order to create these views and constraints, we followed the iterative refinement process we described in Chapter 4. As such, we started out by – for each intensional view and constraint – defining an initial assumption about how the structural source-code regularities governing IntensiVE manifest themselves in the implementation.

As prescribed by our methodology, we subsequently verify the validity of this initial documentation with respect to the source code. Based on the inconsistencies reported by our tool suite, we iteratively refined the source code and/or the documentation of the regularities and

[1] In Smalltalk, all methods are public. However, in order to identify methods that should be considered private, we classify them in the `private` protocol

```
AlternativeView>>model
 model isNil
  ifTrue:
   [model := UndoableModel onObject: self.
    self addDependent: model].
^model
```

Figure 6.3: Example of a lazy-initialized accessor method

repeated this verification process until the set of intensional views and constraints was synchronized with the source code. For certain intensional views and constraints, this verification of the initial version of the documentation revealed that there were no discrepancies between the documentation and the source code, meaning that our initial assumption about how a structural regularity is manifested in the source code was correct. Among others, this was the case for the *Accept methods* and *Visit methods* intensional views, and the binary relation that expresses that all accept methods must contain a call to a visit method.

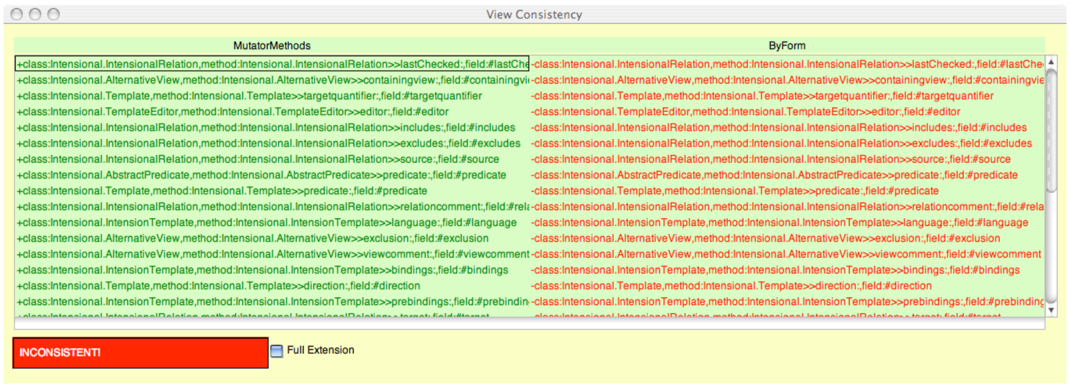
However our initial attempt at specifying documentation for a regularity resulted in a number of discrepancies between the documentation and the implementation for the majority of intensional views and constraints. By inspecting the documentation and discrepancies using our tool suite we identified three reasons for these discrepancies.

Deviations from the documentation A first reason for the inconsistency of the documentation is that the intension of an alternative view or the predicate of a constraint is correct in the general case but there exist some source-code entities for which the documented regularity does not hold. These exceptions to the regularity were then explicitly documented as deviations to the documentation. As such, these deviations demonstrate how the source code of our tool suite had an impact on the documented regularities and how, by updating this documentation, we were able to co-design both artifacts.

For example, we defined the *Accessors* intensional view using the following two alternative views:

```
1.methodWithName(?method,?field),
  instanceVariableInClass(?field,?class)
2.statementsOfMethod(statements(<?statement>),?method),
  instanceVariableInClass(?field,?class),
  equals(?statement, return(variable(?field)))
```

The first alternative expresses the naming convention that the name of an accessor method must correspond to the name of the field that is being accessed; the second alternative states that an accessor method consists of a single statement returning the value of an instance variable. Verifying consistency of this intensional view reported three methods which followed the naming scheme (alternative 1) but which did not adhere to the implementation idiom (alternative 2). A closer inspection of these three methods revealed that they all followed a similar implementation pattern (as illustrated in Figure 6.3). This pattern deviates from the idiomatic implementation we documented by doing a lazy-initialization of the accessed field.



MutatorMethods	ByForm
+class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>lastChecked:,field:#lastChe	-class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>lastChecked:,field:#lastChe
+class:Intensional.AlternativeView,method:Intensional.AlternativeView>>containingview:,field:#containingvi	-class:Intensional.AlternativeView,method:Intensional.AlternativeView>>containingview:,field:#containingvie
+class:Intensional.Template,method:Intensional.Template>>targetquantifier:,field:#targetquantifier	-class:Intensional.Template,method:Intensional.Template>>targetquantifier:,field:#targetquantifier
+class:Intensional.TemplateEditor,method:Intensional.TemplateEditor>>editor:,field:#editor	-class:Intensional.TemplateEditor,method:Intensional.TemplateEditor>>editor:,field:#editor
+class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>includes:,field:#includes	-class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>includes:,field:#includes
+class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>excludes:,field:#excludes	-class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>excludes:,field:#excludes
+class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>source:,field:#source	-class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>source:,field:#source
+class:Intensional.AbstractPredicate,method:Intensional.AbstractPredicate>>predicate:,field:#predicate	-class:Intensional.AbstractPredicate,method:Intensional.AbstractPredicate>>predicate:,field:#predicate
+class:Intensional.Template,method:Intensional.Template>>predicate:,field:#predicate	-class:Intensional.Template,method:Intensional.Template>>predicate:,field:#predicate
+class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>relationcomment:,field:#rel	-class:Intensional.IntensionalRelation,method:Intensional.IntensionalRelation>>relationcomment:,field:#rela
+class:Intensional.IntensionTemplate,method:Intensional.IntensionTemplate>>language:,field:#language	-class:Intensional.IntensionTemplate,method:Intensional.IntensionTemplate>>language:,field:#language
+class:Intensional.AlternativeView,method:Intensional.AlternativeView>>exclusion:,field:#exclusion	-class:Intensional.AlternativeView,method:Intensional.AlternativeView>>exclusion:,field:#exclusion
+class:Intensional.AlternativeView,method:Intensional.AlternativeView>>viewcomment:,field:#viewcomment	-class:Intensional.AlternativeView,method:Intensional.AlternativeView>>viewcomment:,field:#viewcomment
+class:Intensional.IntensionTemplate,method:Intensional.IntensionTemplate>>bindings:,field:#bindings	-class:Intensional.IntensionTemplate,method:Intensional.IntensionTemplate>>bindings:,field:#bindings
+class:Intensional.Template,method:Intensional.Template>>direction:,field:#direction	-class:Intensional.Template,method:Intensional.Template>>direction:,field:#direction
+class:Intensional.IntensionTemplate,method:Intensional.IntensionTemplate>>prebindings:,field:#prebindin	-class:Intensional.IntensionTemplate,method:Intensional.IntensionTemplate>>prebindings:,field:#prebinding

Figure 6.4: The *Extensional Consistency Inspector* opened on the *Mutators* intensional view.

To correct this intensional view, we documented the three methods as explicit deviations from the intension of the second alternative view.

While in the case of the *Accessors* view the deviation was caused by an intension that was too specific, the *Extension* intensional view provides an example of an intension that was too broad. Upon verifying the validity of the *Extension* view, our tool suite pointed out a single discrepancy, namely the method `copy` implemented in the `Extension` class. The *Extension* view captures the regularity that the methods of the `Extension` class form an adapter for the actual underlying abstraction of a set of tuples. As such, the methods belonging to the *Extension* that adapt a method that is part of the interface of `Collection` must forward a request to the wrapped set of tuples. However, since the `Extension` class as well as the `Collection` class implement a method `copy`, this method was also captured by our intensional view as a method that should be adapted. Since this `copy` method was not part of the adapted interface, we explicitly documented it as a deviation to the *Extension* intensional view.

Errors in the documentation A second reason for some of the discrepancies we encountered was that our initial assumption on which we based the intensional view or constraint was erroneous and needed to be altered. Similar to the examples discussed above, these discrepancies illustrate how co-designing the source code impacted the documented regularities. One example of such an erroneous definition of an intensional view was the initial intension of the *Mutators* intensional view. We defined this view as consisting of two alternative views, namely:

```
1.methodWithName(?method,?name),
  instanceVariableInClass(?field,?class),
  [?field asString,':' = ?name asString]
2.statementsOfMethod(statements(<?statement>),?method),
  argumentsOfMethod(<?argument>,?method),
  instanceVariableInClass(?field,?class),
  equals(?statement, assign(variable(?field),?argument))
```

The first alternative expresses that the name of all mutator methods must match the name of a field, followed by a colon. The second alternative captures the implementation idiom that a mutator method must consist of a single statement assigning the value of the only argument to a field. When verifying extensional consistency of this intensional view using our tool suite (see Figure 6.4), we were informed of quite a lot of methods that did not adhere to the second alternative. A detailed inspection of these methods revealed that our initial intension for the second alternative was much too strict. For instance, the method `lastChecked:` implemented in `IntensionalRelation` was defined as:

```
IntensionalRelation>>lastChecked: anObject
  lastChecked := anObject.
  self changed:#relation
```

In addition to assigning the value of `anObject` to the `lastChecked` field, this method also contained a statement that performed a change notification.

We solved this inconsistency by altering the intension of the second alternative view such that it became less strict and verifies whether a method contains an assignment to a field.

```
2.statementsOfMethod(statements(?statements),?method),
  argumentsOfMethod(<?argument>,?method),
  instanceVariableInClass(?field,?class),
  member(assign(variable(?field),?argument),?statements)
```

Note that, since we also encoded a regularity to ensure that all assignments to a field in the core model of IntensiVE are performed strictly by means of a mutator method, this relaxation of the definition of a mutator method did not pose any problems.

Errors in the implementation Another possibility is that our initial assumption was correct, but the source code of IntensiVE (incorrectly) did not adhere to the documented structural source-code regularity. In other words, verifying the validity revealed a number of "errors" in the implementation. While some of these discrepancies were of the stylistic kind, such as source-code entities which were classified in the wrong protocol (in the case of the *Compilation* and *Extension* intensional views) or which did not respect the proper naming scheme, we were able to find a number of more serious violations of regularities. In contrast to the examples above, these errors in the implementation demonstrate how the descriptive use of the documented regularities impacted the source code of IntensiVE.

A first example of such an error we identified was when verifying consistency of the unary constraint:

$\forall$ adapter $\in$ *Extension* :
```
          methodWithNameInClass(?adapter.method,?name,?),
           statementsOfMethod(statements(<
            return(send(
             send(variable(self), tuples,<>),
              ?name,
              ?args))>),?adapter.method)
```

This constraint verifies whether all adapted methods implemented on the `Extension` class follow a similar implementation pattern, namely that they consist of a single statement that returns the value of a delegation of the same method to the variable `tuples`. However, our tool suite informed us of a single method `collect:` implemented in `Extension` that did not adhere to the implementation pattern. This method was implemented as:

```
Extension>>collect: aBlock
  self tuples collect: aBlock
```

The consistency of the constraint failed since we omitted to return the result of the delegation. The correct implementation of the `collect:` method would consist of a single statement `^self tuples collect: aBlock`[2]. Moreover, since in Smalltalk methods without a return statement (implicitly) return the receiver of the message (i.e. the instance of `Extension`), this bug was not immediately apparent at run-time.

Another example of an erroneous implementation of a structural source-code regularity in IntensiVE was identified by checking conformance of the *Hash/equals* intensional view. This intensional view is defined by two alternative views:

```
1. methodWithNameInClass(?method,=,?class)
2. methodWithNameInClass(?method,hash,?class)
```

This intensional view expresses that all methods in IntensiVE implementing the `'='` message must also implement a method `hash` and vice versa. When verifying consistency of this intensional view, the tool suite informed us that the class `ExceptionElement`, which serves as a wrapper around tuples, did implement `'='` but that we incorrectly omitted the hash-operation.

Although these two examples are not critical bugs, they can cause erratic behavior at run-time. What's more, these errors were not detected by the unit tests we have defined for IntensiVE. As such, they provide a nice illustration of how our approach can complement behavioral tests.

All of the discrepancies we discovered in the implementation were resolved by altering the source code. Whether by changing the protocol of a method, an incorrect name, or by making the source code adhere to an implementation pattern, we were able to synchronize the source code and the documented regularities.

[2]Return statements in Smalltalk are indicated by the '^' sign.

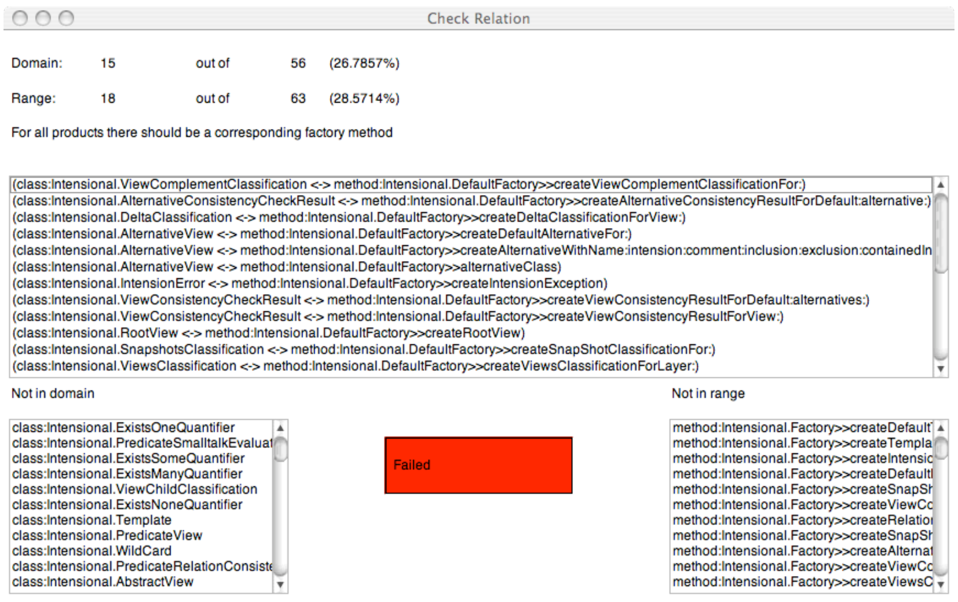
Co-evolution of the documentation and the implementation during the development process

After obtaining a set of consistent intensional views and constraints defined on the implementation of IntensiVE, we incorporated the verification of these views and constraints into the standard testing phase of our tool suite. As such, we used the integration our tool suite offers with the unit testing framework in Smalltalk. This allowed us to verify the validity of the intensional views and constraints at the same time when the unit tests were verified. Consequently we were able to detect infringements on naming conventions, the idiomatic implementation of certain concepts, etc early on in the development process. In particular, we encountered three situations in which either the documentation, the implementation or both artifacts simultaneously needed to co-evolve:

Evolution of the source code We expressed a binary intensional constraint stating that for each class in the core model there should exist a corresponding unit test class. While this regularity held from the start for the vast majority of classes (except for e.g. abstract classes, which we specified as being deviations from the constraint), this implementation regularity aided in ensuring that, whenever the core model of IntensiVE was extended or updated, the corresponding test suite evolved as well. More specifically, it allowed us to maintain some kind of test suite coverage of the implementation. Although this constraint is quite coarse (we do not consider the test coverage of single methods, only of complete classes), this prescriptive use of the documented regularities helped us to assess the impact on the test suite of simple refactorings in the core model.

Evolution of the documentation At the time we created the initial version of the documentation of the regularities underlying IntensiVE, the evaluation of intensional constraints was implemented by a separate class for each combination of a kind of constraint (unary or binary) with a supported query language (SOUL, Smalltalk). For example, the implementation consisted of `SOULUnaryEvaluator`, `SmalltalkBinaryEvaluator` and so on. For the purpose of making it easier to extend our implementation with new query languages and kinds of constraints, we refactored this implementation. We created two separate class hierarchies, namely a class hierarchy implementing the kinds of constraints and a hierarchy of classes representing the evaluators for the specific query languages. We composed the two hierarchies by means of the Strategy design pattern: the evaluator for e.g. a unary constraint can be parameterized with an evaluator for a query language.

This refactoring had a significant impact on the structural source-code regularities underlying the implementation of IntensiVE. In an earlier version, we had a single intensional view *Constraint evaluators* which captured the evaluators for constraints. Furthermore, the naming conventions and inheritance constraints that governed this implementation of the constraint evaluators were documented by a number of constraints imposed on the *Constraint evaluators* view. The verification of this documentation obviously resulted in a large number of reported discrepancies. As such we also had to refactor the documentation. We removed the *Constraint evaluators* view and replaced it with the *Constraint type evaluators* and *Constraint language evaluators* views which correspond with this alteration in the structure of IntensiVE.



```
Domain:   15   out of   56   (26.7857%)

Range:    18   out of   63   (28.5714%)

For all products there should be a corresponding factory method

(class:Intensional.ViewComplementClassification <-> method:Intensional.DefaultFactory>>createViewComplementClassificationFor:)
(class:Intensional.AlternativeConsistencyCheckResult <-> method:Intensional.DefaultFactory>>createAlternativeConsistencyResultForDefault:alternative:)
(class:Intensional.DeltaClassification <-> method:Intensional.DefaultFactory>>createDeltaClassificationForView:)
(class:Intensional.AlternativeView <-> method:Intensional.DefaultFactory>>createDefaultAlternativeFor:)
(class:Intensional.AlternativeView <-> method:Intensional.DefaultFactory>>createAlternativeWithName:intension:comment:inclusion:exclusion:containedIn
(class:Intensional.AlternativeView <-> method:Intensional.DefaultFactory>>alternativeClass)
(class:Intensional.IntensionError <-> method:Intensional.DefaultFactory>>createIntensionException)
(class:Intensional.ViewConsistencyCheckResult <-> method:Intensional.DefaultFactory>>createViewConsistencyResultForDefault:alternatives:)
(class:Intensional.ViewConsistencyCheckResult <-> method:Intensional.DefaultFactory>>createViewConsistencyResultForView:)
(class:Intensional.RootView <-> method:Intensional.DefaultFactory>>createRootView)
(class:Intensional.SnapshotsClassification <-> method:Intensional.DefaultFactory>>createSnapShotClassificationFor:)
(class:Intensional.ViewsClassification <-> method:Intensional.DefaultFactory>>createViewsClassificationForLayer:)

Not in domain                                         Not in range

class:Intensional.ExistsOneQuantifier                 method:Intensional.Factory>>createDefault
class:Intensional.PredicateSmalltalkEvaluat           method:Intensional.Factory>>createTempla
class:Intensional.ExistsSomeQuantifier                method:Intensional.Factory>>createIntensic
class:Intensional.ExistsManyQuantifier      Failed    method:Intensional.Factory>>createDefaultl
class:Intensional.ViewChildClassification             method:Intensional.Factory>>createSnapSh
class:Intensional.ExistsNoneQuantifier                method:Intensional.Factory>>createViewCc
class:Intensional.Template                            method:Intensional.Factory>>createRelatioi
class:Intensional.PredicateView                       method:Intensional.Factory>>createSnapSh
class:Intensional.WildCard                            method:Intensional.Factory>>createAlternat
class:Intensional.PredicateRelationConsiste           method:Intensional.Factory>>createViewCc
class:Intensional.AbstractView                        method:Intensional.Factory>>createViewsC
```

Figure 6.5: The *Relation Consistency Inspector* opened on the binary relation that expresses that for each product there should exist a corresponding factory method.

Moreover, we documented the interface a query language evaluator needs to implement and the composition mechanism for configuring a constraint evaluator with the proper language evaluator.

Evolution of documentation and source code In order to ease with the experimentation of different editors, models of intensional views and constraints, query languages, and so on, we introduced the Factory design pattern in order to abstract from the actual classes implementing the different conceptual entities from the model of intensional views. This factory was not part of the original design of IntensiVE. As such, introducing the factory required us to create factory methods for each of the different entities in the model of intensional views. Moreover, we had to create a hierarchy of factories, each implementing the construction of a different family of entities representing a variation on the model of intensional views.

We supported this evolution task using our approach by creating a number of intensional views and constraints as soon as the base infrastructure of the factory pattern was implemented. These views and constraints document the different factories, factory methods and interactions between these entities.

For instance, we created the binary intensional relation:

$$\begin{array}{l} \forall\, product \ \in\ Products : \\ \quad \exists\, fac \ \in\ Factory\ methods : \\ \qquad \texttt{methodReferencesClass(?fac.method, ?product.class)} \end{array}$$

This relation expresses that for each product, there must exist a corresponding factory method. The documentation of the regularities underlying the instantiation of the factory pattern allowed us not only to verify that our implementation followed the correct implementation pattern and naming scheme or helped us identify improper use of the factory. Since the refactoring of the documentation and the source code happened simultaneously, we were able to "guide" the refactoring using our documentation. For instance, Figure 6.5 demonstrates the *Relation Consistency Inspector* opened on the binary intensional relation we discussed above. This tool reported many violations of the regularity. This is not surprising since, at the time at which we verified this relation, the introduction of the Factory pattern was still work in progress. However, we were able to introduce the design pattern aided by our tool suite. For instance, the information presented in Figure 6.5 allowed us to quickly assess the products for which we still needed to implement a corresponding factory method.

6.1.3 Conclusions

The above case study illustrates the applicability of our methodology. As we observed during the experiment, the step-wise refinement of the intensional views and constraints not only aids in creating accurate documentation of the structural source-code regularities underlying IntensiVE, but particularly helps in identifying source-code entities which do not respect a regularity. As such, this step-wise refinement resulted in the co-design of the documented regularities and the source code. This led for instance to situations such as the *Extension* and *Mutators* intensional views where the discrepancies were caused by documentation that did not prove to be entirely correct. By respectively documenting explicit deviations from the intensional view, or by refining the documentation we were able to resolve these discrepancies. In the experiment above, we also encountered a number of situations (e.g. *Hash/Equals*) for which the discrepancies were the result of an error in the source code. We resolved these situations by adapting the source code of IntensiVE.

Moreover, the case study also demonstrates how our approach can be applied to support the co-evolution of the documentation of structural source-code regularities and the implementation of a system during the development cycle. Incorporating the verification of the documented regularities into the test cycle of our development process enabled us to identify infringements of coding conventions, naming schemes, idiomatic implementations and design conventions early on during development. This was best demonstrated by the gradual introduction of the Factory design pattern. By frequently verifying the intensional views and constraints that document the regularities underlying this design pattern, we were able to detect infringements of naming conventions, proper use of the design pattern, and so on, at the same time that the design pattern was introduced in the implementation.

This case study also illustrated how the refactoring of a part of the implementation – the evaluation of intensional constraints – impacted the structural source-code regularities governing that part of the implementation. This restructuring resulted that, upon verification, our tool suite indicated that a number of intensional views and constraints were no longer synchronized with the implementation. Due to these alterations to the source code of IntensiVE, we updated our documentation such that it reflected the changed set of regularities.

6.2 SmallWiki

6.2.1 Overview

The second case study we consider in this chapter is SmallWiki [Ren03, DRW05]. SmallWiki is a collaborative wiki system implemented in VisualWorks Smalltalk, developed by Lukas Renggli of the university of Bern. Similar to the previous experiment on IntensiVE, we created a number of intensional views and constraints for the SmallWiki system which encode the structural source-code regularities underlying SmallWiki's implementation. In order to assess how our approach is able to maintain the causal link between the regularities and the source code of SmallWiki, we performed the following experiment:

- Following our iterative methodology we created a number of views and constraints on version 1.54 (14/12/2002) of SmallWiki. This version was the first internal release of the project and consists of 63 classes on which 424 methods were implemented. This version of SmallWiki incorporated the base structure for the wiki system: a representation of a wiki document together with basic facilities to display and store such wiki documents;

- In order to assess the impact of evolution on the documented structural source-code regularities, and in order to verify how our approach can support the co-evolution of the documented regularities and the implementation we applied the intensional views and constraints we created on version 1.54 of SmallWiki to version 1.90 of the same project. Version 1.90 (15/01/2003) is an internal release dating one month after the initial 1.54 release and consists of 71 classes and 633 methods. The major difference between this release and the 1.54 release is that the mechanism for outputting HTML had been completely refactored;

- We finally reapplied the documented structural source-code regularities to version 1.304 (16/11/2003) of SmallWiki. This version entailed 108 classes and 1219 methods, thus being considerably larger than the previous two versions. Since this version was released almost a year after the initial version 1.54, on which we started the experiment, this allowed us to assess the impact of evolution of SmallWiki on the regularities over a longer period of time.

The most significant difference between this experiment and the previous one is that during this experiment, we assessed the evolution of the different structural source-code regularities *a posteriori* instead of during the actual development of the system. As such, this case study served as a means to demonstrate that during development of a system, changes to the source code can break existing regularities. Furthermore, this case study also demonstrates that such regularities can evolve over time, and that an approach such as intensional views can be used to identify and resolve these evolution conflicts.

6.2.2 Intensional views and constraints over SmallWiki 1.54

We documented the coding conventions, naming conventions and design dependencies in the initial release of SmallWiki (version 1.54) using intensional views and intensional constraints.

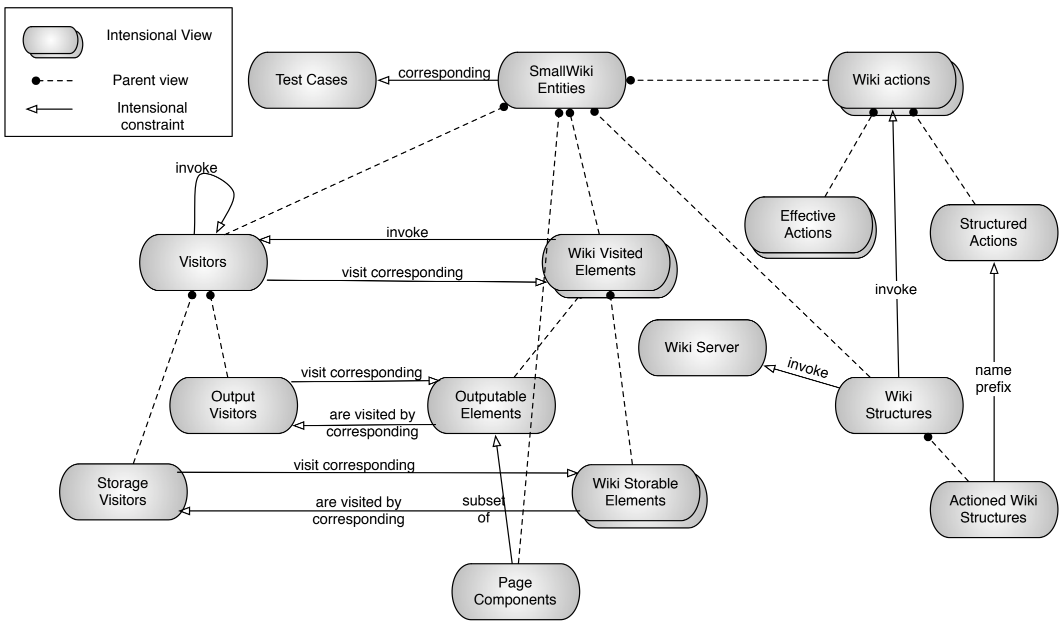


Figure 6.6: Overview of the intensional views and constraints over SmallWiki.

We created 15 intensional views of which 4 have multiple alternative views. Moreover, we defined 12 intensional constraints on these intensional views. An overview of these views and constraints is illustrated in Figure 6.6. For a full overview of the implementation of these views and constraints we refer to Appendix C.

These intensional views and constraints are an adaptation[3] of a previous experiment which we reported on in [MKPW06]. In this experiment, we started out with limited knowledge about the implementation of SmallWiki. By studying both the documentation of SmallWiki and its implementation, and by iteratively refining our documentation we achieved a good understanding of the workings of the wiki system. Each time we discovered a concept or a dependency in the implementation of SmallWiki, we documented it using an intensional view or constraint. By verifying this documentation, we were able to review our assumptions about the internals of SmallWiki.

We started out by creating an intensional view *SmallWiki Entities* grouping all classes and methods in the implementation of SmallWiki. Next, we studied the major class hierarchies that were present in the implementation. Each of these hierarchies aligned with a specific concern in SmallWiki:

- *Wiki Structures*: all classes in the hierarchy of `Structure` represent wiki entities that can be referred to by a single URL such as for instance a web page or a chapter in the wiki;

[3]We restructured the intensional views and expressed them using the new formalism and model we explained in Chapter 3.

- *Page Components*: the different components which a wiki page consists of such as text, links, tables, headers are implemented by a subclass of `PageComponent`;
- *Wiki actions*: the classes in the hierarchy of `Action` each represent an action that can be performed on a SmallWiki entity such as *cancel*, *edit*, *save*, etc. By inspecting this class hierarchy, we noticed that all methods implementing an actual action started with the prefix "execute-" and were classified in the protocol `action`. We encoded this regularity by means of alternative views for the *Wiki actions* view. All effective, non-abstract actions are collected in the *Effective actions* intensional view. We documented the regularity that each of these classes must provide a method name `execute`;
- *Wiki Server*: the classes in the `WikiServer` hierarchy implement a particular kind of SmallWiki server (e.g. the class `SwazooServer` that makes use of the `Swazoo` network library);
- *Visitors*: the `Visitor` hierarchy implements an instantiation of the Visitor design pattern that allows performing various operations on a wiki document. A closer study of this `Visitor` hierarchy revealed that SmallWiki provides two different kinds of visitors:
  - *Output Visitors*: the visitors responsible for generating output (HTML, Latex, ...);
  - *Storage Visitors*: the visitors responsible for persistently storing a wiki document;

A closer inspection of the `Structure` and `Action` hierarchies revealed that both hierarchies are related:

- *Actioned Wiki Structures*: we noticed every class in the `Action` hierarchy implemented an action for a particular element of the `Structure` hierarchy. For example, the *edit* operation on a `Page` was implemented by a class `PageEdit`. We grouped the classes which such an action is implemented for in the *Actioned Wiki Structures* view;
- *Structured actions*: this intensional view is the dual of the *Actioned Wiki Structures* intensional view: while the *Actioned Wiki Structures* intensional view captured all wiki structures which a number of actions are defined on, this intensional view groups the implementation of those actual actions;

The presence of the Visitor design pattern resulted the documentation of the entities that are visited by such a Visitor and regularities that govern these entities:

- *Wiki Visited Elements*: all wiki entities which are visitable by a visitor. This intensional view expresses the regularity that all such elements should implement a method `accept:` which is classified in a protocol `visiting`;
- *Outputable Elements*: all wiki entities which are visitable by an output visitor;
- *Wiki Storable Elements*: all wiki entities which are visited by a storage visitor. By means of an alternative view we document the constraint that this set of storable elements equals the set of wiki structures, i.e. all wiki structures must be storable;

Finally, we grouped all the unit tests over SmallWiki in the intensional view *Test Cases*. We also discovered a number of constraints that govern the above intensional views:

- We documented the naming convention that for all *Actioned Wiki Structures* classes, there must be a *Structured action* class such that the name of the wiki structure is a prefix of the name of the action;
- In order to be able to deal with an action on a wiki document, all *Wiki Structures* classes must invoke a *Wiki action*;
- The *Wiki Structures* must communicate with the *Wiki Server*;
- All *Page Components* must be outputable, as such, they must be a subset of the *Outputable Elements* intensional view;
- For all *Output Visitors* there must be an *Outputable Element* such that the name of the outputable element matches the visit method on the output visitor. We also express the dual of this constraint, namely that for each outputable element there should be a visit method on an output visitor by means of a binary intensional constraint;
- For all *Storage Visitors* there must be a *Wiki Storable Element* such that the name of the storable element matches the visit method on the storage visitor. Similar to the above constraint, the dual of this binary intensional relation is documented;
- For every visit method belonging to *Visitors*, there must be a corresponding *Wiki Visited Element*;
- All accept methods on *Wiki Visited Elements* must invoke a visit method on *Visitors*;
- For all classes in *SmallWiki Entities* there should be a corresponding *Test Case*;
- The visit methods in the *Visitors* intensional view are implemented by either a message send to a `stream` or by an invocation of another visit method.

6.2.3 Supporting evolution of the documented structural source-code regularities and the implementation

In this section we assess the impact of evolution on SmallWiki by applying the intensional views and intensional constraints to versions 1.90 and 1.304 of the wiki system.

Impact of evolution on version 1.90

We verified conformance of the documentation of the structural source-code regularities to version 1.90 of SmallWiki. The result of verification can be seen in Figure 6.7 presenting the output of our *Visualization tool*. In particular, we observed that a number of views and constraints were violated.

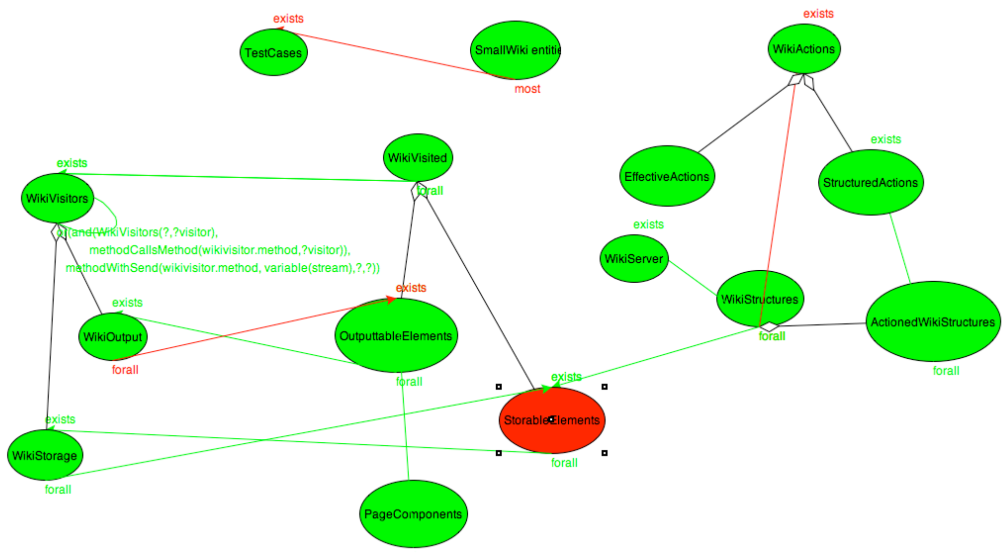
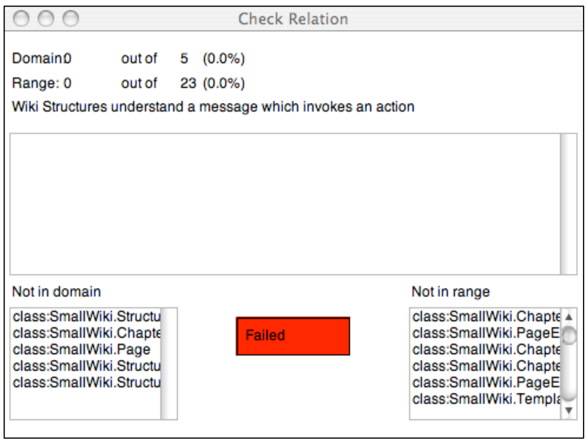


Figure 6.7: The *Visualisation tool* illustrating conformance of the set of intensional views and constraints applied to version 1.90 of SmallWiki

Violation of a stylistic regularity We documented the naming convention that the first argument of a method on an output visitor must correspond to the name of the visited elements. However, the binary intensional relation (between the *Output Visitors* and *Outputable Elements*) expressing this convention failed in version 1.90. A closer inspection of the discrepancy indicated a single method named `acceptOrderedList:` in the `VisitorOutputLatex` that erroneously deviated from this convention (the argument was named `numberedList` instead). We correct this small discrepancy by altering the source code;

Incorrect documentation In our documentation of version 1.54 of SmallWiki, we explicitly included the class `Document` as a deviation to the second alternative view of the *Wiki Storable Elements* view. The reason for this was that the second alternative of this view stated that all *Wiki Structures* must be storable. The `Document` class was not a part of the *Wiki Structures* view; however, according to the implementation it was clearly a storable element since there existed a corresponding method on a storage visitor. In version 1.90 this explicit deviation caused the *Wiki Storable Elements* intensional view to become inconsistent. An inspection of the discrepancies illustrated that due to a change in the implementation, this corresponding method for the `Document` class was no longer present in the storage visitor. We made the documentation consistent again with the implementation by removing this explicit deviation;


Check Relation

Domain0 out of 5 (0.0%)
Range: 0 out of 23 (0.0%)
Wiki Structures understand a message which invokes an action

Not in domain
class:SmallWiki.Structu
class:SmallWiki.Chapte
class:SmallWiki.Page
class:SmallWiki.Structu
class:SmallWiki.Structu

Not in range
class:SmallWiki.Chapte
class:SmallWiki.PageE
class:SmallWiki.Chapte
class:SmallWiki.Chapte
class:SmallWiki.PageE
class:SmallWiki.Templa

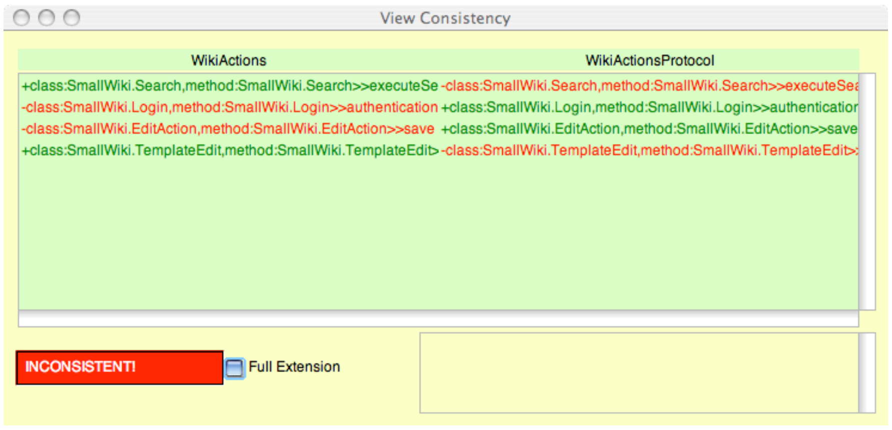
Figure 6.8: The *Relation Consistency Inspector* opened on the relation between the *Wiki Structures* and *Wiki action* views.

Changes in the design of SmallWiki Our tool suite also reported that the binary intensional relation stating that all elements of *Wiki Structures* invoke a *Wiki action* was violated. In version 1.54 of SmallWiki, a wiki structure would receive a request from the server to perform a certain action. To this end, the structure would create an instance of a wiki action, execute the action and return the result. Verification of this regularity (see Figure 6.8) informed us that none of the *Wiki Structures* respected this regularity. An inspection of the source code of version 1.90 of SmallWiki indicated that the mechanism discussed above was refactored. Rather than handling an action by itself, a wiki structure delegated this to an instance from the `Renderer` hierarchy. We documented the effect of this refactoring on the structural source-code regularities by creating an intensional view *Renderer*. We documented the dispatch mechanism by a unary constraint expressing that all `Wiki structures` must dispatch the action to a `Renderer`.

This inspection of the rendering facilities also indicated that the wiki actions in version 1.90 use a separate class for generating HTML code (`HTMLWriteStream`) rather than by outputting HTML code themselves (as in version 1.54). We also documented this change in the implementation of SmallWiki by creating a binary intensional relation expressing that wiki actions must use this mechanism. When verifying this regularity, we noticed a large number of discrepancies. Since this change in the design of SmallWiki was still a work in progress, we did not resolve these discrepancies.

Impact of evolution on version 1.304

We repeated the above experiment by applying the intensional views and constraints we obtained after version 1.90 of SmallWiki to version 1.304 of the wiki system. While the time interval between version 1.90 and 1.304 was quite large, the documented structural source-code regularities appeared to have remained fairly stable. However, verification of the documentation did report discrepancies between the regularities and the implementation. In what



WikiActions	WikiActionsProtocol
+class:SmallWiki.Search,method:SmallWiki.Search>>executeSe	-class:SmallWiki.Search,method:SmallWiki.Search>>executeSea
-class:SmallWiki.Login,method:SmallWiki.Login>>authentication	+class:SmallWiki.Login,method:SmallWiki.Login>>authenticatior
-class:SmallWiki.EditAction,method:SmallWiki.EditAction>>save	+class:SmallWiki.EditAction,method:SmallWiki.EditAction>>save
+class:SmallWiki.TemplateEdit,method:SmallWiki.TemplateEdit>	-class:SmallWiki.TemplateEdit,method:SmallWiki.TemplateEdit>

Figure 6.9: The *Extensional Consistency Inspector* opened on the *Wiki actions* intensional view.

follows, we discuss a number of examples of such violated regularities.

Violation of a stylistic constraint We documented a binary intensional relation that expresses that the single argument of all visit method in *Storage Visitors* must match the name of the *Wiki Storable Element* that is being visited. E.g. the argument of the method `acceptPage:` must contain the string 'Page'. This constraint was violated by the `acceptLinkInternal:` method in the `LinkInternalVisitor` class. In this method, the argument was named `anInternalLink` rather than the expected `aLinkInternal`. We resolved this minor stylistic inconsistency by adapting the source code.

Violation of a regularity as well as an inconsistency in the source code Our tool suite reported that the intensional view *Wiki actions* was inconsistent (see Figure 6.9). This intensional view was defined by two alternative views:

```
1.['execute*' match: ?method selector asString]
2.methodInProtocol(?method, action)
```

The first alternative states that all methods with the prefix "execute-" are wiki actions; the second alternative selects all methods classified in the protocol `action`. An inspection of inconsistencies between the two alternative views of the view brought it to our attention that there were two methods, namely `executeSearch` and `executePermission` which were clearly captured by the first alternative view, but which were incorrectly classified in another protocol than the protocol `action`. We resolved this situation by classifying both methods in the correct protocol. Moreover, there were two methods `save` and `authenticate` which did not follow the first naming convention, but which were classified in the protocol `action`. We documented these two methods as explicit deviations to the first alternative view of *Wiki actions*;

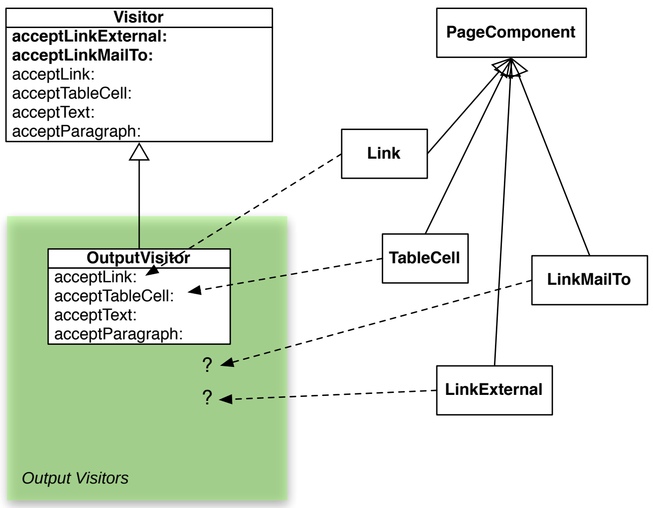


Figure 6.10: Diagram representing the problem with the *Page components* and *Output Visitors* intensional views.

Incorrect documentation The binary intensional relation declaring that all page components have a corresponding visit method on an output visitor failed. This relation was implemented as follows:

$$\begin{aligned} \forall\, \text{component} &\in \textit{Page components} : \\ \exists\, \text{outputable} &\in \textit{Outputable elements} : \\ &\text{component} = \text{outputable} \end{aligned}$$

Figure 6.10 illustrates the reason for the discrepancies of this intensional relation. The right-hand side of the diagram represents a part of the hierarchy of page components in the SmallWiki system. The left-hand side shows two classes from the implementation of the Visitor pattern. The regularity requires that for each page component there exists a corresponding visit method on an output visitor. This is encoded by requiring that each page component is also a part of the *Outputable elements* view (i.e. the set of SmallWiki entities that is visited by an output visitor).

For a number of page components (in the figure the `LinkExternal` and `LinkMailTo` classes), none of the visitors belonging to the *Output Visitors* intensional view implement a corresponding accept method. Rather, the accept methods for these page components were implemented on the abstract class `Visitor`. In the abstract class, these accept methods then delegated to a more specific accept method (e.g. the method `acceptExternalLink:` delegate to `acceptLink:`). As such, these classes are not part of the *Outputable elements* view thus resulting in the discrepancies. We resolved this inconsistency by documenting the discrepancies as explicit deviations from the *Outputable elements* view.

6.2.4 Conclusions

We can make observations about SmallWiki similar to those we made about IntensiVE case study. While the initial version of the SmallWiki system we documented was quite small, resulting in that only a handful of interesting regularities governed the system, the case study allows us to assess the impact of evolution on the documentation of this set of regularities. In analogy to the verification of the intensional views and constraints over IntensiVE, verifying consistency of the documentation we created over the implementation of SmallWiki resulted in the two following situations:

- We encountered a number of situations in which discrepancies between the intensional views/constraints and the implementation were caused by source-code entities which did not abide with certain documented structural source-code regularities. In order to render the documentation and the implementation consistent again, we had to alter the source code of the system in order to correct the violated regularity or in the case of an exception to the regularity we had to explicitly document it as a deviation from the view/constraint;

- In other situations, the structural source-code regularities had evolved. Whether due to changes in the conventions followed by the developer or a refactoring of the design of the system, the documented structural source-code regularities no longer reflected the current conventions which governed the system. In these cases we had to update the definitions of the intensional views and intensional constraints in order to align with the evolved regularities.

This experiment again demonstrates the need for co-design and co-evolution of the documented structural regularities and the implementation. Moreover, it also illustrates that the intensional views and constraints over SmallWiki were able to detect inconsistencies between the documented regularities and the source code of SmallWiki, which arose from the evolution of the SmallWiki system.

Another observation we can make about the experiment on SmallWiki is that the step-wise definition and verification of intensional views and constraints aided in us getting a better understanding of the internals of SmallWiki. When initiating the case study, we started out with little or no knowledge about the SmallWiki system. We created a number of intensional views which capture high-level concepts in SmallWiki like for instance the page structure, wiki actions, and so on. In general, these initial intensional views aligned with the different class hierarchies in the implementation of SmallWiki. Furthermore, we encoded how we assumed the different concepts in SmallWiki are structured or how they interact by means of constraints over these intensional views. By verifying these assumptions with respect to the source code, we gained a better understanding of the implementation of SmallWiki. This resulted in us being able to refine the documentation of SmallWiki by creating intensional views which capture more detailed concepts such as the different types of visitors (for storage, for output), and so on. While this gradual construction of documentation for the regularities governing SmallWiki required a larger effort than the effort required for documenting a system for which we have expert knowledge like IntensiVE, we feel that this documentation process aided in understanding the internals of SmallWiki. While we cannot claim in general that documentation of structural regularities using intensional views aids in comprehending a piece of software, for SmallWiki we experienced that the process of alternating between exploring the source code of SmallWiki and documenting/verifying how we assumed that SmallWiki is structured, resulted in a better understanding of the SmallWiki system.

6.3 DelfStoF

6.3.1 Overview

The final case study we consider in this chapter is DelfSTof. DelfSTof is a framework for performing formal concept analysis (FCA) [GW99]. FCA is a data mining technique that, given a set of objects[4] and a set of attributes describing those objects, returns maximal groups of objects according to the attributes they share. In other words, the result of performing formal concept analysis on a group of objects is that the algorithm returns the subsets of objects that, according to the attributes describing the objects, are similar. DelfSTof was implemented mainly by Tom Tourwé and Kim Mens in the context of their work on using FCA in order to identify crosscutting concerns [MT05, TM04]. DelfSTof's initial release consists of 86 classes implementing 440 methods.

While the implementation of the core FCA algorithm is monolithic and low-level for efficiency reasons, DelfSTof also consists of a number of classes which provide additional functionality to ease experimenting with different sets of objects and attributes. More precisely, DelfSTof provides a number of customization points:

- An instantiation of DelfSTof can specify the set of objects which FCA is applied to;
- DelfSTof provides a set of classes for deriving attributes for each object. Especially for analyzing source code, the framework includes a parse tree traversal algorithm that can be configured to extract certain properties;
- In order to decrease the size of the output, DelfSTof provides an extensive filtering framework to filter the considered attributes before the actual concept analysis is performed as well as the set of concepts returned by the algorithm;
- DelfSTof provides a means to implement the grouping of concepts with similar properties (e.g. all concepts containing polymorphic methods, ...). These so-called concept analyzers can be instantiated to analyze the concepts returned by the FCA algorithm and group them in a specific way;
- The framework provides the notion of a *context*. Such a context is a class representing the configuration of the FCA algorithm. This context specifies the set of objects that is used by the algorithm and the attribute creator used to generate the attributes of the objects. Furthermore, such a context also selects the filters and analyzers that are applicable to a specific experiment.

[4]Notice that in FCA terminology the term objects not used in the sense of 'object-oriented programming' but rather the set of objects the FCA algorithm will analyze.

6.3.2 Intensional views and constraints over DelfSTof

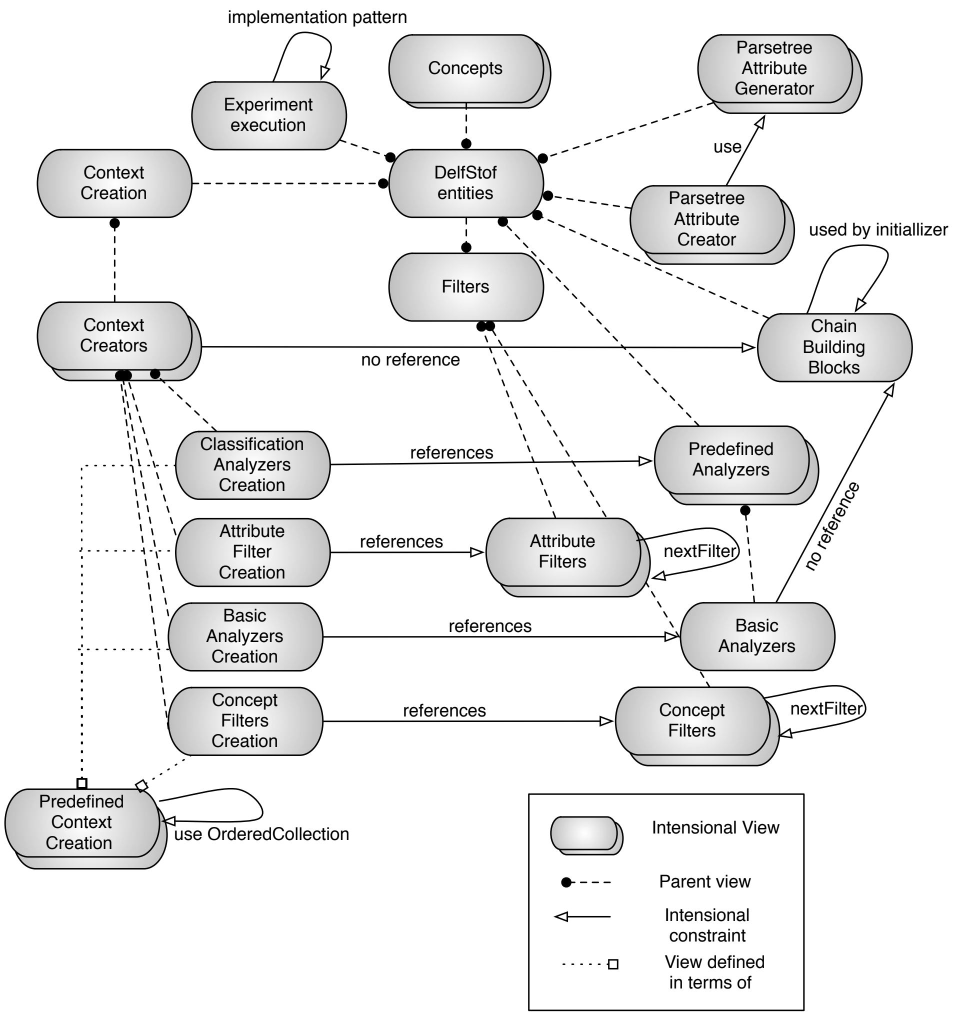


Figure 6.11: Overview of the intensional views and constraints over DelfSTof.

Figure 6.11 gives an overview of the intensional views and constraints we defined on DelfSTof. For this experiment – which is based on our previous work as reported in [MK06] – we explicitly focussed on documenting the structural source-code regularities that are concerned with properly instantiating the framework. Therefore we documented a number of regularities which document the set of methods (and their implementation pattern) that need to be implemented by a class that customizes a particular aspect of DelfSTof. Moreover, we also documented the correct usage of the context creation: since the filters and analyzers are arranged as a Chain of responsibility, we created a number of views and constraints that detect instances of improper usage of the chain. Furthermore, we also provided a number of regularities that aid in verifying that the context creation process is performed correctly. More specifically,

- The *DelfStof entities* intensional view captures all the source-code entities in the implementation of DelfSTof;

- *Filters* groups the implementation of the filtering mechanism. Since DelfSTof makes a distinction between filters that are applicable to attributes and filters that are applicable to concepts, we provide two children of the *Filters* intensional view namely *Attribute Filters* and *Concept Filters*. We impose a number of constraints over these intensional views which govern the implementation of the filters.

- In analogy to filters, we also document the concept analyzers (i.e. the classes responsible for grouping related concepts returned by the FCA algorithm). The analyzers that group concepts containing the same kind of objects (i.e. classes, methods and parse trees) are grouped in the *Basic Analyzers* view. The concept analyzers that perform a more "semantic" grouping of concepts (for instance all concepts that contain only polymorphic methods) are gathered in the *Predefined Analyzers* view;

- The most important concept in the DelfSTof framework is that of a context creator. A subclass of `ContextCreator` contains a number of factory methods that specify the filters and analyzers used by a particular FCA experiment. We document these context creators using the intensional view *Context Creators*. The set of concrete context creators supplied by DelfSTof is grouped by the *Predefined Context Creation* view. Furthermore, we create for each kind of factory method implemented by a context creator a separate intensional view. Namely, we implement the *Classification Analyzers Creation*, *Attribute Filter Creation*, *Attribute Filter Creation* and *Basic Analyzers Creation* views.

  We also document a number of regularities that govern these context creators. First of all, we specify the regularity that the methods in the *Predefined Context Creation* must return an instance of `Collection` containing the different filters or analyzers that are applicable. Second, we document four binary intensional relations that express the regularity that each kind of factory method must return a collection containing the correct kind of entities. For example, we require that the methods in the *Basic Analyzers Creation* view only refer to *Basic Analyzers*.

- The *Parsetree Attribute Creator* and *Parsetree Attribute Generator* intensional views group source-code entities which are responsible for extracting properties from a parse tree of a method. Moreover, we define a constraint which states that the parse tree attribute creator must use a parse tree attribute generator;

- The *Concepts* intensional view groups the implementation of different kinds of concepts, as they are created by the concept analyzers;

- *Experiment execution* contains the top-level methods which are used to initialize and invoke an experiment using FCA. Since all these methods follow a similar implementation pattern, we document this pattern.

Similarly to the two other case studies, we do not illustrate the actual implementation of all the intensional views and intensional constraints here but rather refer the interested reader to Appendix D.

6.3.3 Supporting evolution of the documented structural source-code regularities and the implementation

In order to assess how our approach and tool suite support the evolution of the regularities we documented on the implementation of DelfSTof, we performed two experiments. First, we applied the set of intensional views and constraints to multiple subsequent versions of the DelfSTof framework. We studied how changes in the implementation affected the structural source-code regularities and how our approach allowed us to deal with this evolution. Second, we applied our documentation to one particular instantiation of the DelfSTof framework.

Supporting evolution over different versions of DelfSTof

In line with the other case studies, we applied our iterative methodology to create a first version of the intensional views and constraints over DelfSTof. We thus started by encoding an assumption about the structural source-code regularities in the framework. By verifying the validity of this assumption with respect to the implementation and by refining our views/constraints and the source code, we ended up with a set of intensional views and constraints which reflected the structural source-code regularities of DelfSTof. During this iterative process, we did not only refine our documentation but also encountered a number of source-code entities which did not respect some of the naming conventions or which deviated from the documented implementation patterns.

When verifying the validity of this set of intensional views and constraints to subsequent versions of DelfSTof, we did not encounter much discrepancies. Our tool suite identified a number of instances of source-code entities which did not respect a naming convention. Furthermore, we encountered one refactoring in the implementation of the classification analyzers that resulted in that we needed to update our documentation. However, over time the structural source-code regularities in DelfSTof did evolve little, nor did the implementation deviate much from these regularities.

This can be explained by the fact that the architecture of the framework was simple and did not significantly evolve during the different versions. Although the size of the code base of DelfSTof increased significantly (from 86 classes and 440 methods to 192 classes on which 747 methods are implemented), most of these extensions did not alter the framework but rather extended the default set of filters and analyzers that were offered by DelfSTof. Moreover, since all of the changes were made by the two original developers of the framework, and due to the simplicity of the framework, almost none of these extensions infringed the structural source-code regularities governing DelfSTof.

Applying the documentation to the instantiation of DelfSTof

Applying the intensional views and constraints we defined over DelfSTof to a particular instantiation of the framework yielded more interesting results. A developer who was novel to DelfSTof had customized and used the framework in order to detect refactoring opportunities based on duplicated code, i.e. pieces of code with similar parse trees. As such the developer had to customize the framework in a number of ways:

- The parse tree attribute creators, i.e. the part of DelfSTof that allows the traversal of a parse tree of a program and extracts the information of interest for a certain experiment, needed to be extended such that for any given method, an abstract parse tree (a parse tree in which all literals have been omitted) can be extracted;
- A number of analyzers had to be implemented which group concepts on which a similar object-oriented refactoring is applicable;
- In order to reduce the number of false positives, the developer needed to implement certain attribute filters which removed trivial parse trees, as well as a couple of concept filters which pruned away uninteresting results;
- Depending on the case study which the developer's instantiation of DelfSTof needed to be applied to, a different context needed to be created.

The developer did not have any *a priori* knowledge about DelfSTof and studied other instantiations of the framework in order to learn its internals before implementing his own instantiation.

We used the set of intensional views and constraints we defined over DelfSTof as a kind of prescriptive documentation to verify whether the particular instantiation of the framework respected the different regularities that govern DelfSTof. When verifying consistency of these views and constraints we were notified by our tool suite of two situations in which the framework customizer circumvented the facilities offered by the framework, and where the regularities governing the framework were thus violated.

Violation against the implementation pattern of *Experiment execution* Verifying the validity of the documented regularity informed us that the implementation pattern *Experiment execution* was violated.
This regularity was implemented as follows:

$\forall$ experiment $\in$ *Experiment execution* :

```
statementsOfMethod(statements(<?statement>),?experiment.method),
equals(?statement, send(variable(self),
      [#runAnalysisOnObjects:forCase:],
      <send(variable(self),?,<>),literal(?project)>)),
 [('runOn', ?project asString)
    match:(?experiment.method selector asString)]
```

The above unary constraint specifies that all methods implementing the top-level invocation of an experiment must consist of a single statement that invokes the `runAnalysisOnObjects:forCase:` message. Furthermore, the regularity requires that the second argument of this invocation (i.e. the project name) matches the name of the method implementing the experiment execution. Let us illustrate this regularity by an example. The method `runOnSoul` in the `ContextCreator` class is implemented as follows:

```
runOnSoul
  self
    runAnalysisOnObjects: self soulObjectCreatorClass
    forCase: 'Soul'
```

This method respects the regularity since it invokes the `runAnalysisOnObjects:forCase:` message and the second argument (namely 'Soul') matches the method name (`runOnSoul`).

The customizer's implementation violated the above regularity by not using the `runAnalysisOnObjects:forCase:` but rather invoking the `performAnalysisOnObjects:` method. While this method also initializes a FCA experiment, this was not the intended use of the framework. As a result, the additional behavior provided by `runAnalysisOnObjects:forCase:` – namely logging and benchmarking of the performed experiment – did not get executed.

Violation of chain of responsibility Our tool suite also informed us that the instantiation of DelfSTof violated the regularity that *Context Creators* are not allowed to refer directly to the basic *Chain Building Blocks*. This regularity documents the creation of the chain of responsibility for the filters and analyzers in the DelfSTof framework. A developer implementing a customization of the framework must, among others, implement a method `basicAnalyzers` that returns a collection of analyzers that are to be used in the FCA experiment. The internals of this framework will then transform this collection of analyzers into a chain and correctly terminate this chain by an instance of the `DefaultAnalyzer` class. Consequently, a framework customizer should not manually add this `DefaultAnalyzer` class to the collection of analyzers. The framework customizer was not aware that this termination of the chain was performed by the framework and explicitly added the `DefaultAnalyzer` to the collection of analyzers in his instantiation. Consequently, our tool suite reported that this addition of `DefaultAnalyzer` violated the regularity.

6.3.4 Conclusions

Although the second phase of the experiment we described above – in which we applied our documentation to an instantiation of DelfSTof– is technically not an instance of co-design or co-evolution, since only the source code evolves without us having to evolve the documentation, it illustrates an interesting application of our approach. We started by documenting the regularities underlying the framework in order to co-evolve the implementation of the framework with this documentation. Since our intensional views are expressed by means of an intensional description, they cannot only be used to capture the source-code entities belonging to the implementation of the framework but also to capture source-code entities belonging to extensions and instantiations of this framework. As the different structural source-code regularities we documented not only express a number of naming conventions, implementation schemes, prohibitions, and so on about the implementation of the framework, but also about the instantiation of this framework, this documentation can serve as a means to verify the instantiation of the framework. By verifying conformance of the intensional views and constraints that are defined over a framework during the implementation of an instantiation, it is possible to detect certain infringements of the structural and stylistic regularities governing the framework.

6.4 Discussion

Evolution of structural source-code regularities

The set of structural source-code regularities of a software system is not a static entity. One of the observations we were able to make about the case studies we conducted above is that over time, not only the source code of a software system evolves, but also the regularities which govern this software system. For instance, we encountered a number of situations in which the implementation pattern for a certain concept needed to evolve in order to capture changes in the system's design. One example of such an evolution conflict occurred in the SmallWiki case study. In this case study, we expressed – using a binary intensional relation – that for all storable elements of a wiki page, there must be a corresponding visit method on the storage visitor. While this regularity held for the first versions of SmallWiki, the introduction of a number of page elements which reused the storage mechanism from another element rather than implementing its own resulted in that we needed to update the documentation of the regularities slightly in order to capture the new situation. Similarly, other changes in the design of the application can have an impact on regularities which express how certain design elements must interact. For example, in IntensiVE we refactored the evaluation mechanism of intensional constraints. This refactoring had a drastic impact on the structural source-code regularities that were documented in this part of the implementation of IntensiVE: while the existing regularities were no longer valid, the refactoring introduced a number of new regularities. We encountered a similar situation in the SmallWiki case study. In between version 1.54 and version 1.90 of the wiki system, the rendering of HTML pages was significantly altered. As such, we had to alter our set of intensional views in order to capture this new rendering system and the structural dependencies which were introduced upon the rendering system. Finally, changes to the implementation of a system can give rise to totally new structural source-code regularities. One instance of such a situation occurred with the introduction of the Factory design pattern in IntensiVE's implementation.

Support for co-evolution

The case studies we explained in this chapter illustrate that our approach is suitable for documenting structural source-code regularities in object-oriented systems, as well as for supporting the co-evolution of this documentation with the implementation of the system. First, this support is achieved by the fact that the documentation we create using intensional views and intensional constraints is active and verifiable. The intensional views defined on a system are expressed in terms of the actual implementation entities, thus providing active documentation of concepts in the source code of the system. Furthermore, the constraints imposed on these views are verifiable with respect to the source code. This enables the identification of discrepancies between the documented regularities and the source code, thus aiding in maintaining the causal link between these regularities and the implementation.

Second, the intensional description of the set of tuples belonging to an intensional view aids in supporting evolution of the documented source-code regularities. Since the set of tuples belonging to an intensional view is calculated, this allows – upon evolution – the automatic classification of newly added or modified source-code entities, thus providing a means to verify the validity of the different constraints with respect to these changes in the source code.

Third, our approach is non-coercive: the different constraints we impose on a system are not enforced by our system but our tool suite rather informs a developer of discrepancies between the regularities and the implementation. This enables a developer to deviate temporarily from the documented regularities during development, providing more flexibility than a coercive approach in which the different regularities are enforced.

Iterative definition of the documentation and co-design

One of the cornerstones of the methodology we advance in this dissertation is the incremental and iterative definition of the documentation of structural source-code regularities as a means to support the co-design of regularities and source code. Using this scheme, a developer performs a step-wise refinement of documented structural source-code regularities by alternating between adapting the documentation and verifying the documentation with respect to the implementation. In addition to providing support for verifying conformance of the documentation, our approach and tool suite support this iterative methodology by the construct of explicit deviations from an intensional view or an intensional constraint. This mechanism of deviations makes it possible to explicitly include or exclude a particular tuple from the extension of an intensional view, or explicitly to mark for a tuple that a certain constraint does (not) hold. In practice, these deviations proved to be quite useful: although the iterative methodology we applied when constructing a set of views and constraints resulted in that upon each iteration, the documentation more closely matched the actual concretization of a regularity in the source code, we often encountered source-code entities which substantially deviated from the regularity and as such were documented as an exception to the general rule. For documentation purposes, the fact that deviations to a view/constraint are explicitly present in the model of intensional views aids in constantly keeping a developer aware of them.

A side-effect of this iterative inception of the documentation for structural source-code regularities is that the co-design of documentation and source code can be considered to be a kind of "co-evolution in the small". While the discrepancies between the documented regularity and the implementation during this step-wise process often indicated a possible refinement of the documentation, we also encountered a number of situations – as we have illustrated in the case studies – in which inconsistencies were the result of an error in the source code of the system. As such, the discrepancies which are encountered at definition-time of the documentation can be – depending on the expert knowledge of the developer – interpreted as being an error in the documentation, an error in the implementation or an exception to the regularity.

Mutual dependencies between intensional views

We learned from our case studies that a developer cannot consider the different intensional views and constraints in isolation. For instance, when resolving an inconsistency between an intensional view or constraint and the implementation by either updating the view/constraint and/or the implementation, it does not suffice to verify the consistency of that single intensional view or constraint. Due to the parent view scoping mechanism of intensional views, the fact that one view can be expressed in terms of another and that intensional constraints rely on the intensional views for which they are defined, changes in a single intensional view or changes to a source-code entity which belongs to the extension of an intensional view, can have an impact on multiple entities in our documentation.

While this property of our approach contrasts with the isolation property of unit tests which by definition are independent of each other, this mutual dependency between different intensional views and constraints does not have to be considered a disadvantage of our approach. Often, local changes in the source code of a system can impact a number of concerns spread throughout the implementation of a system. The same observation holds for structural source-code regularities: in order to address an inconsistency of a certain regularity, the source code of a system can be altered such that another regularity in the system is violated. While this has as effect that in practice, often an entire set of intensional views and constraints needs to be verified with respect to the source code, it allows us to detect instances of changes in the implementation (or the documentation) which have widespread consequences throughout the implementation and/or documentation.

Costs versus benefits

An important question concerning the usefulness of our approach is whether the effort that a developer needs to invest in order to document the structural source-code regularities governing a system using intensional views and constraints is outweighed by the advantages that are gained by the consistent adherence of the source code to the documented regularities. Since the three case studies we reported on in this chapter are modestly-sized, and only a relatively small number of intensional views (up to 41) and constraints (up to 23) have been defined, it is impossible to draw any conclusions about whether or not applying our approach is worth the effort for large-scale, real-life systems. Such an assessment is thus the topic of further study.

In this chapter we have illustrated only a single use of documentation created using intensional views and constraints. In particular, we have demonstrated the applicability of our approach for documenting the different structural source-code regularities governing a system. By integrating the regularities in the development process, we were able to keep this

	IntensiVE	**SmallWiki**	**DelfSTof**
Code size (classes/methods)	148/2118	108/1219	192/747
Number of views	41	15	18
Number of constraints	23	12	11
Execution time	48 sec	5 sec	7 sec

Table 6.1: Overview of the case studies and the execution time of the test suites.

documentation consistent with the source code of that system. While the obvious advantage of this approach is that we support the consistent implementation of structural source-code regularities in the source code of a system, and the integration with the testing process results in that discrepancies between the regularities and the implementation are discovered early on during the development process, this is not the only application of our approach. Since intensional views and constraints created using our tool suite are first-class entities, other software engineering tools can exploit them. As such, while the developer needs to invest an initial cost in order to document the structural source-code regularities using IntensiVE, this documentation can serve multiple purposes.

As we already mentioned earlier, using intensional views and constraints in order to express the regularities governing a system results in the creation of explicit, first-class documentation of these regularities. The intensional views defined over a system align with a concept in the source code of a system, the constraints imposed on these intensional views each document a certain regularity. As such, the documentation created using our approach provides a kind of "conceptual model" of the different concepts in the source code, along with the different conventions and dependencies that govern these concepts. Since our documentation is incorporated in the development environment, and our tool suite provides means to browse the source code entities belonging to this conceptual model, it can serve as a means to make a developer aware of the different concepts and regularities governing a system and provides an initial point-of-access in order to navigate the source code based on this conceptual model.

In Chapter 7 we will discuss another application of the documentation created using intensional views and constraints. In particular, we will show that the problem of fragile pointcuts, one of the open issues in aspect-oriented programming, is caused by the fact that aspect developers heavily rely on how the source code of the base system which aspects are defined over is structured. We propose to alleviate this problem of fragile pointcuts by no longer expressing aspects directly in terms of how the source code is structured at a given moment in time, but rather propose to express pointcuts in terms of the entities in the conceptual model created using IntensiVE.

Run-time of the test suite

In addition to the discussion above about the costs/benefits of applying our approach to large-scale systems, an important factor that might limit the scalability of our approach is the additional run-time it imposes on the test suite of an application. Especially since we promote the frequent verification of the intensional views and constraints that are defined on a system, it is important that the time necessary to verify this set of views and constraints does not hamper the development process by imposing too much of an overhead.

Table 6.1 gives an overview of the code size of each of the three case studies we discussed in this chapter, along with the number of intensional views and constraints defined on each case study, and the time needed[5] to verify all the intensional views/constraints with respect to the source code. The execution time of each of the test suites ranges from a couple of seconds for DelfSTof and SmallWiki up to 48 seconds for the intensional views and constraints we defined on the source code of IntensiVE. Although neither our tool suite nor the SOUL query language is extensively optimized, it does not take excessively long to verify the intensional views and constraints defined on the case studies. As can be seen in the table, the larger the code size and the number of intensional views/constraints, the more time was necessary for verifying consistency of the documented regularities. Due to the computation-intensive nature of some of the regularities we documented in IntensiVE, the relative execution time of this test suite was considerably larger than the time required for the other two case studies. For instance, one such regularity we documented in IntensiVE that has proven to be quite expensive in terms of execution time was the verification that all state changes also trigger the proper change notification. However, by cleverly optimizing the intensions/predicates and by not applying time-consuming analysis of the source code, we were able to limit the execution time of even this largest case study to less than one minute.

It is reasonable to assume that if the size and complexity of the source code of a system increases, so will the number of intensional views and constraints that document the regularities governing that system. One possible way to deal with this increased complexity and to limit the execution time of the test suite, is for a developer to divide a large (sub)system in smaller subsystems and create a separate set of intensional views and constraints for each subsystem.

Evaluation of the methodology

For the three case studies we reported on in this chapter, we rigorously applied the methodology we advanced in Section 4.6. This methodology provides a lightweight set of guidelines for documenting structural source-code regularities using the model of intensional views and the associated tool suite, along with directions how to co-design and co-evolve this documentation with the source code. Furthermore, the methodology advocates a "test-often" philosophy in which the consistency of the documentation is verified at regular times during the development cycle, such that infringements of the documented regularities can be detected as soon as possible. We can conclude the following from the experience we gained performing these three case studies:

- Our iterative refinement describes three possible means of resolving discrepancies between the documented structural source-code regularities and the source code. By inspecting the documentation and the implementation, a developer can choose to either correct the intension/predicate of an intensional view/constraint, to document exceptions to the intensional view or constraint or to correct errors in the source code. This

[5]Benchmarks performed on an Apple Mac Mini with an Intel Core Duo 1.66Ghz processor and 1Gb of RAM.

resolution of discrepancies aligns with evolution conflicts we encountered in all three case studies. For each case study, upon evolution we did not only encounter situations in which the source code of a system did not adhere to the regularities governing this system. We also encountered situations in which either our implementation contained source-code entities that were exceptional cases which the regularities did not apply to. Moreover, we also noticed that during the development of the system, changes in the design or underlying regularities resulted in an update of the documentation. As such, these case studies seem to indicate that co-design and co-evolution of the documentation and source code is necessary and that our approach and methodology are able to support this;

- Another cornerstone of our methodology is the integration of the verification of the documented regularities with the regular unit testing phase. In our case studies, we only integrated this verification in the development cycle of IntensiVE(for the other case studies, we studied the effects of evolution *a posteriori*). However, our experiences with the IntensiVE case study illustrate that the frequent verification of the documented regularities allows detection of discrepancies between the regularities and the source code early on during development.

False positives and false negatives

Applying our approach to the different case studies we reported on in this chapter allowed us to identify a number of interesting evolution conflicts. However, if our tool suite does not report on any discrepancies between the documented regularities and the source code, this does not necessarily imply that the different regularities governing a system are correctly adhered to.

For example, using SOUL as a language to express the predicate of an intensional constraint, we approximate the calling relationship between two methods by verifying that the parse tree of the first method contains a call to the message implemented by the second method. While such an approximation can identify methods in which a specific call is lacking, it does not provide any guarantees whether the call is actually performed at run-time (e.g. when it is part of an ‘if’ statement), thus possibly resulting in false negatives.

Similarly, false negatives can also be the result of an intension that is too specific. Although our model can accommodate such overly-specific intensions by explicitly including particular tuples (in case of an exception to the general rule) in the extension of an intensional view, our approach does not guarantee that the correct set of tuples is captured in general. Consequently, the verification of an intensional constraint imposed on such an overly-specific intensional view can result in that certain discrepancies between the regularity and the source code are omitted.

Our model of intensional views provides one construct which aids in dealing with such false negatives, namely alternative views. When multiple intensions have been specified for the same intensional view, verifying extensional consistency of these alternatives can reveal certain tuples which belong to one of the intensions but not to all of the other intensions of the same view. If one of these intensions is too specific and omits a particular tuple which is captured by any of the other intensions, this false negative will be reported by our tool suite. However, this mechanism does not provide any certainties: in the event that all intensions are too specific and the same tuple is missed by all alternatives, our tool suite is unable to report this false negative.

Analogously, our approach can report false positives: tuples which are identified as being discrepancies between the documented regularities and the source code, but which are in practice not. However, this forms less of a problem than false negatives since these false positives do not result in inconsistencies in the implementation and they can be explicitly documented as exceptions to an intensional constraint.

6.5 Conclusions

This chapter presented the experiences we gained performing three case studies in which we applied our approach, methodology and tool suite for documenting the set of structural source-code regularities underlying the implementation of IntensiVE, SmallWiki and DelfSTof. We integrated this documentation in the development process (the IntensiVE case study), or applied it to subsequent versions of the case study (SmallWiki, DelfSTof). As we have demonstrated, this allowed us to deal with both evolution of the source code as well as of the regularities that govern the system. We were able to detect discrepancies between the documentation and the source code and supported the resolution of these discrepancies by either updating the intensional views/constraints and/or the source code. This co-design and co-evolution of the documentation and implementation resulted in that the causal link between both artifacts remained valid during development and upon evolution of the system.

As we have illustrated, the set of intensional views defined over a system provides documentation for some of the concepts that are prevalent in the source code. For instance, in the SmallWiki case study we created a number of intensional views that grouped the source code entities implementing wiki page components, actions on wiki pages, and so on. In the next chapter, we introduce an application of this first-class documentation. More specifically, we explain the problem of fragile pointcuts and discuss how we can alleviate it by expressing pointcuts in terms of the conceptual model created using intensional views.

Chapter 7

Using model-based pointcuts to tackle the fragile pointcut problem

In the previous chapter we already hinted that first-class documentation for structural source-code regularities that is created using intensional views and constraints is general enough to support other software engineering considerations. In this chapter, we illustrate how the integration of our approach with an aspect-oriented programming language can be used to tackle the fragile pointcut problem, one of the important open evolution problems in research on aspect-orientation. The underlying cause for this fragile pointcut pointcut is that aspect developers often implicitly rely on different structural source-code regularities. If these regularities are not systematically adhered to, this may lead to erratic behavior of the aspects when the software evolves, which is to be avoided.

This chapter is structured as follows:

- In Section 7.1 we give a brief overview of aspect-oriented programming (AOP). In particular, we take a look at the problem of crosscutting concerns and illustrate by means of a number of examples how AOP addresses this problem;

- In Section 7.2 we introduce the fragile pointcut problem. We study the underlying reasons for the fragility of pointcuts using illustrative examples and discuss how the fragile pointcut problem relates to the lack of support for verification of structural source-code regularities;

- We present our approach – model-based pointcuts – in Section 7.3. We provide one concrete instantiation of this approach that uses the formalism of intensional views and constraints;

- To illustrate how model-based pointcuts tackles the fragile pointcut problem, we discuss a small aspect-oriented extension of the SmallWiki system. In Section 7.4, we provide an assessment of the impact of evolution on these aspects and show how our approach can support this evolution;

- In Section 7.5 we give a short overview of other approaches that aim at alleviating the fragile pointcut problem and discuss how these approach relate to model-based pointcuts.

The work we discuss in this chapter is based on our earlier work on model-based pointcuts, as reported in [KMBG06] and [BKG$^+$06].

7.1 Aspect-oriented programming

We start by introducing some common terminology from the domain of aspect-oriented programming.

7.1.1 Modularization of crosscutting concerns

The goal of aspect-oriented programming is to provide an advanced modularization mechanism to separate the core functionality of a software system from system-wide concerns that cut across the implementation of this core functionality. When designing a piece of software, one of the driving forces is to obtain a modularization of the software into a number of components such that there exists a minimum of overlap of functionality between these components [Par72]. In other words, the idea is to modularize the system in such a manner that each module implements a single concern. The intent of this separation of concerns in the implementation of a system is to obtain software that is easier to develop and maintain. Since a developer working on the implementation of one concern is not hassled with the implementation of other concerns in the system, the complexity of the development process can be reduced.

However, using the modularization mechanisms offered by classic programming languages – such as for example procedures and classes – the modularization of a system is performed in terms of a single functional decomposition of the system. The modularization schemes offered by these programming languages thus suffer from the "tyranny of the dominant decomposition" [TOHS99b], which states that a software system can only be modularized in one decomposition at one time. Consequently, in a sufficiently large system there can be certain concerns that do not align with this dominant decomposition. The implementation of these so-called crosscutting concerns is spread throughout multiple modules. Since a single module then contains source code belonging to multiple concerns, the maintainability and evolvability of this module is hampered. Typical examples of crosscutting concerns are logging, synchronization, transaction management, exception handling, and so on.

Crosscutting concerns are characterized by high degrees of *scattering*, meaning that the extent of their implementation encompasses multiple modules, and *tangling* indicating that the source code implementing such crosscutting concerns is intertwined with the source code that implements a part of the core functionality of a system.

7.1.2 Aspect-oriented programming

Aspect-oriented programming (AOP) [KLM$^+$97, KM05b] focusses on extending the modularization capabilities of existing programming paradigms with novel language constructs

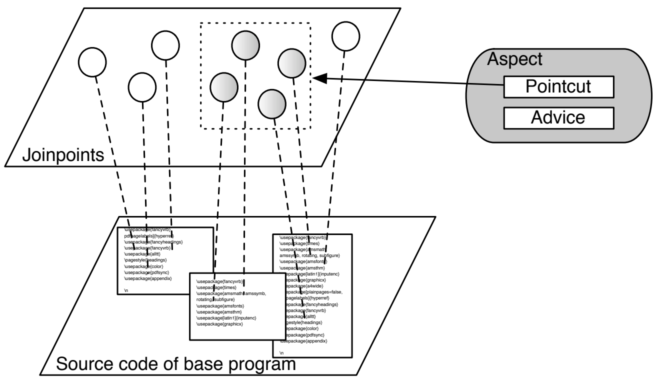


Figure 7.1: Overview of the concepts from aspect-oriented programming

that allow a developer to modularize crosscutting concerns cleanly. To this end, the concept of an *aspect* is introduced: an abstraction mechanism that modularizes the implementation of a crosscutting concern. Although a large number of aspect-oriented languages exists, most of these languages share the same set of basic concepts. These concepts are illustrated in Figure 7.1.

One of the characteristics of aspect-oriented programming languages is that they employ an *implicit* calling mechanism, in contrast to traditional programming languages where the crosscutting behavior is explicitly invoked from within the *base code* of the system. This entails that at certain events during the execution of the system (dubbed *joinpoints*) the crosscutting behavior implemented by the aspect will be executed. As such, the source code of the base program – the part of the system implementing the core functionality – is unaware of the different aspects that are invoked at runtime. This property is called the 'obliviousness' property of aspects with respect to the base program.

To select this set of joinpoints at which a certain aspect is applicable, a *pointcut* is used. A pointcut is a query that reasons about the joinpoints present in a system and selects those joinpoints at which the aspect needs to intervene. For instance, if a developer wishes to execute a piece of crosscutting behavior each time a particular method gets executed, the pointcut must select the joinpoints in the system that correspond to the invocation of that method. The *advice* of an aspect contains the actual implementation of the crosscutting behaviour. This advice will be invoked at the joinpoints selected by the pointcut during the execution of the system.

While it is not our goal to give a complete overview of aspect-oriented programming languages, we illustrate the concepts discussed above by means of some examples. In particular, we briefly discuss the AspectJ and CARMA aspect languages, as we will encounter examples expressed in these two languages throughout this chapter. A first example of an aspect can be found in Figure 7.2. This aspect is implemented in the AspectJ [Lad03, KHH$^+$01] language. AspectJ is an aspect-oriented extension to Java that offers a pointcut language consisting of a number of composable primitive pointcuts which capture low-level events in the execution of a program such as message sends, assignments to variables, and so on. The advice code of an

```
1 pointcut setXY(FigureElement fe, int x, int y):
2    call(void FigureElement.setXY(int, int))
3    && target(fe)
4    && args(x, y);
5
6 after(FigureElement fe, int x, int y): setXY(fe, x, y)
7 {
8   System.out.println(fe + " moved to (" + x + ", " + y + ").");
9 }
```

Figure 7.2: An example of an AspectJ aspect

AspectJ aspect is expressed using regular Java program code.

The aspect in Figure 7.2 implements simple logging facilities in a graphical editor. Each time a figure is moved using the editor, we wish to write a log entry to the screen stating that a certain figure has moved to new coordinates `x` and `y`. Lines 1 – 4 in the figure contain the pointcut of this logging aspect. In AspectJ, pointcuts can be assigned a name (in this case the pointcut is named `setXY`). This pointcut captures all call-joinpoints associated with a call to the method `setXY(int,int)` implemented in the class `FigureElement`. Pointcuts are also provided with a means to pass contextual information to the advice. In our example, the `setXY` pointcut associates `fe` with the target of the call-joinpoint (line 4), i.e. the actual `FigureElement` that is being moved. Furthermore, the pointcut also passes the new coordinates `x` and `y`, bound (line 3) to the arguments of the call to `setXY(int,int)`, as contextual information to the advice. The advice (lines 6–9) of the aspect will be executed *after*[1] a joinpoint captured by the `setXY` pointcut has occurred. The body of the advice simply logs that figure `fe` has moved to coordinates x and y.

```
after ?joinpoint matching
  reception(?joinpoint, setXY, ?args),
  inObject(?joinpoint,?obj),
  objectClass(?obj,[FigureElement]),
  equals(?args,<?x,?y>)
do
  Transcript show: 'Figure ', ?obj,
                    ' moved to coordinates: ', ?x,',',?y.
```

Figure 7.3: An example of a CARMA aspect

Figure 7.3 shows an example of an aspect implemented in the CARMA [GB03] language. CARMA is an aspect language for Smalltalk that uses the logic programming paradigm in order to express pointcuts. CARMA is based on the SOUL logic language and extends this language with a number of predicates that allow reasoning over the joinpoints in a system. This use of a logic pointcut language which offers unification, backtracking and facilities to

[1]Similarly, AspectJ supports the execution of advice *before* a certain event in the system or *around* an event, thus replacing the behavior of the base program.

reason over an entire reification of a program has proved to provide an expressive means to create generic, reusable pointcuts [GB03]. The aspect in Figure 7.3 implements the CARMA variant of the AspectJ pointcut we discussed above. The pointcut comprises lines 2 – 5 of the aspect shown in Figure 7.3. Line 2 of the pointcut results in a number of bindings of the variable `?joinpoint` that are a reception joinpoint of the message `setXY`. These joinpoints are further restricted (lines 3 and 4) to those that occur in an instance of the class `FigureElement`. Finally, line 5 binds the first and second argument of the call to `setXY` to the variables `?x` and `?y` respectively. After a joinpoint that matches the pointcut is encountered during the execution of a program, the advice of the aspect (lines 7 and 8) is executed. Similar to the advice of the AspectJ aspect, this advice displays some logging information on the `Transcript`. In order to pass contextual information from the pointcut to the advice, the logic variables from the pointcut can be used in the advice.

7.2 The fragile pointcut problem

7.2.1 Definition

It has been observed in the literature [KS04, SG05, KM05b, SGS$^+$05] that the semantics of a pointcut can change unexpectedly when the structure of the base program is altered. Even if no changes are made to the actual pointcut, the evolution of the base program can result in the set of joinpoints in the system changing. As a result, it can happen that the pointcut no longer captures the expected set of joinpoints, thus resulting in erratic behavior.
This problem has been dubbed the fragile pointcut problem (FPP). We define it as [KMBG06]:

> The **fragile pointcut problem** occurs in aspect-oriented systems when pointcuts unintentionally capture or miss particular joinpoints as a consequence of their fragility with respect to seemingly safe modifications to the base program.

Consequently, one cannot assess whether changes to the base program are safe by solely studying this base program. Pointcuts that refer to joinpoints originating from the base program must as well be inspected in order to ensure that changes to the base program do not affect the pointcut.

We illustrate the fragile pointcut problem in Figure 7.4. The left-hand side of the Figure represents a software system on which a single aspect has been defined. The source code of the system gives rise – at execution time – to a number of joinpoints. The pointcut of the aspect selects, by referring to a structural or behavioral property of the source code, four of these joinpoints where the aspect will intervene.

Upon evolution of the base program (right-hand side of Figure 7.4), the source code of the system changes. Consequently, the set of joinpoints that is associated with the entities in the source code of the base program also changes. Since the pointcut refers to a structural or behavioral property of the base program, it is possible that these changes in the base program result in a pointcut that no longer captures the correct set of joinpoints. We encounter two kinds of such *joinpoint mismatches*:

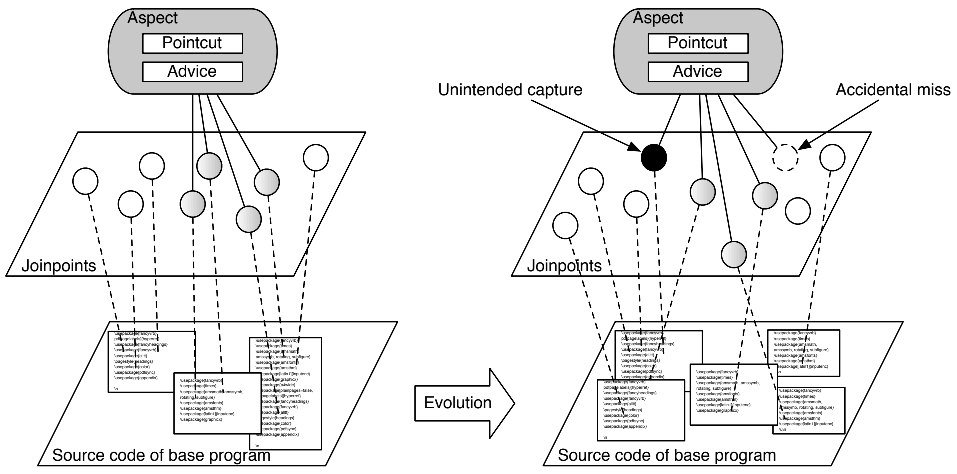


Figure 7.4: Illustration of the fragile pointcut problem

- If changes in the base program result in the pointcut accidentally selecting a wrong joinpoint related to those source-code entities, we encounter an *unintended joinpoint capture* (e.g. the joinpoint indicated in black in Figure 7.4);
- Conversely, changes in the source-code of the base system can also result in certain joinpoints that were previously selected by the pointcut no longer being selected, even though they were supposed to be. We denote these cases as an *accidental joinpoint miss* (the dashed joinpoint in the Figure).

Note that a pointcut can also break when the source-code entities which it explicitly refers to are no longer present in the system. For instance, both pointcuts we discussed in Section 7.1.2 explicitly rely on the presence of a class `FigureElement`. If this class is removed, or if its name is altered, the pointcut will also fail. However, similar to a compiler error, such instances of fragile pointcuts can easily be detected when evaluating the pointcut description of an aspect.

7.2.2 Examples of fragile pointcuts

To understand the fundamental causes of the fragile pointcut problem, we analyze how different "styles" of defining pointcuts cause the pointcut to be fragile with respect to changes in the base program. To this end, we introduce a simple example, namely the implementation of a buffer in Java. For this buffer, we demonstrate how we can implement the pointcut of a synchronization aspect by applying each of the different pointcut styles. We then provide an example of how changes in the source code of the base program impact the pointcut.

A part of the implementation of this Buffer class is shown in Figure 7.5. The buffer contains two fields, namely a field representing the `contents` of the buffer and a field `ind` that contains the current index into the buffer. Furthermore, the buffer implements two methods for accessing the contents of the buffer: a method `get()` that returns the object in the buffer

```
class Buffer {
  private Object contents[];
  private Int ind = 0;

  public Object get()
   {
     ...
     return contents[ind];
   };
  public void set(Object obj)
   {
     ...
     contents[ind] := obj;
   };

}
```

Figure 7.5: Part of the implementation of a simple Buffer object

at the current index and a method `set(Object)` that places an object in the buffer. The pointcut of the synchronization aspect we define on this buffer implementation must capture these `get()` and `set(Object)` methods. In the following sections, we demonstrate such a pointcut expressed using each of the different styles of pointcuts (i.e. enumeration-based, pattern-based, structural-property based and behavioral-property based).

Enumeration-based pointcuts

Enumeration-based pointcuts are the simplest way of defining the set of joinpoints at which an aspect needs to intervene. Such a pointcut consists of an *enumeration* of the *exact set* of joinpoints which the crosscutting behavior needs to be invoked at. For instance, for the synchronization aspect from our buffer example, we can specify an enumeration-based pointcut to capture all "accessors" of the buffer by directly referring to the method signatures of these methods. We can define this pointcut in AspectJ as:

```
pointcut accessors()
  call(void Buffer.set(Object)) ||
  call(Object Buffer.get());
```

Upon evolution of the base program, this pointcut is unlikely to result in unintended joinpoint captures. Since the pointcut explicitly refers to the methods at which the aspect needs to intervene, it is not possible that this pointcut captures additional joinpoints that result from changes in the base program.

However, the enumeration-based pointcut is brittle with respect to accidental misses. If the developer of the buffer for instance adds additional accessors such as `setAll` or `getAll` that respectively add or retrieve a collection of objects in the buffer, these newly added methods do not get captured by the pointcut. A developer thus needs to update the pointcut manually in order to take these accessors into account. If the pointcut is not updated, the synchronization aspect would incorrectly miss these `setAll` and `getAll` methods.

The pointcut is equally brittle if the `Buffer` class or any of the getter and setter methods are removed from the system. In this case, the pointcut refers to source-code entities that no longer exist thus becoming ill-defined.

Pattern-based pointcuts

The second way of defining pointcuts we discuss is by means of a pattern. Rather than referring directly to the signature of a method, a pattern-based pointcut makes use of wildcards for the method signatures. For instance, for the getter methods in the implementation of the buffer, we see that the names of these methods start with the prefix "get-" and "set-". We can define an AspectJ pointcut based on this naming scheme as:

```
pointcut accessors()
  call(* set*(..))) ||
  call(* get*(..));
```

At first glance this pointcut is less fragile than the enumeration-based version we discussed above. The addition of methods like `getAll` and `setAll` will automatically and correctly be captured by this pattern-based pointcut. As long as the accessor methods respect the naming convention that they should be prefixed "get-" or "set-", this pointcut will correctly capture them. However, this does not mean that pattern-based pointcuts are not fragile. Suppose for instance that a developer adds a method `settings()` to the implementation of the Buffer class. Since this method also starts with the prefix "set-" but does not represent an accessor method it will be incorrectly captured by the synchronization aspect, resulting in an unintended joinpoint capture.

Structural-property based pointcuts

More expressive pointcut languages, such as the CARMA language we briefly discussed above, are not restricted to expressing pointcuts in terms of simple run-time events such as message sends, field access and so on, but also provide facilities for expressing pointcuts based on fine-grained structural properties in the source-code entities of the base program. For instance, instead of relying on naming conventions in order to capture the accessor methods in the `Buffer` class, we can base a pointcut on the structural pattern that is used to implement these accessors. The getter methods on the buffer are all characterized by the fact that they return the value of an instance variable; the setter methods all perform an assignment to an instance variable. Using CARMA, we can express these structural patterns using two pointcuts.

```
call(?joinpoint,?method),
methodInClass(?method,?class),
instanceVariableInClass(?iv,?class),
methodWithReturn(?method,return(?iv));
```

The above pointcut captures all calls to getter methods. It selects all call-joinpoints of a `?method` (line 1) such that the `?class` which `?method` is implemented in contains an instance variable `?iv` (lines 2 and 3) and `?method` contains a return statement that returns the value of the instance variable[2].

Likewise, we can implement a pointcut that captures all setter methods as:

```
call(?joinpoint,?method),
methodInClass(?method,?class),
instanceVariableInClass(?iv,?class),
methodWithAssignment(?method,assign(?iv,?value));
```

This pointcut is similar to the pointcut that captures all getter methods. It selects all method calls to a method that perform an assignment to an instance variable.

The above structural-property based pointcuts are not fragile with respect to some of the changes in the base program we discussed above. For instance, the `getAll` and `setAll` methods would correctly be captured by the above pointcuts. The method `settings`, which resulted in the fragility of the pattern-based pointcut, will not be captured by these two pointcuts. As long as the implementation of accessors follows one of the two structural patterns we encoded in the above CARMA pointcuts, the pointcuts will remain correct upon evolution of the base program.

However, suppose we replace the implementation of `get()` in the `Buffer` class by the following method:

```
public Object Buffer.get()
{
  Object temp := contents[ind];
  ... //some operations
  return temp;
}
```

The above `get()` method does not return the value of the instance variable, but rather stores it in a temporary variable `temp`. After performing some other operations which do not affect the value of `temp`, the `get()` method returns the value of the temporary variable. While technically speaking the above method is an accessor method, it deviates from the structural pattern we discussed above. Consequently, the structural-property based pointcut will not capture this alternative implementation of a getter method.

Behavioral-property based pointcuts

Behavioral-property based pointcuts are expressed in terms of the execution history of an application or the values that appear at run-time in the system. One such construct that allows for the specification of pointcuts in terms of a behavioral property is `cflow`. The `cflow` predicate can be used to restrict joinpoints captured by a pointcut to those that lie in the control flow of a specific set of joinpoints. For instance, suppose we want to optimize the implementation of the synchronization aspect such that the synchronization will not be executed again

[2]The `methodWithReturn` predicate considers the indexing into arrays for variable accesses.

if the buffer is already synchronized. We can implement this optimization using the following AspectJ pointcut:

```
pointcut optimizedAccessors():
  accessors() &&
  !cflow(accessors());
```

This pointcut captures all joinpoints that are associated with an accessor method and that are not in the control flow of another accessor method. The pointcut thus selects all joinpoints associated with top-level invocations of accessor methods. While this pointcut does not refer directly to the source code of the system, it still is fragile upon evolution of the source code. Since the `optimizedAccessors()` pointcut refers to the `accessors()` pointcut, and this latter pointcut is fragile this results in that the `optimizedAccessors()` pointcut is fragile too.

Another cause for the fragility of behavioral-property based pointcuts is that, although they are defined in terms of the run-time values of a system rather than the actual structure of that system, they often need to refer explicitly to the names of source-code entities in order to retrieve these run-time values.

7.2.3 Analysis of pointcut fragility

The fragility of the pointcuts we discussed above is caused by the tight coupling that exists between the pointcut definition and the source code of the base program. A first contributor to this tight coupling is the fact that in a pointcut definition pointcut developers often explicitly refer to the different symbols of the base program: method signatures, variable names, and so on. For example, enumeration-based pointcuts directly refer to the signature of the methods at which the aspect needs to intervene. The slightest change in the signature of a method that ought to be captured thus results in pointcut fragility.

Another contributor to the fragile pointcut problem is that pointcut developers often make assumptions about how a certain concept of the base program manifests itself in the source code. For instance, the structural-property based pointcuts we discussed above rely on the assumption that the implementation of getter and setter methods in the implementation of the `Buffer` class all follow a similar implementation scheme. Since this pointcut does not directly refer to entities in the source code of the base program, it is more flexible with respect to evolution. For example, changes to the method signatures of the getters and setters do not influence the pointcut. However, if the base code developer – when evolving the system – deviates from the implementation scheme on which the pointcut developer relied, the pointcut becomes fragile.

As a means to prevent accidental misses and unintended captures, pointcut developers rely on the presence of certain 'design rules' in the source code of a base program. Since a pointcut does not have direct access to certain concepts that are prevalent in a system, it relies on the structure of the implementation of that concept in the source code. For example, the synchronization aspect on the `Buffer` class we discussed above is defined in terms of the concepts of 'getters' and 'setters'. However, these concepts are not explicit, first-class entities in the pointcut language. Consequently, a developer tries to capture these concepts by

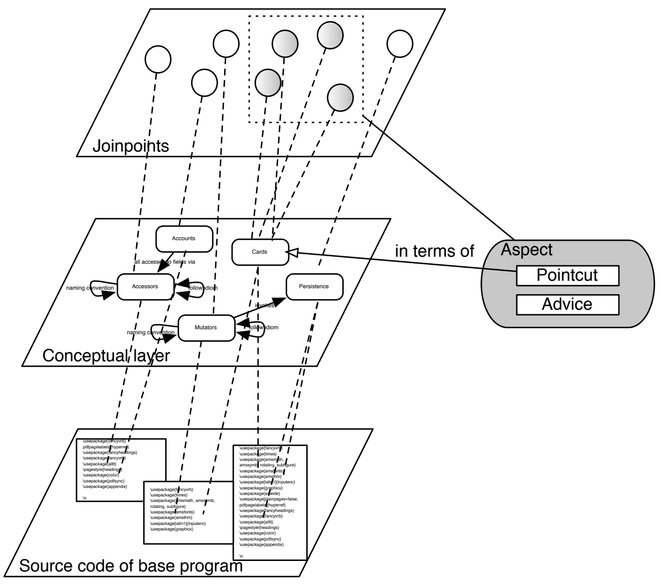


Figure 7.6: Illustration of model-based pointcuts

relying on how these concepts are implemented. By directly referring to method signatures or by relying on naming conventions, implementation patterns and so on the pointcut developer will try to capture the 'getters' and 'setters' in the source code.

As long as base code developers adhere to the design rules which the pointcut relies on, the pointcut will capture the correct set of joinpoints. However, since the concept captured by a pointcut as well as these design rules are only *implicitly* available in the source code, seemingly safe modifications to the base program can result in unexpected and erratic behavior of the aspects imposed on the base program.

7.3 Model-based pointcuts

In this section we present *model-based pointcuts*, the technique we propose to alleviate the fragile pointcut problem. We start by explaining the different concepts underlying model-based pointcuts; afterwards, we introduce a concrete instantiation of model-based pointcuts based on our formalism of intensional views and constraints.

7.3.1 Definition

Model-based pointcuts provide a new pointcut definition mechanism that reduces the coupling between the pointcut definition and the base program. This mechanism aims at rendering both the concepts that are implicitly captured by the pointcut as well as the design rules that govern these concepts explicit to the developer. A schematic illustration of model-based pointcuts can be found in Figure 7.6.

Rather than defining a pointcut by referring to how the base program is structured at a given point in time, model-based pointcuts are defined in terms of a *conceptual model* of the base program. This conceptual model provides a representation of the different conceptual entities that exist in the base program and makes them accessible to the pointcut language. This way, the joinpoints captured by the pointcut are not characterized by a structural property of the base program but rather by a *conceptual* property. At the same time, this conceptual model contains verifiable constraints on concepts that express the different design rules (i.e. regularities) that govern these concepts, as well as how these concepts are mapped to the actual source-code entities.

Let us illustrate this mechanism by means of an example. A conceptual model of the `Buffer` implementation we discussed above would contain a concept “AccessorMethods” representing the accessor methods (i.e. getters and setters). Indeed, the notion of accessing data is an important concept both for the buffer data structure and for the synchronization aspect that we wish to apply to that data structure. Using model-based pointcuts, the pointcut of the synchronization aspect we defined over the buffer is implemented as:

```
pointcut accessors():
  call(* *(..)) &&
  classifiedAs(AccessorMethods);
```

The above AspectJ-like pointcut captures all call-joinpoints to methods classified as accessor methods. To this end, the pointcut directly refers the conceptual entity *AccessorMethods* by means of the special joinpoint predicate `classifiedAs`. Notice that this pointcut no longer needs to rely on the structural regularities that govern the accessor methods in order to capture this concept in the pointcut definition.

Such model-based pointcuts alleviate the fragile pointcut problem by:

1. Adding an additional layer of indirection between the pointcut definition and the base program. This conceptual layer contains a representation of the various concepts of interest in the base program. Since the pointcuts are defined in terms of this conceptual layer rather than by referring directly to how the source code of the base program is structured, the tight coupling between a pointcut and the base program is removed;

2. While this conceptual layer effectively decouples the pointcut from the base program structure thus reducing the fragility of the pointcut, it reintroduces this fragility at the level of the conceptual layer itself. As long as the different concepts in this layer classify the correct set of source-code entities the pointcut will still intervene at the correct joinpoints upon evolution of the base program. However, if changes in the source code result in the conceptual layer no longer being consistent to the base program, the concepts becomes fragile. In order to deal with this fragility, verifiable constraints are imposed on the different concepts in the conceptual layer to aid in detecting discrepancies between the conceptual layer and the base program. Using this information, a developer can resolve inconsistencies between the conceptual layer and the source code.

In a sense, model-based pointcuts impose a contract between pointcut developers and base program developers:

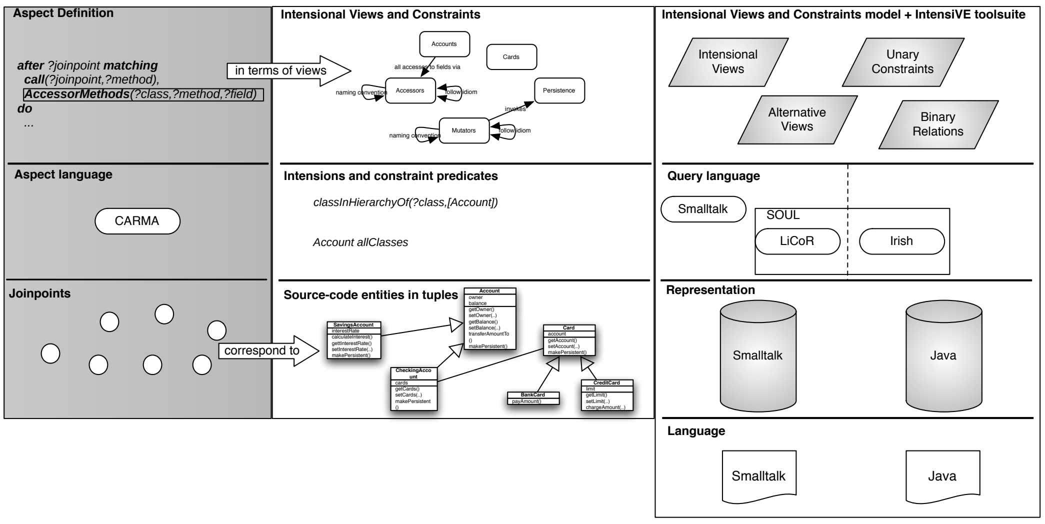


Figure 7.7: An overview of how view-based pointcuts fit into IntensiVE.

- For the pointcut to function correctly, the pointcut developer *requires* that the conceptual layer provides access to certain concepts of interest in the base program. Furthermore, the developer expects upon evolution these concepts to remain consistent with respect to the source code of the base program;

- The base program developer *provides* a number of concepts to the pointcut developer by means of the conceptual layer. It is the responsibility of the base program developer to ensure that the concepts in the conceptual layer remain consistent with respect to the source code of the base program. To this end, the base program developer imposes verifiable constraints on the conceptual layer.

7.3.2 Concrete instantiation: View-based pointcuts

Now that we have explained the mechanism of model-based pointcuts, we discuss one instantiation of this mechanism, namely *view-based pointcuts*. View-based pointcuts present a particular instantiation of model-based pointcuts in which we use our formalism of intensional views and constraints as a means to create a conceptual layer. Figure 7.7 illustrates how view-based pointcuts fit into the implementation of IntensiVE. This figure is an extended version of Figure 4.1 from Chapter 4. The left-most column illustrates the extension of the tool suite with support for model-based pointcuts.

While the different concepts in the conceptual layer correspond to the set of intensional views defined over a system, the constraints imposed on these concepts align with the use of alternative views, unary constraints and binary intensional relations. We extended the CARMA logic pointcut language with a means to express pointcut definitions in terms of the intensional views that are defined on a system. Consequently, the joinpoints associated with the source-code entities belonging to the extension of the intensional views are, via the conceptual layer, made accessible to a view-based pointcut.

For example, the synchronization aspect on the implementation of the `Buffer` class can be implemented using the following view-based, CARMA aspect:

```
after ?joinpoint matching
  call(?joinpoint,?method),
  AccessorMethods(?class,?method,?field)
do
  .. "perform synchronisation of the buffer"
```

First of all, notice that in the above pointcut we do not use a `classifiedAs` predicate. Since the CARMA language is an extension of SOUL, the CARMA pointcut mechanism can use the first-class representation of intensional view we discussed in Section 4.2.2 of Chapter 4. As a result, the condition `AccessorMethods(?class,?method,?field)` uses this first-class representation to retrieve all methods that are classified by the *AccessorMethods* intensional view. The above pointcut will select all call-joinpoints that are associated with such an accessor method.

The integration of intensional views in a pointcut language makes it possible for a base program developer to extend the pointcut language with certain concepts from the domain of the base program, that were previously not accessible as first-class entities from within the pointcut language. Furthermore, since the conceptual model also imposes a number of constraints on the intensional views, the IntensiVE tool suite can be employed to aid the developer in keeping the conceptual model in sync with the implementation.

7.4 Model-based pointcuts in practice

We performed a small case study in order to illustrate how model-based pointcuts alleviate the fragile pointcut problem. In particular, we defined two aspects on the implementation of SmallWiki, one of the case studies discussed in Chapter 6. We first defined these aspects on version 1.54 of SmallWiki, using both a traditional pointcut mechanism and model-based pointcuts. Afterwards, we assessed the impact of base program evolution on both types of pointcuts by reapplying the same pointcuts to version 1.304 of SmallWiki, which is a release of almost one year after version 1.54.

7.4.1 Aspects on SmallWiki

For this case study, we extended the implementation of version 1.54 of SmallWiki with two aspects:

1. *Action logging*: this aspect provides basic logging facilities for the different actions that occur in the wiki system. For example, each time a page is opened, saved, edited and so on, this aspect will write an entry to the Transcript;

2. *Italics output*: the second aspect we define on SmallWiki changes the rendering of the wiki documents such that all text is rendered in italics rather than in the regular font.

Although both aspects provide a rather trivial extension to the functionality of SmallWiki, these simple aspects do allow us to illustrate the impact of base program evolution on both the traditional pointcut and the model-based pointcut.

The *Action logging* aspect

We first discuss the implementation of the pointcut for the *Action logging* aspect using both a traditional pointcut description and a model-based pointcut in the CARMA aspect language. Since the emphasis of this case study lies in studying how evolution of the base program impacts the pointcut definition, we omit the advice code from these examples.

Traditional pointcut for *Action logging* The pointcut for the *Action logging* aspect should capture all call joinpoints to methods that perform an action on a Wiki page. We rely on how the concept of an action is implemented in SmallWiki to select these joinpoints. As we already mentioned in Chapter 6 the actions on a wiki document are characterized by two regularities: all wiki actions are implemented by a method prefixed by “execute-” and are classified in a protocol `action`. Based on these regularities, we can express two variants for the *Action logging* aspect’s pointcut:

```
1  classInNamespace(?class,[SmallWiki]),
2  methodWithNameInClass(?method,?selector,?class),
3  ['execute*' match: ?selector],
4  call(?joinpoint,?method)
```

The above pointcut is based on the regularity that all actions in SmallWiki are implemented by a method that is prefixed by “execute-”. This pointcut first selects all classes in the namespace `SmallWiki` (line 1); in lines 2 and 3 all methods are selected that are implemented by those classes and that start with the prefix “execute-”; line 4 captures all call-joinpoints associated with these methods.

Alternatively, we can define a pointcut that is based on the regularity that all methods implementing an action are classified in the protocol `action`:

```
1  classInNamespace(?class,[SmallWiki]),
2  methodInClass(?method,?class),
3  methodInProtocol(?method,action),
4  call(?joinpoint,?method)
```

Model-based pointcut for *Action logging* We also provide the model-based variant of the pointcut for the *Action logging* aspect. As the conceptual model in terms of which the pointcut is expressed, we use the set of intensional views and constraints we defined on the implementation of SmallWiki in Section 6.2 of Chapter 6.

We use the intensional view *Wiki Actions* discussed in Chapter 6 to define the pointcut for the *Action logging* aspect. This intensional view has two attributes, namely ‘class’ and ‘method’ and consists of two alternative views: one alternative contains all methods in SmallWiki that start with the prefix “execute-”; the other alternative contains all methods in the protocol `action`. Note that these alternative views align with the structural source-code regularities we discussed above as being characteristic for the implementation of actions in SmallWiki. Consequently, our definition of the traditional pointcuts relies on that these regularities are respected in the implementation of SmallWiki. Based on the *Wiki Actions* intensional view, we define the pointcut for the *Action logging* aspect as:

```
WikiActions(?class,?method),
call(?joinpoint,?method)
```

This pointcut captures all call-joinpoints that are associated with a method that belongs to the *Wiki Actions* intensional view.

The *Italics output* aspect

Traditional pointcut for *Italics output* For the *Italics output* aspect, we want to alter the outputting of Wiki pages. In SmallWiki, a document is rendered by means of a visitor object that traverses the document and generates suitable output (HTML, Latex, . . .) for each component of the page. As such, for the *Italics output* aspect to function properly, we want the aspect to intercept all calls originating from a page component to a method on a visitor that generates output.

For the *Italics output* aspect, we define the following pointcut:

```
1  classInNamespace(?class,[SmallWiki]),
2  classInHierarchyOf(?class, [PageComponent]),
3  classInHierarchyOf(?output, [OutputVisitor]),
4  methodWithNameInClass(?method,?selector,?output),
5  withinClass(?joinpoint,?class),
6  send(?joinpoint,?selector)
```

Line 1 selects all classes in the `SmallWiki` namespace; line 2 captures all page components (i.e. all classes in the hierarchy of `PageComponent`). Lines 3 and 4 select all methods that generate output (all methods implemented on a class in the hierarchy of `OutputVisitor`). Finally, lines 5 and 6 select all send joinpoints of a message implemented by an output generator that is invoked from within a page component.

Model-based pointcut for *Italics output* The model-based version of the pointcut for the *Italics output* aspect relies on two concepts from the base code of SmallWiki, namely the implementation of page components and output generation. For each of these concepts, we already defined an intensional view in Chapter 6 that groups the source-code entities related to each concept. In particular, we defined an intensional view *Page components* with a single attribute ‘class’ that groups all the components of a wiki document. We also created an intensional view *Output Visitors* with as attributes ‘class’ and ‘method’ containing all methods that generate output for a certain page component. We can thus define the view-based pointcut for the *Italics output* aspect as:

```
1  Pagecomponents(?component),
2  OutputVisitors(?visitor,?method),
3   methodWithName(?method,?selector),
4   send(?joinpoint,?selector,?args),
5   withinClass(?joinpoint,?component)
```

In analog to the traditional pointcut we defined above, this pointcut selects all send joinpoints (line 4) that originate from within a page component (lines 1 and 5) of a message that implements the generation of output (lines 2 and 3).

7.4.2 Impact of evolution on the aspects

In this section we assess the impact of evolution of the base code of SmallWiki on the different pointcuts we defined above. To this end, we reapply the pointcuts to version 1.304 of SmallWiki.

Impact of evolution on the *Action logging* aspect

Traditional pointcut for *Action logging* Changes to the implementation of SmallWiki resulted in that version 1.304 implemented a considerably larger set of actions on wiki page components. For instance, the system was extended with a login mechanism, search functionality, and so on. While most of these actions were correctly captured by the traditional pointcut, we did encounter some methods that caused this pointcut to select the wrong set of joinpoints.

We defined two variants of the pointcut that captures all executions of actions in SmallWiki: one pointcut defined in terms of the convention that all actions start with the prefix "execute-"; the other pointcut is based on the regularity that actions are classified in the protocol `action`. The following changes in the implementation resulted in fragility of those pointcuts:

- Two methods named `save` and `authenticate` were added to SmallWiki. While these methods implement a Wiki action and should thus be intercepted by the *Action logging* aspect, they clearly do not start with the prefix "execute-" thus rendering the pointcut based on this naming convention fragile. However, since both methods are classified in the `action` protocol, they were correctly captured by the other variant of the pointcut we specified;

- The evolution of SmallWiki also resulted in the introduction of the methods `executeSearch` and `executePermission`. Since these two methods are prefixed "execute-", they were captured by the pointcut that relies on this regularity. However, these methods were missed by the second pointcut since they were respectively classified in the protocols `actions` (with an extra 's') and `private`.

Consequently, neither of the two pointcuts we defined for the *Action logging* aspect was able to correctly capture *all* of the Wiki actions. While the pointcut based on the naming convention resulted in that the `save` and `authenticate` methods were accidentally missed, the pointcut based on the `action` protocol missed the `executeSearch` and `executePermission` methods.

Model-based pointcut for *Action logging* We also reapplied the model-based version of the *Action logging* pointcut to version 1.304 of SmallWiki. However, *before* applying the aspect, the conceptual model that was defined over SmallWiki was verified using the IntensiVE tool suite. This allowed us to ensure that the conceptual model was still consistent with the new version of the source code.

As we already discussed in Chapter 6, verification of the *Wiki actions* intensional view – which the view-based pointcut of the *Action logging* aspect is based on – indicated that this view was not extensionally consistent. A closer inspection of the discrepancies reported that the methods `save` and `authenticate` were captured by the second alternative view (stating that all actions must belong to the `action` protocol) but not to the first alternative view (i.e. all methods with as prefix "execute-"). Analogously, it was reported that the methods `executeSearch` and `executePermission` belonged to the first alternative but not to the second.

Since the alternatives of this intensional view correspond to the regularities which each of the traditional pointcuts we defined for the *Action logging* aspect is based on, it is not surprising that, upon verification of the intensional view, exactly those methods that resulted in the fragility of the traditional pointcuts were reported as discrepancies between the intensional view and the implementation of SmallWiki. While the base code developer still needed to update the conceptual layer and the source code of the base program in order to synchronize both artifacts, after having done so the model-based pointcut itself needed no alterations in order to remain valid.

Impact of evolution on the *Italic output* aspect

Traditional pointcut for *Italics output* A number of the page components added to the implementation of version 1.304 of SmallWiki resulted in that the traditional pointcut we defined for the *Italics output* aspect became fragile. The pointcut of the *Italics output* aspect is based on the assumption that for all page components, a corresponding visit method in the hierarchy of `OutputVisitor` is invoked. The aspect must intercept this corresponding method and change the output of the page component into italics. However, certain page components in version 1.304 of SmallWiki (for instance `LinkExternal` and `LinkMailTo`) did not have such a corresponding output method on the `OutputVisitor` class. Instead, for reasons of code reuse, their visit method was implemented on the abstract `Visitor` class. This method on the abstract class then delegated the visit operation to other visit methods (that were implemented on a class in the `OutputVisitor` hierarchy). While the implementation of these newly added page components deviated only subtly from the implementation scheme on which the pointcut for the *Italics Output* aspect relied, the newly added components were unintentionally missed by the pointcut.

Model-based pointcut for *Italics output* The model-based pointcut for the *Italics output* aspect is based on two intensional views: *Output Visitors* and *Page Components*. Upon application of these intensional views to the evolved version of SmallWiki, these intensional views remained extensionally consistent. However, as we already discussed in Chapter 6, the binary intensional relation that expressed that for all page components there should exist a corresponding visit method failed. When analyzing the discrepancies between this constraint and the implementation of SmallWiki, the same set of page components were reported as those that resulted in the fragility of the traditional pointcut for the aspect.

7.4.3 Conclusion

The above experiment demonstrated model-based pointcuts. As we have illustrated these model-based pointcuts did not have to be adapted to reflect the changes in the source code upon evolution of the base program. This is in contrast to the traditional pointcuts that became fragile since the regularities which they relied on were no longer adhered to in the evolved version of SmallWiki. This does not imply at all that model-based pointcuts avoids the problem of pointcut fragility. However, since the pointcuts are defined in terms of a conceptual model rather than in terms of the structure of the base program, the fragility appears at the level of this conceptual model instead of at the level of the pointcuts. The conceptual model contains a number of constraints imposed on the concepts that make the link between the concepts and the source code explicit and verifiable. Consequently, evolution conflicts that result in fragility can be detected and resolved at the level of this conceptual layer, thus preventing the pointcut from becoming brittle.

7.5 Related work

In this section we give a brief overview of other approaches that have been proposed as a means to tackle the fragile pointcut problem and discuss their strengths and weaknesses.

7.5.1 More expressive pointcut languages

A first group of approaches aim to provide a mechanism for defining more robust pointcuts by offering more expressive pointcut languages. Rather than the low-level, restricted set of constructs that are offered by AspectJ, these approaches provide more elaborate means to reason about the source code or the run-time state of a system. We already encountered such a more expressive pointcut language, namely the CARMA language which we also extended as a means to express model-based pointcuts. This pointcut language offers a complete logic programming language with access to a full reification of the abstract syntax tree of the program on which the aspects are defined [GB03]. The Alpha language [OM05] also uses the logic paradigm to define pointcuts. However, this language enhances the expressiveness of the pointcut language by automatically deriving models of the program and by providing predicates to query these models. This way, Alpha makes it possible to define e.g. pointcuts that query the entire state or execution history of a program. Finally, event-based AOP [DFL$^+$05] and JAsCo [VSCD05] allow the definition of so-called stateful aspects: aspects that are triggered by the occurrence of a certain sequence of events during the program's execution.

While these more expressive pointcut languages make it possible to define pointcuts that are more flexible with respect to changes in the base code, they do not make fragile pointcuts disappear. As we have illustrated in Section 7.2.2, these pointcuts still need to refer to a structural or behavioral property of the program to specify the set of joinpoints captured by the pointcut. Although they enable the definition of more generic and reusable pointcuts, they do not aid in making the concepts in the source code which pointcuts depend on explicit in the pointcut language.

7.5.2 Annotation-based pointcuts

Kiczales et al. [KM05c] and Havinga et al. [HNB05] propose expressing pointcuts in terms of explicit annotations in the source code as a means of making pointcuts more robust. While these annotations aid in making the concepts that are implicitly captured by the pointcut explicit by means of an annotation, they shift the fragility of the pointcut to the annotations themselves. Instead of requiring that the base program respects the structural source-code regularities which the pointcut relies on, annotation-based pointcuts require that the base program is annotated consistently. If upon evolution of the system the source code is not correctly annotated, the pointcut based on these annotations becomes fragile. Havinga et al. propose to alleviate this problem by weaving the annotations themselves into the source code using an aspect. However, this results in the fragility of the annotation-based pointcut being shifted to the aspect used to introduce the annotations in the source code.

7.5.3 Pointcut delta analysis

Stoerzer et al. [KS04, SG05] propose the use of pointcut delta analysis for detecting instances of fragile pointcuts. Pointcut delta analysis (using the associated tool PCDiff) compares a static approximation of the set of joinpoint that are captured by each pointcut *before* changes are made to the source code to the set of joinpoints captured *after* the source code has been changed. The developer can then analyze the delta between these sets of joinpoints and identify possible discrepancies.

While this approach is very useful for at finding instances of unintended captures, it is impossible for this technique to find accidental misses that result from the addition or modification of source-code entities. If for example the addition of a new method causes new joinpoints to appear in the system for which an aspect *should* intervene, but the pointcut of that aspect does not capture these new joinpoints, this results in an accidental miss. However, these joinpoints will not be part of the delta of the sets of joinpoints before and after evolution of the base program.

It would be interesting to extend our own tool suite with similar facilities as those offered by PCDiff. Rather than saving the set of joinpoints captured by a pointcut at a given moment in time, and comparing this saved set with the joinpoints captured by the pointcut after the system evolves, we could offer similar functionality for intensional views. Such a mechanism would allow a developer to save the extension of an intensional view at a given moment in time and, upon evolution, compare this saved extension to the current extension of the view.

7.5.4 Explicit pointcut interfaces

Similar to model-based pointcuts, a number of approaches try to alleviate the fragility of pointcuts by introducing an interface in between the pointcuts and the base program. A first such approach is *Open Modules* [Ald05]. Using this approach, a developer of a module must specify the joinpoints that are accessible to aspects that are defined outside of the module. An alternative approach is proposed by Sullivan et al. [SGS$^+$05] by means of *design rules*. These design rules present an interface specification which must be adhered to by base code developers and which must be used by pointcut developers. Once in place, the base code and the pointcuts can be altered independently of each other as long as the design rules remain valid. *Explicit Pointcut Interfaces (XPI)* are an implementation of design rules. An XPI consists of a global set of pointcuts together with some constraints that serve as a means to verify the validity of the pointcuts. These constraints are also implemented using pointcuts (using e.g. the `declare warning` construct of AspectJ). Using XPIs, the set of pointcuts is fixed beforehand, resulting in that the aspect developer losing some flexibility. Conversely, model-based pointcuts do not restrict the set of pointcuts that can be used by aspect developers. Using our approach a base code developer presents the pointcut developer with a conceptual model in terms of which pointcuts can be defined.

7.6 Discussion

Detection of fragile pointcuts

Model-based pointcuts, and our particular instantiation of view-based pointcuts, alleviate the fragile pointcut problem by:

- Decoupling the pointcut from the structure of the base program. This is achieved by expressing the pointcut in terms of the entities of a conceptual layer rather than in terms of the source-code entities of the base program directly. These conceptual entities are a first-class representation of the concepts that traditional pointcuts capture by means of relying on the structural source-code regularities that govern these concepts. More concretely, our extension of the CARMA aspect language allows a definition of pointcuts in terms of intensional views that are specified over a system;
- Providing a means to keep the conceptual layer synchronized with the base program's source code. To this end, we incorporate constraints into the conceptual layer that aid in verifying that each concept classifies the correct source-code entities. In view-based pointcuts, the constraints over intensional views as well as the support offered by the IntensiVE tool suite allow a developer to keep the conceptual layer and the source code mutually consistent.

Our approach does not prevent the occurrence of the fragile pointcut problem. If the source-code of the base program is altered in such a way that the regularities governing the concepts on which a pointcut relies are violated, the pointcut does become fragile. However, using model-based pointcuts this fragility is shifted from the actual pointcut definition to the conceptual layer. Not only are the concepts on which a pointcut relies made explicit in this conceptual layer, we can also provide support to deal with the evolution conflicts that result in the fragility of the pointcut in this conceptual layer.

The constraints that are imposed on the conceptual layer document the "design rules" that govern a certain concept in the source code of the base program. It are these rules which traditional pointcut mechanisms are forced to rely on in order to capture concepts that are only implicitly available in the source code of the base program. Since the constraints of the conceptual layer make these rules both explicit as well as verifiable with respect to the source code, upon evolution of the base code certain evolution conflicts that would otherwise lead to the fragility of a pointcut can be detected and resolved. Our approach does not guarantee that *all* occurrences of the fragile pointcut problem are detected. This is highly dependent on the different concepts that are part of the conceptual layer and especially the constraints that govern these concepts.

Supporting other conceptual layers and pointcut languages

In this chapter we have reported on one concrete instantiation of model-based pointcuts which we implemented as a proof-of-concept. In particular, we demonstrated view-based pointcuts, an extension to the CARMA aspect-oriented language that provides support for expressing pointcuts in terms of intensional views.

In Section 2.2 of Chapter 2 we discussed a number of classification mechanisms that can be employed to document the different concepts that are prevalent in the source code of a system, such as for instance CME, Cosmos and Concern Graphs. In analog to the use of intensional views, such classification mechanisms could also be adopted as a means to express the conceptual layer in an instantiation of model-based pointcuts. However, model-based pointcuts impose the following two requirements on such classification mechanisms:

- The classifications must provide an "active" documentation for the concepts in the base program. In other words, the pointcut language must have access to the source-code entities in the base program *via* the classifications;

- The classification mechanism must provide a mechanism (e.g. the use of constraints) to keep the classifications synchronized with the implementation. If such a mechanism is not provided, no support is offered to detect evolution conflicts that are caused by changes in the base program. As a result, the pointcuts expressed in terms of this conceptual layer might become fragile.

Furthermore, existing pointcut languages can also be extended with predicates to query the conceptual layer defined for a system. While in our implementation of view-based pointcuts, we were able to integrate this conceptual model elegantly by means of the fact that intensional views can be accessed as first-class entities from within SOUL, in practice a predicate like `isClassifiedAs` that verifies whether a certain source-code entity belonging to a specific classification should suffice.

Extensive validation

In this chapter we did not provide an in-depth validation of model-based pointcuts. Instead, we demonstrated how our approach was able to detect evolution conflicts that resulted in the fragility of two aspects defined on the SmallWiki case study. Although this was only a small case study that allowed us to illustrate our approach, we feel that these initial results are promising. However, a more detailed validation on larger cases is necessary to fully validate our claims.

7.7 Conclusions

In this chapter we discussed how our approach for documenting and co-evolving structural source-code regularities can be used to manage the evolution of aspect-oriented software, or more precisely to support the co-evolution of pointcuts and the base program to which they are applied. We have illustrated that the fragile pointcut problem, one of the open issues in aspect-orientation that can seriously hamper the evolvability of aspect-oriented programs, is caused by too tight a coupling between a pointcut definition and the source code of the base program. Using a number of examples, we illustrated that this tight coupling originates from the fact that aspect developers rely heavily on how the base program is structured. For a pointcut developer to capture a certain concept that is implicitly available in the source code, the developer assumes the presence of "design rules" that govern how this concept is implemented in the source code. If upon evolution these design rules are not respected, this can lead to the pointcut definition becoming brittle.

Our approach, model-based pointcuts, provides a means to tackle the fragile pointcut problem by decoupling pointcut definitions from the actual structure of the source code and expressing them in terms of a conceptual model of the software. By providing verifiable constraints over this conceptual model, it becomes possible to detect and resolve evolution conflicts that would otherwise lead to the fragile pointcut problem. As a proof-of-concept of this approach, we implemented view-based pointcuts. View-based pointcuts provide an extension to the CARMA language in which pointcuts can be expressed using a conceptual model created by means of intensional views and constraints on these views.

Model-based pointcuts serve as a nice illustration of how the documentation created using intensional views and constraints is applicable to other software engineering purposes. By extending an aspect-oriented programming language and integrating IntensiVE in this language, we are able to support the co-evolution of pointcuts and base programs. This application of intensional views shows that our approach is not restricted to supporting structural source-code regularities throughout the development process, but that our formalism is general enough to support other software development tasks.

Chapter 8

Conclusion and Future work

8.1 Summary

The premise of this dissertation is that developers introduce structural source-code regularities in the implementation of a system as a means to deal with the inherent complexity of such systems. Through rigorous use of naming conventions, programming idioms, design patterns, and so on, developers aim to communicate their intent, to improve comprehensibility of the source code or to direct the correct implementation of certain concerns.

Since these regularities are only *implicitly* available in the source code, evolution of the system can result in the causal link between the regularities and the source code no longer being upheld. When the source code of the system is altered, it is not guaranteed that these changes respect the different regularities in the system. In the same vain changes in the regularities, for example due to altered coding or design policies, can result in mismatches between these regularities and the source code. Consequently, there is a need for approaches that aid a developer in sustaining this implicit link between regularities and the implementation.

In this dissertation we introduced a novel approach that aims at alleviating the aforementioned problem. We presented the model of intensional views and constraints as a formalism for documenting a wide variety of structural source-code regularities and to verify these regularities with respect to the source code. This documentation turns the *implicit* structural source-code regularities into *explicit*, first-class entities. Our model is based on a software classification mechanism in which sets of related source-code entities are grouped to document a particular concept in the source code. By imposing constraints on these classifications, the different regularities that govern the concept documented by the classification are made explicit and verifiable.

To complement this model of intensional views, we proposed a methodology that aims at actively integrating structural source-code regularities into the development process. To this end, our methodology advocates the co-design and co-evolution of regularities and source code. We propose to turn the causal link between regularities and source code into an integral part of the development process. Rather than considering regularities and source code as two separate, isolated entities, the methodology we proposed in this dissertation enables the development and evolution of both regularities and source code in unison.

We presented our research prototype – IntensiVE – as a technical contribution for validating our approach. This tool suite provides a concrete instantiation of the model of intensional views and constraints implemented as an extension to the VisualWorks Smalltalk development environment. To support our methodology, this tool suite is tightly integrated with the surrounding development environment and the testing framework of this environment.

Finally, we introduced *model-based pointcuts* as a means to tackle the fragile pointcut problem, one of the open evolution problems within aspect-orientation. We have demonstrated that this fragile pointcut problem is caused by the too tight coupling between pointcut definitions and the structure of the source code of the base program which aspects have been defined on. Consequently, changes to the structure of the source code can result in the pointcuts becoming fragile. To alleviate this problem we effectively proposed to decouple the pointcut definition from the source code and rather express this pointcut in terms of a *conceptual layer* defined using intensional views and constraints.

8.2 Conclusion

Due to their importance to software development, it is important that the different structural source-code regularities that are introduced into the source code of a system are (and remain) correctly adhered to. Throughout this dissertation we have argued that the model of intensional views complemented with the associated methodology and tool suite provides a viable means for maintaining the causal link between regularities and source code, aiding in the consistent propagation of regularities in the implementation of a software system.

First, our formalism of intensional views presents a dedicated conceptual framework, independent of the used query language, that is sufficiently expressive to create verifiable documentation for a wide range of structural source code regularities. As we have shown in our experiments, our concrete instantiation of the model can be used to document the regularities underlying implementation-specific, application-specific and domain-specific concepts and verify these regularities against the source code.

Second, our approach succeeds at emphasizing the link between regularities and source code by integrating the regularities into the development process. This is achieved by co-designing the regularities and the source code. The documentation and the source code are developed in unison. Through an iterative process both regularities and source code are refined and matched. Furthermore, the test-often philosophy we propose aims at identifying discrepancies between documentation and implementation early on, thus supporting the co-evolution of both. We put our approach to practice in a number of case studies. These case studies illustrate that our approach is able to identify interesting evolution conflicts in which both updates to the source code as well to the documented regularities are needed in order to maintain the causal link between regularities and implementation.

We also observed from our case studies that this documentation and verification of regularities does not only aid in maintaining the causal link between regularities and source code. Our case studies indicated that the documentation and the process of creating documentation can also support software comprehension (SmallWiki) or verification of framework instantiations (DelfSTof). Furthermore, model-based pointcuts – the technique we propose to alleviate the fragile pointcut problem – illustrate that our model of intensional views and constraints is general enough to support other software evolution problems.

8.3 Limitations and Future work

In this section we discuss some of the limitations of our approach and propose future work aimed at alleviating these limitations. Furthermore, we introduce some directions of future research that build upon the work presented in this dissertation.

8.3.1 Improvements

We start our discussion of future work by proposing a number of technical improvements of the work we have presented and in particular of the IntensiVE tool suite.

A template mechanism for intensional views and constraints

In Chapter 5 we already discussed one possible improvement to our tool suite, namely the introduction of a template mechanism that renders intensional views and constraints more reusable. We have noticed that up to a certain degree views and constraints contain reusable information. For instance, the intensional views and constraints we defined on the instantiations of a design pattern are often portable to other instantiations of that pattern. Similarly, we observed certain "patterns" in intensional views and constraints we specified. E.g. we often encountered intensional views that grouped classes and methods in one particular class hierarchy.

Therefore we want to offer developers the ability to turn views and constraints into reusable entities. We wish to provide a template mechanism in which developers can give a partial specification of an intensional view or constraint. This enables a developer to create a "library" of intensional views and constraints which can be instantiated by completing the specification of the view or constraint.

Incremental verification of intensional views and constraints

Throughout this dissertation we have argued that our tool suite provides a tight integration with the surrounding development environment. This tight integration makes it possible for developers to directly browse the source-code entities that belong to the extension of an intensional view. Similarly, the discrepancies between a constraint and the implementation can be accessed from within the IntensiVE tool suite. Furthermore, the documentation created using our tools is a first-class entity offered to the surrounding development environment. For instance, this allows us to extend the CARMA pointcut language with a means to specify model-based pointcuts in terms of the intensional views that are defined over a system. Furthermore, we proposed a test-often philosophy as a means to detect discrepancies between the documented regularities and the source code as early as possible during development. Our tool suite supports this methodology by means of an extension to the unit testing framework of the surrounding development environment.

To take this integration one step further, we wish to alter the implementation of IntensiVE such that the *incremental verification* of the set of intensional views and constraints is supported. Instead of the test-driven approach our tool suite provides in its current incarnation, such incremental verification would enable the change-driven verification of the documented regularities. We envision tool support in which changes to the source code of the system, or to the documented regularities automatically trigger the verification process. Upon evolution of the system, this would *immediately* offer a the developer information concerning possible discrepancies between the regularities and the implementation.

However, this active verification of the documentation poses some technical challenges. The integration of this change-driven verification into the development environment should be conceived in a non-intrusive way. As a result, this iterative verification must not be allowed to impose a significant run-time overhead during the development process. Consequently, it is not feasible to verify the entire set of documentation whenever a change is made in the system. Rather, only the documentation that is related to the changes of the system needs to be verified.

One possible technology we wish to use to achieve such an iterative verification of the documentation is a forward chaining algorithm [For82]. For a logic-based query language (cf. SOUL) forward chaining presents a data-driven evaluation of queries. Using such a forward chainer, changes in the system will not trigger the evaluation of all rules but rather propagate the changes throughout the rules that are influenced by these changes.

Improved visualization

While our tool suite offers the visualization of the intensional views and constraints to provide a general overview of the validity of the regularities in a system, this visualization is rather primitive. Our *Visualization Tool* is restricted in the sense that it only provides a visual representation of the validity of the entirety of the documented regularities. We indicate by means of a color scheme whether or not an intensional view or an intensional constraint is consistent with respect to the implementation. Given the richness of information that visualization tools can provide a developer, we aim to improve our *Visualization Tool* such that it can offer a developer a wealth of information concerning the validity of the documented regularities in a single view.

As we already hinted in [MKPW06], more advanced visualization schemes such as for instance CodeCrawler [Lan03, DL05] provide a visual representation of different metrics on the system. We can adopt such a scheme by letting the geometrical properties of the entities in the visualization depend on quantitative information concerning the intensional views and constraint. For example, if we visualize an intensional view by means of a rectangle, we could define the height of that rectangle in terms of the number of tuples that belong to the view's extension. We could let the width of the rectangle vary with the number of alternative views, the number of discrepancies, etc. Similarly, if we visualize a binary relation as a line between two intensional views, we could let the width of the line depend on the number of discrepancies that violate the relation.

Such visualizations are not only applicable to provide an overview of the entire set of intensional views and constraints. In our current implementation the *Extensional Consistency*

Inspector and the *Relation Consistency Inspector* – our tools for verifying consistency of respectively the alternative views and intensional constraints – provide a textual representation of the discrepancies between the documentation and the source code. If the intensional view contains multiple alternatives or if the number of discrepancies grows, this textual representation can become cluttered. It is our intent to replace these tools with versions that provide a graphical representation of the discrepancies. For instance, such a graphical representation of the extensional consistency of the alternative views would not only make it possible to identify the tuples that violate extensional consistency, but would enable a developer to quickly assess which tuples are violating which of the alternative views.

8.3.2 Advanced querying facilities

Another direction of further research we propose is to investigate how other query languages and paradigms can be combined with our approach as a means to express the intension of a view or the predicate of an intensional constraint. Our current implementation offers two such languages, namely Smalltalk and SOUL. As we have demonstrated, these languages allow us to create intensional views and constraints that can document a wide variety of structural source-code regularities. However, in the following sections we discuss a number of approaches that can improve over these query languages.

Combination with static analysis techniques

The two languages, namely SOUL and Smalltalk, that our tool suite supports in order to express the intension of a view or the predicate of a constraint provide a purely structural analysis of the source code in terms of the parse tree of a program. For instance, if we specify a predicate that verifies whether a method m contains a call to a method n, this relationship is verified by checking that the parse tree of method m contains a send of the message implemented by method n. The obvious advantage of this verification scheme is that it can be performed quite efficiently. However, this naive approximation of a calling relation between two methods comes at the cost of a lack of precision. Consequently, this can result in the introduction of false positives and false negatives.

To alleviate these problems we wish to investigate how our formalism and tool suite can be complemented with source code meta-models and query languages that offer a more precise approximation of the semantics of a program. In particular, the use of static analysis [NNH99] and abstract interpretation [CC77] techniques seem to offer a viable candidate for improving the accuracy of the statically verified relations between source-code entities. These techniques aim at providing a conservative approximation of the semantics of a program without actually executing this program. Examples of such techniques are for instance call-graph analysis and pointer analysis [Hin01].

However, the application of such static analysis techniques can introduce a significant run-time overhead on the verification process. Some techniques provide a complex, time-consuming analysis to extract a particular property of a program. Since we advocate a test-often philosophy as a means to detect infringements against structural source-code regularities early on during the development process, we need to find a good trade-off between precision

```
jtClassDeclaration(?c) {
  class ?c {
    private ?type ?field;
    public ?type ?name() {
      return ?field;}
    }
}
```

Figure 8.1: Example of template queries for retrieving the accessor methods in a system.

and efficiency.

Application of fuzzy reasoning

Another extension we propose to the query facilities offered by our approach is the use of fuzzy reasoning. In particular, fuzzy set theory [Zad65] and fuzzy logic programming [Lee72] seem interesting techniques to combine with our approach. In fuzzy set theory, the elements belonging to a set are attributed a certain membership degree (a value between 0 and 1) which expresses to what degree an element belongs to the set.

One concrete way of integrating such fuzzy reasoning techniques with our approach is by supporting the fSOUL [DBD06] language, a fuzzy logic programming extension to SOUL, as a query language to specify the intension of a view. By using this language, we can take uncertainty into account during the evaluation of an intension. This uncertainty is reflected in the extension of the intensional view by associating a membership degree to each tuple in the extension.

When reasoning about software, this fuzzy interpretation can be used to deal with source-code entities that slightly deviate from the pattern expressed by the query [DBN^{+}07, DBD06]. Consequently, elements that *almost* match an intension but not entirely will be included in the extension of a view. However, a lower membership degree will be associated with these elements. One application we wish to explore of this integration of fuzzy reasoning with intensional views is to compare the results of 'crisp' evaluation with those of a fuzzy evaluation of the intension of a view. We believe that the tuples that are part of the fuzzy evaluation of the intension but not of the crisp evaluation might often reveal interesting information to the developer.

For example, fSOUL provides a unification mechanism for identifiers that takes linguistic deviations into account. If we would evaluate an intension `methodWithName(?method,drawFigure)` using the crisp SOUL evaluator, this will result in all methods that are precisely named `drawFigure`. Evaluating the same intension using fSOUL would also capture a method misspelled `drawFigrue`, however, a lower membership degree would be associated with this method.

Source code template queries

As a means to improve the ease of use of the querying facilities offered by our approach to express the intension of a view, we propose the use of *Concrete source code template queries* [DBN$^+$07]. This work by De Roover et al. provides an extension to logic-based query languages (such as SOUL) in which developers can query the source code of a system by specifying an implementation template. Instead of specifying a logic program that reasons over the source code of a software system, a developer can query the system by providing a prototypical implementation containing logic variables of a concept that is being queried. This template is then matched with the implementation of the system.

An example of source code templates (adopted from [DBN$^+$07]) is shown in Figure 8.1. This example illustrates how template queries can be used to retrieve the accessor methods in a Java program. Recall that we encountered the implementation idiom for accessors numerous times throughout this dissertation. We characterized a getter method as a method returning the value of a field. The template query in Figure 8.1 literately translates this implementation pattern into a query. The template matches all classes `?c` that contain a private field `?field` of type `?type`. Lines 4 and 5 capture the definition of the actual accessor methods. Namely, a method called `?name` is considered an accessor method if it is public, and contains a return statement that returns the value of the `?field` variable. Furthermore, this template requires that the return type of the method and the type of the field match.

As a means to match a template query with the source code, a combination of structural and behavioral representations of the source code is used. Rather than syntactically matching the template with the source code, template queries offer a means to introduce variability in the expressed pattern by applying a number of static analysis techniques as we have mentioned above such as points-to analysis and call-graphs. This allows template queries to match a template to implementations that deviate slightly from the implementation pattern. For example, consider the following getter method:

```
public Integer getIndex() {
  Integer temp;
  temp = index;
  return temp;
}
```

This getter method deviates from the prototypical implementation by assigning the value of the field `index` to a temporary variable and then returning the value of this temporary variable rather than directly returning the value of the field. However, due to the use of points-to analysis, the above method will be captured by the template query in Figure 8.1.

8.3.3 Structural regularity mining

To aid a developer in uncovering the structural source-code regularities that govern a system, we propose further research into the development of (semi-)automated techniques that mine the source code of a system for regularities. This process of discovering structural source-code regularities seems strongly related to *aspect mining* [KMT07], a novel research direction in aspect-oriented software development that aims at uncovering crosscutting concerns in existing code bases.

Such aspect mining techniques are based on the assumption that crosscutting concerns are characterized by a number of symptoms such as for instance the rigorous use of naming and stylistic conventions [STP05, MT05], high fan-in values [MvM04] or code duplication [BvvT05]. By using source code analysis approaches such as clone detection, or techniques from data mining such as cluster analysis [JD88] and formal concept analysis [GW99], these aspect mining techniques mine the source code for groups of source-code entities that all exhibit a similar symptom of crosscutting.

The task of mining for structural source-code regularities is based on the same assumptions as aspect mining, namely that certain concepts (crosscutting or not) in the source code of a system are characterized by a recurring pattern in their implementation. We feel that techniques similar to those for mining aspects can be used to discover structural source-code regularities. Although some aspect mining techniques are particularly devised to only detect instances of crosscutting concerns, techniques such as the work of Tourwé et al. [MT05] and He et al. [HBZH05] propose a strategy in which the source code is mined for recurring patterns in general. Using extensive post-filtering, the set of results is reduced to those sets of entities that represent a crosscutting concern. Our intuition that these techniques are suitable for identifying regularities is strengthened by the findings of Tourwé and Mens [MT05]. They report on an experiment in which they applied formal concept analysis [GW99] to group source-code entities which contained similar substrings. While this experiment identified a number of crosscutting concerns, they were also able to identify certain implementation idioms, design patterns and domain-specific concepts that were characterized by a similar naming scheme.

The output of such a mining algorithm is a collection of sets of related source-code entities. As such, a developer still needs to transform this *extensional* set of entities into an *intensional* view. To automate this process, we propose the technique of inductive logic programming [BG95]. This machine learning technique takes as input a set of facts and returns a set of logic rules that provide a description for those facts. We have performed some initial experiments [TBKG04] using ILP as a means to induce an intension of a view automatically from a set of source-code entities. While these experiments yielded promising results, further research is needed.

8.3.4 Using intensional views and constraints to generate source code

Another avenue of future research we intend to investigate is the combination of intensional views and constraints with code generation [CE00] techniques. While our current approach allows maintaining the causal link between the regularities that govern a system and the system's implementation, using code generation will allow for a more pro-active use of this documentation.

In particular, we envision a system in which the specification of the intensional views and constraints lie at the basis of a code generation process that, within the bounds of possibility, aids a developer when implementing a concept by creating template code that adheres to the different regularities that govern the concept. In the same way, this code generation process can serve as a means to alter the source code of the system upon changes in the regularities. As a technical platform to implement this integration of intensional views and generative technology, the work of Brichau [Bri05] seems a suitable candidate. First, the work of Brichau is based on the declarative meta programming paradigm and the SOUL language thus easing technical integration of the program generators into our tool suite. Second, Brichau's work provides a means to compose different generators and provides (semi-automatic) conflict resolution for these generators. Due to the crosscutting nature of regularities, a single entity is often governed by multiple regularities. As such, the support for the composition of generators and for resolving conflicts that arise from the generation process play an important role in integrating the generative technology with intensional views.

8.3.5 Large-scale validation

In Chapters 5 and 6 we discussed a number of case studies which we performed in the context of the validation of this dissertation. However, none of these case studies was executed in an industrial, large-scale setting. As such, an important continuation of our work lies in validating our approach in an practical environment. Applying our formalism, methodology and tool suite in a real-life context provides us with means to assess the scalability of our approach and will result in better insights on whether the benefits our approach offers outweigh the costs they introduce. Furthermore, such a more elaborate evaluation of intensional views will most likely lead to other improvements and refinements of our tools and formalism.

8.4 Contributions

We conclude by repeating the contributions the work we have presented in this dissertation offers:

- A first contribution of our approach lies in the fact that we explicitly take the causal connection between regularities and source code into account during the development process. To this end, we propose the co-design and co-evolution of regularities and source code. We do not consider the development and evolution of regularities and implementation as two separate entities but rather develop them in unison;

- We propose the model of intensional views and constraints as a formalism to create verifiable documentation for structural source-code regularities. This model is independent of the software meta-model of the programming language on which a developer imposes views and constraints as well as of the query language used to specify the views/constraints. Our formalism, tailored towards the documentation and evolution of regularities, offers the advantage that it is expressive enough to support a wide variety of regularities.

- A methodology is proposed that supports our model of intensional views and constraints. This methodology describes a structured way of documenting regularities using our formalism and offers a set of guidelines of how to incorporate the documented structural source-code regularities into the development process. Our methodology proposes the step-wise, incremental refinement of intensional views and source code as a means to support co-design of both artifacts. By advocating a test-driven verification of the documented regularities, our methodology aims at maintaining the causal link between regularities and source code upon evolution;

- As a technical contribution of our dissertation, we implemented the IntensiVE tool suite. This research prototype is a concrete instantiation of the model of intensional views and constraints that allows the definition of verifiable documentation for regularities in Java and Smalltalk programs. Our tool suite offers the developer the SOUL logic language and the Smalltalk language as a means to express the intension of an intensional view and the predicate of a constraint. Furthermore, our tools tightly integrate with these query languages as well as with the surrounding development environment. This enables IntensiVE to fully support our methodology. Moreover, the documented intensional views and constraints are first-class entities that are accessible by other software engineering tools;

- Finally, our approach is general enough to support other software evolution problems. As one example of such a problem that can be supported by intensional views, we presented *model-based pointcuts*. This contribution to aspect-oriented programming alleviates the fragile pointcut problem by decoupling pointcut definitions from the structure of the source code. Model-based pointcuts are defined in terms of a *conceptual layer* which presents a reification of the important concepts in the system. In our instantiation of model-based pointcuts, we propose to define this conceptual layer by means of intensional views and constraints. Since this allows us to keep this conceptual layer synchronized with the source code, our approach can detect evolution conflicts that cause pointcut fragility and provide support for resolving these conflicts.

Appendix A

Proof

In this appendix we present the proof of equivalence between the regular version of intensional views and the version of intensional views that provides alternatives. We introduce the theorem before giving a formal proof.

Theorem 1

Given:
an intensional view $V = (Parents, default, Alt)$ as defined by Figure 3.13 with:

$$\begin{aligned}
&Parents = \emptyset \\
&default = alt_1 \\
&Alt = \{alt_2, alt_3, \ldots, alt_n\} \\
&alt_i = (Attr, query_i, Incl_i, Excl_i) \text{ for } 1 \leq \ i \ \leq n
\end{aligned}$$

If
for this view V we construct a regular intensional view

$$V' = (Attr, query, Parents, Incl, Excl)$$

as defined by Figure 3.6 with n unary intensional constraints U_i as follows:

$$\begin{aligned}
&Parents = \emptyset, Incl = \emptyset, Excl = \emptyset \\
&query : t \ \rightarrow \ query_1(t) \ \vee \ query_2(t) \ \vee \ \ldots \ \vee \ query_n(t) \\
&\forall \, i \ \in \ \{1..n\} : U_i = (V', \forall, pred_i, Incl_i, Excl_i) \\
&\forall \, i \ \in \ \{1..n\} : pred_i : t \ \rightarrow \ query_1(t) \ \wedge \ query_i(t) \\
&\text{(where } query_i \text{ is defined by the alternatives } alt_i \text{ belonging to } V\text{)}
\end{aligned}$$

then
we can show that V and V' are semantically equivalent in the following sense:

1. $extension(V) = extension(V') \Leftarrow consistent(V)$.
2. $consistent(V) \iff \forall\, i \in \{1..n\} : consistent(U_i)$
3. $discrepancies(V) = \bigcup_{i \in \{1..n\}} discrepancies(U_i)$

To simplify the proof, we will assume that the $Incl_i$ and $Excl_i$ sets are empty. Notice that this does not have any impact on the validity of our proof since, for each intension $query_i$ of alternative view alt_i we could define an intension $query'_i$ such that:

$$query'_i(t) = (query_i(t) \lor t \in Incl_i) \land t \notin Excl_i$$

This intension $query'_i(t)$ takes the $Incl_i$ and $Excl_i$ directly into account. The proof for the general case can thus be "transformed" in a proof where the $Incl_i$ and $Excl_i$ sets are empty by replacing the occurrences of $query_i$ by $query'_i$.

Part 1

To prove:

$$extension(V) = extension(V') \Leftarrow consistent(V)$$

Proof:

$$\begin{aligned}
extension(V) &= extension(V_{default}, V) && \text{(Definition 23)}\\
&= \{t|query_1(t)\} && \text{(Definition of } alt_1)\\
&= \{t|query_1(t)\} \lor \{t|query_2(t)\} \lor \ldots \lor \{t|query_n(t)\} && \text{(Since } consistent(V))\\
&= \{t|query_1(t) \lor query_2(t) \lor \ldots \lor query_n(t)\}\\
&= extension(V') && \square
\end{aligned}$$

Part 2

To prove:

$$consistent(V) \iff \forall\, i \in \{1..n\} : consistent(U_i)$$

Proof:

$$\begin{aligned}
consistent(V) \iff\ & \forall\, alt_i \in V_{Alt} : extension(alt_i, V) = extension(V_{default}, V) && \text{(Definition 24)}\\
\iff\ & extension(alt_2, V) = extension(V_{default}, V) \wedge\\
& \ldots \wedge\\
& extension(alt_n, V) = extension(V_{default}, V)\\
\iff\ & extension(alt_2, V) = extension(alt_1, V) \wedge && (V_{default} = alt_1)\\
& \ldots \wedge\\
& extension(alt_n, V) = extension(alt_1, V)\\
\iff\ & extension(alt_1, V) = extension(alt_2, V) \wedge\\
& \ldots \wedge\\
& extension(alt_1, V) = extension(alt_n, V)\\
\iff\ & (\forall t \in extension(alt_1, V) : t \in extension(alt_2, V) \wedge\\
& \forall t \in extension(alt_2, V) : t \in extension(alt_1, V)) \wedge\\
& \ldots \wedge\\
& (\forall t \in extension(alt_1, V) : t \in extension(alt_n, V) \wedge\\
& \forall t \in extension(alt_n, V) : t \in extension(alt_1, V))\\
\iff\ & \forall t \in extension(alt_1, V) : query_2(t) \wedge && \text{(Definition of intension of } alt_i\text{)}\\
& \forall t \in extension(alt_2, V) : query_1(t) \wedge\\
& \ldots \wedge\\
& \forall t \in extension(alt_1, V) : query_n(t) \wedge\\
& \forall t \in extension(alt_n, V) : query_1(t) \wedge\\
\iff\ & \forall t \in extension(alt_1, V) : (query_2(t) \wedge \ldots \wedge query_n(t)) \wedge\\
& \forall t \in extension(alt_2, V) : query_1(t) \wedge\\
& \ldots \wedge\\
& \forall t \in extension(alt_n, V) : query_1(t)\\
\iff\ & \forall t \in extension(alt_1, V) : (query_1(t) \wedge query_2(t) \wedge \ldots \wedge query_n(t)) \wedge\\
& && \text{(Definition of } alt_1\text{)}\\
& \forall t \in extension(alt_2, V) : query_1(t) \wedge\\
& \ldots \wedge\\
& \forall t \in extension(alt_n, V) : query_1(t)\\
\iff\ & \forall t \in \{t | query_1(t)\} : (query_1(t) \wedge query_2(t) \wedge \ldots \wedge query_n(t)) \wedge\\
& && \text{(Definition of intension of } alt_i\text{)}\\
& \forall t \in \{t | query_2(t)\} : query_1(t) \wedge\\
& \ldots \wedge\\
& \forall t \in \{t | query_n(t)\} : query_1(t)\\
\iff\ & \forall t \in \{t | query_1(t) \vee \ldots \vee query_n(t)\} : query_1(t) \wedge query_2(t) \wedge \ldots \wedge query_n(t)
\end{aligned}$$

$$\begin{aligned}
\forall\, i \in \{1..n\} : consistent(U_i) &\iff consistent(U_1) \wedge \ldots \wedge consistent(U_n) \text{ with } 1 \leq i \leq n \\
&\iff (\forall\, t \in extension(V') : query_1(t) \wedge query_2(t)) \wedge \quad \text{(Definition of } U_i\text{)} \\
&\quad\ \ldots \wedge \\
&\quad\ (\forall\, t\; in\; extension(V') : query_1(t) \wedge query_n(t)) \\
&\iff \forall\, t \in extension(V') : query_1(t) \wedge query_2(t) \wedge \ldots \wedge query_n(t) \\
&\iff \forall\, t \in \{t | query_1(t) \vee \ldots \vee query_n(t)\} : \\
&\qquad query_1(t) \wedge query_2(t) \wedge \ldots \wedge query_n(t) \quad \text{(Definition of } V'\text{)} \\
&\iff consistent(V) \qquad \square
\end{aligned}$$

Part 3

To prove:

$$discrepancies(V) = \bigcup_{i \in \{1..n\}} discrepancies(U_i)$$

Proof:

$$\begin{aligned}
discrepancies(V) &= \bigcup_{i \in \{1..n\}} extension(alt_i) \setminus \bigcap_{i \in \{1..n\}} extension(alt_i) \quad \text{(Definition 25)} \\
&= \{t | query_1(t) \vee \ldots \vee query_n(t)\} \setminus \{t | query_1(t) \wedge \ldots \wedge query_n(t)\} \\
&\qquad \text{(Definition of intension of } alt_i \text{ + conjunction/disjunction)} \\
&= \{t | (query_1(t) \vee \ldots \vee query_n(t)) \wedge \neg(query_1(t) \wedge \ldots \wedge query_n(t))\}
\end{aligned}$$

$$\begin{aligned}
\bigcup_{i \in \{1..n\}} discrepancies(U_i) &= discrepancies(U_1) \cup \ldots \cup discrepancies(U_n) \text{ with } 1 \leq i \leq n \\
&= \{t | t \in extension(V') \wedge \neg(query_1 \wedge query_2)\} \cup \ldots \cup \quad \text{(Definition 12)} \\
&\quad\ \{t | t \in extension(V') \wedge \neg(query_1 \wedge query_n)\} \\
&= \{t | t \in extension(V') \wedge (\neg(query_1 \wedge query_2) \vee \ldots \vee \neg(query_1 \wedge query_n))\} \\
&= \{t | (query_1(t) \vee query_2(t) \vee \ldots query_n(t)) \wedge (\neg query_1(t) \vee \ldots \vee \neg query_n(t))\} \\
&\qquad \text{(Definition of } V'\text{)} \\
&= \{t | (query_1(t) \vee query_2(t) \vee \ldots query_n(t)) \wedge \neg(query_1(t) \wedge \ldots \wedge query_n(t))\} \\
&= discrepancies(V) \qquad \square
\end{aligned}$$

Appendix B

Intensional Views and Constraints over IntensiVE

B.1 Intensional views

B.1.1 Core Model

We define three intensional views *Core Model*, *Views Model* and *Constraints Model* which capture the source-code entities that implement the core of the formalism underlying the IntensiVE tool suite.

Name: Core Model	**Attributes:** class, method **Parent view:** None
Description: All classes and methods in the implementation of the core model, i.e. the union of all entities implementing views and constraints.	
Intension:(Smalltalk) `Views.ViewsModel union:(Views.RelationsModel)`	
Alternative intensions: /	

Name: Views Model	**Attributes:** class, method **Parent view:** None
Description: All classes and methods implementing the model of intensional view.	
Intension:(SOUL)	
Alternative intensions: /	

```
or(packageWithName(?pack,['Intensional Views Model']),
   packageWithName(?pack,['Predicate Views Model'])),
classInPackage(?class,?pack),
methodInClass(?method,?class)
```

Name: Constraints Model

Attributes: class, method
Parent view: None

Description:
All classes and methods implementing the model of intensional constraints.

Intension:(SOUL)

```
or(packageWithName(?pack,['Intensional Relations Model']),
   packageWithName(?pack,['Predicate Relations Model'])),
classInPackage(?class,?pack),
methodInClass(?method,?class)
```

Alternative intensions: /

B.1.2 Intensional view implementation

The following intensional views capture concepts from the domain of intensional views and its saving mechanism.

Name: Intension Evaluators

Attributes: class, method
Parent view: Views Model

Description:
The entities responsible for evaluating an intension. Three alternatives:

1. All classes in the correct hierarchy;
2. All classes implementing the correct interface;
3. All classes of which the name ends with “IntensionEvaluator”.

Alternative intensions:(SOUL)

```
1.classInHierarchyOf(?class,[AbstractIntensionEvaluator])
2.methodWithNameInClass(?,[#'checkIntension:'],?class),
  methodWithNameInClass(?,[#'eval:inContext:'],?class),
  methodWithNameInClass(?,[#'eval:inContext:bindings:'],?class),
  methodWithNameInClass(?,[#language],[?class class])
3.['*IntensionEvaluator' match: ?class name]
```

Name: Saving

Attributes: method
Parent view: Views Model

Description:
All methods that implement a save operation on an intensional view.

Intension:(SOUL)

```
methodWithName(?method,[#saveIn:])
```

Alternative intensions: /

Name: Extension	**Attributes:** method **Parent view:** Core Model
Description: All methods on the `Extension` class which are compatible with the collection hierarchy. Notice this is the example of the Adapter design pattern from Chapter 5. We define two alternatives for this view: 1. all methods in the `Extension` class implemented in the correct protocol; 2. all methods in the `Extension` class which are also implemented by the `Collection` class.	

Alternative intensions:(SOUL)
`1.methodInClass(?method,[Extension]),`
`  methodInProtocol(?method,[#'collection compatibillity'])`
`2.methodWithNameInClass(?method,?name,[Extension]),`
`  classUnderstands([Collection],?name)`	

Name: Compilation	**Attributes:** class, method **Parent view:** Core Model
Description: All methods involved in the compilation of intensional views and constraints.	

Intension:(SOUL)
`methodInProtocol(?method,?protocol),`
`['*compil*' match: ?protocol asString]`	

Alternative intensions: /	

B.1.3 Constraints model

The intensional views in this section describe a number of domain concepts related to constraints over intensional views.

Name: Predicate Evaluation	**Attributes:** class, method **Parent view:** Constraints Model
Description: All evaluators of the predicate of a constraint. Two alternatives: 1. All classes in the correct hierarchy; 2. All classes implementing the correct interface.	

Alternative intensions:(SOUL)
`1.classInHierarchyOf(?class,[AbstractPredicate])`
`2.methodWithNameInClass(?,[#'evaluateForSource:target:'],?class)`	

Name: Quantifiers	**Attributes:** class, method **Parent view:** Constraints Model
Description: The implementation of quantifiers. Three alternatives: 1. Classes in the correct hierarchy; 2. Classes containing the name “Quantifier”; 3. Classes implementing the correct interface.	

Alternative intensions:(SOUL)
`1.classInHierarchyOf(?class,[AbstractQuantifier])`
`2.['*Quantifier*' match: ?class name]`
`3.methodWithNameInClass(?,[#symbol],?class),`
`  methodWithNameInClass(?,[#'evaluateFor:predicate:'],?class)`	

Name: Quantifier Evaluation	**Attributes:** method **Parent view:** Quantifiers
Description: The methods implementing the actual evaluation of a quantifier.	
Intension:(SOUL) `methodWithName(?method,[#’evaluateFor:predicate:’])`	
Alternative intensions: /	

Name: Constraint Evaluators	**Attributes:** class, method **Parent view:** Constraint Model
Description: The evaluation of the predicate in a constraint.	
Intension:(SOUL) `classInHierarchyOf(?class,[RelationEvaluator])`	
Alternative intensions: /	

Name: Constraint Type Evaluators	**Attributes:** class, method **Parent view:** Constraint Evaluators
Description: The evaluation of a unary or binary intensional constraint. These classes use a strategy design pattern to identify the actual SOUL/Smalltalk evaluator.	

Intension:(SOUL)
`classInHierarchyOf(?class,[PredicateRelationConstraintEvaluator]),`
`methodInProtocol(?method,evaluation)`	

Alternative intensions: /	

Name: Constraint Language Evaluators	**Attributes:** class, method **Parent view:** Constraint Evaluators
Description: Evaluators of a constraint depending on the SOUL or Smalltalk query language. 1. All classes in the correct hierarchy; 2. All classes implementing the correct interface.	

Alternative intensions:(SOUL)
`1. classInHierarchyOf(?class,[PredicateRelationEvaluator])`
`2. methodWithNameInClass(?,[#’evaluateConstraints:withSource:for:withTarget:for:’],`
`                         ?class)`	

B.1.4 Graphical user interface

Name: GUI	**Attributes:** class, method **Parent view:** None
Description: All classes and methods involved in the implementation of the user interface part of IntensiVE.	

Intension:(SOUL)
`or(packageWithName(?package,[’Intensional Views UI’]),`
`   packageWithName(?package,[’Intensional Relations UI’]),`
`   packageWithName(?package,[’Predicate Views UI’])),`
`classInPackage(?class,?package),`
`methodInClass(?method,?class)`	

Alternative intensions: /	

Name: Editors

Attributes: class, method
Parent view: GUI

Description:
All editors (e.g. *Intensional View Editor*, *Extensional Consistency Inspector*, ...).

Intension:(SOUL)

```
classInHierarchyOf(?class,[UI.ApplicationModel])
```

Alternative intensions: /

Name: Editor Actions

Attributes: class, method
Parent view: Editors

Description:
All the actions (like `save`, `check`, and so on) which are executable from within an editor.

Intension:(SOUL)

```
methodInProtocol(?method,[#'interface actions'])
```

Alternative intensions: /

Name: Editor model change

Attributes: class, method, messages
Parent view: Editor actions

Description:
All actions in the editor that change the state of an intensional view or constraint. Notice that the intensional view has an attribute 'messages' in which the actual change messages are stored. Smalltalk is used to specify the intension of this view due to the large number of change methods that need to be queried.

Intension:(Smalltalk)

```
| mut ext |
ext := Extension new.
"collect all selectors of mutator methods"
mut := (Views.CoreModelMutators forVariables:(Array with:#method))
      collect:[:tup | (tup valueFor:#method) selector].
"if an editor action sends a message that corresponds to a mutator,
 add a tuple to the extension"
Views.EditorActions do:[:tup | |mess t|
    mess := (((tup valueFor:#method) messages) intersection: mut).
    (mess size > 0) ifTrue:[
       t := tup copy.
       t attribute: #messages value: mess.
       ext add:t ]].
ext
```

Alternative intensions: /

Name: Exception Elements

Attributes: class, method
Parent view: GUI

Description:
Exception elements serve as wrappers around tuples in order to visualize them in the `Extensional Consistency Inspector`. We define this intensional view by two alternatives: 1. All classes in the correct hierarchy and 2. All classes implementing the correct interface.

Alternative intensions:(SOUL)

```
1. classInHierarchyOf(?class,[ExceptionElement])
2. methodWithNameInClass(?,asText,?class)
```

Name: Drag & Drop	**Attributes:** class, method **Parent view:** GUI
Description: All methods implementing drag & drop facilities in IntensiVE.	
Intension:(SOUL) `methodInProtocol(?method,[#'drag & drop'])`	
Alternative intensions: /	

Name: Drop actions	**Attributes:** class, method **Parent view:** Drag & Drop
Description: The methods implementing the actions which occur when an element is dropped. We define three alternative views for this view: • The naming convention that all drop methods must start with the prefix “drop-”; • All methods with as argument `aDragDrop`; • All methods which – by analyzing the windowSpec of an editor – are considered to be a drop action. The `dropMethod` predicate analyzes the windowSpec implemented by `?class` and binds `?method` to all actions that perform a drop.	

Alternative intensions:(SOUL)
`1.methodWithName(?method,?name),`
`  ['drop*' match: ?name]`
`2.argumentsOfMethod(<variable(aDragDrop)>,?method)`
`3.dropMethod(?class,?,?method)`	

Name: Drag initialize	**Attributes:** class, method **Parent view:** Drag & Drop
Description: All methods initializing a drag operation. Defined as either all methods which are a drag method in a windowSpec (alternative 1) or which send the selector `doDragDrop`. The `dragEnterMethod` predicate analyses the windowSpec defined by `?class` and extracts all bindings of `?method` that initialize a drag.	

Alternative intensions:(SOUL)
`1.dragEnterMethod(?class,?,?method)`
`2.methodSendsSelector(?method,[#doDragDrop])`	

B.1.5 Factory design pattern

The following intensional views are used to encode the instantiation of the Factory design pattern in IntensiVE.

Name: Factory	**Attributes:** class, method **Parent view:** IntensiVE
Description: All the Abstract Factories in IntensiVE. Defined as 1. All classes in the hierarchy of `Factory` and 2. All classes containing the string “Factory”.	

Alternative intensions:(SOUL)
`1. classInHierarchyOf(?class,[Factory])`
`2. ['*Factory*' match: ?class name]`	

Name: Factory methods	**Attributes:** method **Parent view:** Factory
Description: The Factory methods, i.e. all methods starting with the prefix “create-”.	

Intension:(SOUL)
`methodWithName(?method,?name),`
`['create*' match:?name]`	

Alternative intensions: /	

Name: Factory products	**Attributes:** class **Parent view:** Core Model
Description: The actual Products produced by the Factory methods. This intensional view is defined as all the classes in the Core Model which are not a Factory itself, because the Factory should instantiate all kinds of entities in the core model. Notice that this intensional view is defined in terms of both the *CoreModel* and the *Factory* intensional views.	

Intension:(SOUL)
`CoreModel(?class,?),`
`not(Factory(?class,?))`	

Alternative intensions: /	

Name: Factory clients	**Attributes:** class **Parent view:** Core Model
Description: The classes using a Factory to instantiate Products. It is defined as all classes that reference a Factory	

Intension:(SOUL)
`Factory(?factory,?)`
`methodReferencesClass(?method,?factory)`	

Alternative intensions: /	

B.1.6 Observer design pattern

The following two intensional views express a number of constraints related to the Observer design pattern.

Name: State changes	**Attributes:** class, method **Parent view:** IntensiVE

Description:
All the state changes in IntensiVE, i.e. all the methods that invoke a mutator method. For performance reasons, we opted to implement the intension of this view in Smalltalk.

Intension:(Smalltalk)

```
| extension |
extension := Extension new.
setters := Views.Mutators for:#method.
Intensional allClasses do:[:class |
   class selectors do:[:selector |
    setters do:[:setter | ((class compiledMethodAt: selector)
             sendsSelector:((setter valueFor: #method) compiledMethod selector))
                    ifTrue:[extension add:
                (Tuple new attribute:#class value:class;
                           attribute:#method value:
                              (SmalltalkMethod class:class selector: selector))]
]]].
extension
```

Alternative intensions: /

Name: Subjects	**Attributes:** class, method **Parent view:** IntensiVE

Description:
All the subjects of the Observer pattern in IntensiVE, defined as all the methods that send self `changed` or `changed:`.

Intension:(SOUL)

```
or(methodWithSend(?method,?,[#changed:],?),
   methodWithSend(?method,?,[#changed],?))
```

Alternative intensions: /

B.1.7 Deduce tool

The deduce tool is an experimental sub-tool of IntensiVE which is used to automatically mine for intensional constraints. In its implementation a chain of responsibility is used to implement the filtering mechanism which prunes the set of resulting constraints. The following two intensional views document this chain of responsibility.

Name: Deduce tool	**Attributes:** class, method **Parent view:** IntensiVE

Description:
All classes and methods in the implementation of the Deduce Tool.

Intension:(SOUL)

```
packageWithName(?package,['Deduce Tool']),
classInPackage(?class,?package),
methodInClass(?method,?class)
```

Alternative intensions: /

Name: Filter chain	**Attributes:** class **Parent view:** Deduce tool

Description:
The implementation of the filters of the Deduce tool, which are arranged by means of a chain. This intensional view consists of the following three alternatives:

- All classes in the `AbstractFilter` hierarchy;
- All classes with the suffix “-Filter”;
- All classes implementing the correct interface, i.e. the method `filterElement:`

Alternative intensions:(SOUL)

```
1.classInHierarchyOf(?class,[AbstractFilter])
2.['*Filter' match: ?class name asString]
3.methodWithNameInClass(?,[#'filterElement:'],?class)
```

B.1.8 Implementation idioms

A number of intensional views are used to capture an idiomatic implementation of a concept in the implementation of IntensiVE.

Name: Accept methods	**Attributes:** class, method **Parent view:** Core Model

Description:
The different services in the StarBrowser2 are implemented by means of a Visitor design pattern. The *Accept methods* intensional view groups all accept methods in the core model which accept such a visitor.

Intension:(SOUL)

```
methodWithName(?method,[#'acceptService:'])
```

Alternative intensions: /

Name: Visit methods	**Attributes:** class, method **Parent view:** None

Description:
All the visit methods implemented in the StarBrowser2 framework. These visit methods implement a certain service for a given element. This view consists of two alternative views: 1. All methods with as prefix “do-” in the correct hierarchy and 2. All methods in the protocol `operations`.

Alternative intensions:(SOUL)

```
1.classInHierarchyOf(?class,[Service]),
  methodWithNameInClass(?method,?name,?class),
  ['do*' match:?name]
2.classInNamespace(?class,[Classifications2]),
  methodInClass(?method,?class),
  methodInProtocol(?method, operations)
```

Name: Accessor methods	**Attributes:** class, method, field **Parent view:** Core Model
Description: All accessor methods characterized by either 1. A method with the same name as a field and 2. a method consisting of a single statement returning the value of a field.	

Alternative intensions:(SOUL)
`1.methodWithName(?method,?field),`
`  instanceVariableInClass(?field,?class)`
`2.statementsOfMethod(statements(<?statement>),?method),`
`  instanceVariableInClass(?field,?class),`
`  equals(?statement, return(variable(?field)))`	

Name: Core model mutator methods	**Attributes:** class, method, field **Parent view:** Core Model
Description: All mutator methods in the core model. Either defined as 1. all methods which correspond with the name of a field followed by a colon or 2. all methods containing a statement which performs an assignment to a field.	

Alternative intensions:(SOUL)
`1.methodWithName(?method,?name),`
`  instanceVariableInClass(?field,?class),`
`  [?field asString,’:’ = ?name asString]`
`2.statementsOfMethod(statements(?statements),?method),`
`  argumentsOfMethod(<?argument>,?method),`
`  instanceVariableInClass(?field,?class),`
`  member(assign(variable(?field),?argument),?statements)`	

B.1.9 Implementation constraints

The intensional views in this section are used to express constraints which are not directly related to the concepts in the implementation of IntensiVE, but rather group entities which are used to impose a number of constraints over the implementation. Such views are used to e.g. express bad smells, coding guidelines, . . .

Name: Unit tests	**Attributes:** class **Parent view:** Core Model
Description: All the unit tests in the implementation of IntensiVE.	
Intension:(SOUL) `classInHierarchyOf(?class,[XProgramming.SUnit.TestCase])`	
Alternative intensions: /	

Name: Hash/Equals	**Attributes:** class **Parent view:** IntensiVE
Description: This intensional view expresses the constraint that all classes which implement `equals` must also implement `hash` in order for comparison between objects to behave correctly. This view consists of two alternatives: 1. All classes implementing `hash` and 2. All classes implementing `equals`.	

Alternative intensions:(SOUL)
`1. methodWithNameInClass(?,hash,?class)`
`2. methodWithNameInClass(?,equals,?class)`	

Name: Private methods	**Attributes:** class **Parent view:** IntensiVE
Description: All methods classified in the private protocol.	
Intension:(SOUL) `methodInProtocol(?method,private)`	
Alternative intensions: /	

Name: Initialize methods	**Attributes:** class **Parent view:** IntensiVE
Description: All methods named `initialize`.	
Intension:(SOUL) `methodWithName(?method,initialize)`	
Alternative intensions: /	

Name: Saving mechanism	**Attributes:** class **Parent view:** IntensiVE
Description: The IntensiVE saving mechanism. This is implemented by all classes in the hierarchy of `IntensionalStorage`. These classes provide means to make the intensional views and constraints persistent.	
Intension:(SOUL) `classInHierarchyOf(?class,[IntensionalStorage])`	
Alternative intensions: /	

Name: Overridden initialize methods	**Attributes:** class **Parent view:** Initialize methods
Description: All methods overriding the `initialize` method.	
Intension:(SOUL) `overridingSelector(?class,initialize)`	
Alternative intensions: /	

B.2 Intensional Constraints

On the above intensional views, we define 23 intensional constraints:

Description:
The argument name of the single argument of a saving method must be `aLayer`.

Quantification:
∀ save ∈ *Saving*:

Predicate:
```
argumentsOfMethod(<variable(aLayer)>,?save.method)
```

Description:
The evaluator for the constraint of a type evaluator must contain an invocation of a language evaluator (use of the strategy design pattern).

Quantification:
∀ constraint ∈ *Constraint type evaluators*: ∃ language ∈ *Constraint language evaluators*:

Predicate:
```
methodCallsMethod(?constraint.method, ?language.method)
```

Description:
A quantifier evaluation method takes two arguments, namely `aCollection` and `aPredicate`.

Quantification:
∀ quant ∈ *Quantifier evaluation*:

Predicate:
```
argumentsOfMethod(<variable(aCollection), variable(aPredicate)>,?quant.method)
```

Description:
In order for the Visitor pattern to function properly, all accept methods should contain a call to a visit method.

Quantification:
∀ accept ∈ *Accept methods*: ∃ visit ∈ *Visit methods*:

Predicate:
```
methodCallsMethod(?accept.method, ?visit.method)
```

Description:
The name of the message invoked in the double dispatch of the visitor must match the name of the class that accepts the visitor. E.g. the method that accepts the class `Element` must be named `doElement:`

Quantification:
∀ accept ∈ *Accept methods*:

Predicate:
```
statementsOfMethod(statements(<?statement>), ?accept.method),
equals(?statement,return(send(?,?message,?))),
methodInClass(?accept.method,?class),
[('do', ?class name asString,':') match: ?message asString]
```

Description:
All compilation methods have a single argument `aStream`.

Quantification:
∀ compile ∈ *Compilation*:

Predicate:
```
argumentsOfMethod(<variable(aStream)>,?compile.method)
```

Description:
Compilation methods either invoke another compilation method or write to the stream.

Quantification:
∀ compile ∈ *Compilation*:

Predicate:
```
methodWithSend(?compile.method,?rec,?message,?args),
or(equals(?rec,variable(aStream)),
   and(Compilation(?,?method),
     equals(?message,[?method selector])))
```

Description:
All adapted methods on `Extension` must contain a call to the wrapped object.

Quantification:
∀ adapter ∈ *Extension*:

Predicate:
```
methodWithNameInClass(?adapter.method,?name,?),
statementsOfMethod(statements(<
return(send(
       send(variable(self), tuples,<>),
            ?name,
            ?args))>),?adapter.method)
```

Description:
Primitive dead code check: all classes of the core model should be referenced somewhere in the implementation of IntensiVE.

Quantification:
∀ core ∈ *Core Model*:

Predicate:
```
(Intensional allClasses select:[
   :class |
    (class methodDictionary values select:[
        :method | method classesReferenced
             includes:(core valueFor:#class)]) size > 0]) size > 0
```

Description:
Primitive test coverage: for each class in the core model there should exist a corresponding test case.

Quantification:
∀ core ∈ *Core Model*: ∃ test ∈ *Unit tests*:

Predicate:
```
[?core.class name,'Test' match: ?test.class name]
```

Description:
All filters must implement a method named `filterElement:`.

Quantification:
∀ handler ∈ *Filter chain*:

Predicate:
```
methodWithNameInClass(?,[#filterElement:], ?handler.class)
```

Description:
An element of the chain is not allowed to invoke `next`.

Quantification:
$\nexists$ handler $\in$ *Filter chain*:

Predicate:
```
methodWithSend(?handler.method, ?, next,?)
```

Description:
Filters can not reference each other; they need to be created via the chain.

Quantification:
$\nexists$ handler $\in$ *Filter chain*:

Predicate:
```
FilterChain(?,?method),
methodReferencesClass(?method, ?handler.class)
```

Description:
The implementation of a drop action must use the standard facilities of StarBrowser2. This is ensured by verifying that the `dragDropContext:` method implemented by `DropHelper` is used.

Quantification:
$\forall$ handler $\in$ *Drop actions*:

Predicate:
```
methodWithSend(?action.method, variable([#'Classifications2.DropHelper']),
        [#dragDropContext:],<variable(aDragDrop)>)
```

Description:
All changes in an editor to an intensional view/constraint must be performed by invoking a mutator method on the `model` field.

Quantification:
$\forall$ change $\in$ *Editor model change*: $\exists$ mutator $\in$ *Core mutator methods*:

Predicate:
```
member(?message,?change.messages),
methodWithSend(?change.method,send(variable(self),model,<>),
               [?mutator.method selector],?a)
```

Description:
All factory clients, i.e. methods that reference a Factory, must also send the `current` method.

Quantification:
$\forall$ client $\in$ *Factory client*:

Predicate:
```
methodSendsSelector(?client.method, current)
```

Description:
A client of the factory is not allowed to reference products directly; instead, such accesses should happen via the factory.

Quantification:
$\forall$ client $\in$ *Factory client*: $\nexists$ product $\in$ *Products*:

Predicate:
```
methodReferencesClass(?client.method, ?product.class)
```

Description:
All factory methods must reference a product class in order to instantiate it.

Quantification:
$\forall$ fac $\in$ *Factory methods*: $\exists$ product $\in$ *Products*:

Predicate:
```
methodReferencesClass(?fac.method, ?product.class)
```

Description:
There should exist a factory method for each product.

Quantification:
$\forall$ product $\in$ *Products*: $\exists$ fac $\in$ *Factory methods*:

Predicate:
```
methodReferencesClass(?fac.method, ?product.class)
```

Description:
All state changes should trigger a “changed” method.

Quantification:
$\forall$ statechange $\in$ *State changes*: $\exists$ subject $\in$ *Subjects*:

Predicate:
```
methodCallsMethod(?statechange.method, ?subject.method)
```

Description:
The Saving Mechanism is implemented as a singleton, as such, none of the classes are allowed to override `new`.

Quantification:
$\nexists$ saving $\in$ *Saving mechanism*:

Predicate:
```
overridingSelector([?saving.class class], initialize)
```

Description:
All calls to private methods must originate from the class hierarchy in which they are defined.

Quantification:
$\nexists$ private $\in$ *Private methods*:

Predicate:
```
CoreModel(?class,?method),
methodCallsMethod(?method,?private.method),
methodInClass(?private.method,?c),
not(classInHierarchyOf(?class,?c))
```

Description:
All overridden initialize methods must contain a super call to initialize.
Quantification:
$\forall$ override $\in$ *Overridden initialize methods*:
Predicate:
`methodWithSend(?init.method,variable(super),initialize,?)`

Appendix C

Intensional Views and Constraints over SmallWiki

C.1 Intensional views

C.1.1 SmallWiki Entities

The following intensional view groups all the entities in the implementation of SmallWiki.

Name: SmallWiki Entities	**Attributes:** class, method **Parent view:** None
Description: All classes and methods in the implementation of the SmallWiki, i.e. all the entities in the `SmallWiki` namespace.	

Intension:(SOUL)
`classInNamespace(?class,[SmallWiki]),`
`methodInClass(?method,?class)`	

Alternative intensions: /	

C.1.2 Wiki Structures

Name: Wiki Structures	**Attributes:** class **Parent view:** SmallWiki Entities
Description: The main structural elements which are part of a Wiki page like pages and chapters. This intensional view is defined by a single alternative capturing all classes in the hierarchy of the `Structure` class.	
Intension:(SOUL) `classInHierarchyOf(?class,Structure])`	
Alternative intensions: /	

Name: SmallWiki Entities	**Attributes:** class **Parent view:** Actioned Wiki Structures
Description: The structural elements for which there are corresponding actions.	
Intension:(SOUL) `WikiActions(?action,?),`	
Alternative intensions: /	

Name: Page Components	**Attributes:** class **Parent view:** SmallWiki Entities
Description: The different entities belonging to a page like tables, links, ...	
Intension:(SOUL) `classInHierarchyOf(?class,[PageComponent])`	
Alternative intensions: /	

C.1.3 Wiki Actions

The set of intensional views capturing actions on Wiki documents.

Name: Wiki Actions	**Attributes:** class, method **Parent view:** SmallWiki Entities
Description: All the actions which can be performed on Wiki elements (e.g. save, cancel,. . .). This intensional view is defined by two alternative views: 1. all the methods with as prefix "execute-" implemented on a class in the hierarchy of `Action` and 2. all the methods in the protocol action.	

Alternative intensions:(SOUL)
`1.classInHierarchyOf(?class,[Action]),`
`  ['execute*' match: ?method selector asString]`
`2.methodInProtocol(?method,action)`	

Name: Effective Actions	**Attributes:** class **Parent view:** Wiki Actions
Description: All concrete actions on wiki elements. Defined either as 1. all leaf classes in the `Action` hierarchy or 2. all classes which implement a a method named `execute`.	

Alternative intensions:(SOUL)
`1.not(superclassOf(?class,?))`
`2.methodWithNameInClass(?,execute,?class)`	

Name: Effective Actions	**Attributes:** class **Parent view:** Wiki Actions
Description: All actions that correspond with a Wiki Structure.	

Intension:(SOUL)
`WikiStructures(?structure),`
`[?structure name,'*' match:?class name]`	

Alternative intensions: /	

C.1.4 Wiki Visitor

An important part of the implementation of SmallWiki is based on the Visitor design pattern. The following intensional views capture the different entities which implement either a visited element or a visitor.

Visited Elements

Name: Wiki Visited Elements	**Attributes:** class, method **Parent view:** SmallWiki Entities
Description: All the elements of a Wiki document which can be visited. This intensional view is defined as 1. all the `accept:` methods in SmallWiki or 2. all the methods in the protocol `visiting`.	

Alternative intensions:(SOUL)
`1.methodWithNameInClass(?method,[#accept:],?class)`
`2.methodInProtocol(?method,[#visiting]),`
`  methodInClass(?method,?class)`	

Name: Wiki Storable Elements	**Attributes:** class **Parent view:** Wiki Visited Elements
Description: All the elements which can be persistently stored. This intensional view is defined as: 1. all the classes for which there exists a corresponding method on a *Storage Visitor* or 2. all Wiki Structures. This intensional view thus expresses the constraint that all Wiki Structures should be accepted by a storage visitor.	

Alternative intensions:()
`1.StorageVisitors(?,?visitmethod),`
`  [?visitmethod selector = ('accept', ?class name ,':') asSymbol]`
`2.WikiStructures(?class,?)`	

Name: Outputable Elements	**Attributes:** class **Parent view:** Wiki Visited Elements
Description: All elements which can be outputted to a web page. This intensional view is defined as all the visited elements which are accepted by an output visitor.	

Intension:(SOUL)
`OutputVisitors(?,?visitmethod),`
`[?visitmethod selector = ('accept', ?class name ,':') asSymbol]`	

Alternative intensions: /	

Visitors

The actual web page rendering mechanism and storage mechanism of SmallWiki are implemented as a Visitor over the elements in a Wiki document. This collection of Visitors is documented by the following three intensional views.

Name: Visitors	**Attributes:** class, method **Parent view:** SmallWiki Entities
Description: This intensional view groups all the behaviour concerning visiting of the elements of a Wiki document. As such, it is defined as all the methods classified in the protocol with as prefix `visiting`, implemented by a class in the hierarchy of `Visitor`.	

Intension:(SOUL)
`classInHierarchyOf(?class,[Visitor]),`
`methodInProtocol(?method,?protocol),`
`['visiting*' match:?protocol]`	

Alternative intensions: /	

Name: Storage Visitors	**Attributes:** class, method **Parent view:** Visitors
Description: The Visitors responsible for storing a Wiki document to e.g. an XML document, and so on. This view is defined by a single alternative capturing all Visitor entities in the `VisitorStore` hierarchy.	
Intension:(SOUL) `classInHierarchyOf(?class,[VisitorStore])`	
Alternative intensions: /	

Name: Output Visitors	**Attributes:** class, method **Parent view:** Visitors
Description: Similar to the intensional view above, a visitor is used to generate a web page from an object structure representing a Wiki document. To this end, all the Visitors in the hierarchy of `VisitorOutput` are used.	
Intension:(SOUL) `classInHierarchyOf(?class,[VisitorOutput])`	
Alternative intensions: /	

C.1.5 Other

In this section we provide the definition of two intensional views capturing the implementation of the Wiki server itself and the unit tests over SmallWiki.

Name: Wiki Server	**Attributes:** class **Parent view:** SmallWiki Entities
Description: All the classes in the hierarchy of `WikiServer`.	
Intension:(SOUL) `classInHierarchyOf(?class,[WikiServer])`	
Alternative intensions: /	

Name: Test Cases	**Attributes:** class **Parent view:** SmallWiki Entities
Description: All the test cases over SmallWiki.	
Intension:(SOUL) `classInHierarchyOf(?class,[XProgramming.SUnit.TestCase])`	
Alternative intensions: /	

C.2 Intensional Constraints

Over the above intensional views we defined the following intensional constraints:

Description:
For all Wiki structures for which actions are defined there should exist at least one corresponding action, i.e. an action for which the name of the wiki structure prefixes the name of the action.

Quantification:
∀ action ∈ *Actioned Wiki Structures*: ∃ structure ∈ *Structured Actions*:

Predicate:
```
((action valueFor:#class) name asString,'*')
   match:
    ((structure valueFor:#class) name asString)
```

Description:
For all classes in SmallWiki there should exist a corresponding test case. Notice that for this intensional view, all abstract classes are considered to be explicit deviations.

Quantification:
∀ entity ∈ *SmallWiki Entities*: ∃ test ∈ *Test Cases*:

Predicate:
```
((entity valueFor:#class) name asString,'*')
  match:
    ((test valueFor:#class) name asString)
```

Description:
For all outputable elements there must exist a method on an output visitor which has an argument matching the name of the outputable element.

Quantification:
∀ outputable ∈ *Outputable Elements*: ∃ output ∈ *Output Visitors*:

Predicate:
```
argumentsOfMethod(?arguments,?output.method),
member(?argument,?arguments),
['*',( ?outputable.class name asString),'*' match:?argument asString]
```

Description:
All output methods on the output visitor correspond with an outputable element. Notice that this binary intensional relation is the dual of the relation above.

Quantification:
$\forall$ output $\in$ *Output Visitors*: $\exists$ outputable $\in$ *Outputable Elements*:

Predicate:
```
argumentsOfMethod(?arguments,?output.method),
member(?argument,?arguments),
['*',( ?outputable.class name asString),'*' match:?argument asString]
```

Description:
All page components should be outputable.

Quantification:
$\forall$ component $\in$ *Page Components*: $\exists$ outputable $\in$ *Outputable Elements*:

Predicate:
```
component = outputable
```

Description:
For all storable elements there must exist a method on an storage visitor which has an argument matching the name of the storable element.

Quantification:
$\forall$ storable $\in$ *Wiki Storable Elements*: $\exists$ storage $\in$ *Storage Visitors*:

Predicate:
```
argumentsOfMethod(?arguments,?storage.method),
member(?argument,?arguments),
['*',( ?storable.class name asString),'*' match:?argument asString]
```

Description:
All storage methods on the storage visitor correspond with an storable element. Notice that this binary intensional relation is the dual of the relation above.

Quantification:
$\forall$ storage $\in$ *Storage Visitors*: $\exists$ storable $\in$ *Wiki Storable Elements*:

Predicate:
```
argumentsOfMethod(?arguments, ?storage.method),
member(?argument,?arguments),
['*',( ?storable.class name asString),'*' match:?argument asString]
```

Description:
All Wiki structures communicate with the Server.

Quantification:
$\forall$ structure $\in$ *Wiki Structures*: $\exists$ server $\in$ *Wiki server*:

Predicate:
```
methodInClass(?wikistructure, ?structure.class),
methodInClass(?wikiserver, ?server.class),
methodCallsMethod(?wikistructure, ?wikiserver)
```

Description: For all visited elements there must exist a method on an visitor which has an argument matching the name of the visited element.
Quantification: $\forall$ visited $\in$ *Wiki Visited Elements*: $\exists$ visitor $\in$ *Visitors*:

Predicate:
`argumentsOfMethod(?arguments, ?visitor.method),`
`member(?argument,?arguments),`
`['*',( ?visited.class name asString),'*' match:?argument asString]`

Description: All `accept:` methods implemented on a visited element invoke a method on the visitor.
Quantification: $\forall$ visited $\in$ *Wiki Visited Elements*: $\exists$ visitor $\in$ *Visitors*:
Predicate: `methodCallsMethod(?visited.method, ?visitor.method)`

Description: Visitor methods either write to a stream or invoke another visitor method.
Quantification: $\forall$ wikivisitor $\in$ *Visitors*:

Predicate:
`or(and(WikiVisitors(?,?visitor),`
`           methodCallsMethod(?wikivisitor.method,?visitor)),`
`   methodWithSend(?wikivisitor.method, variable(stream),?,?))`

Appendix D

Intensional Views and Constraints over DelfSTof

In this appendix we give an overview of the intensional views and constraints we declared over the DelfSTof case study.

D.1 Intensional views

D.1.1 DelfSTof entities

The following intensional view groups all the classes and methods belonging to DelfSTof:

Name: DelfSTof entities	**Attributes:** class, method **Parent view:** None
Description: All classes and methods belonging to the implementation of DelfSTof, i.e. all the classes and methods in the `ConceptAnalysis` namespace.	

Intension:(SOUL)
`classInNamespace(?class,[ConceptAnalysis]),`
`methodInClass(?method,?class)`	

Alternative intensions: /	

D.1.2 Filtering

This section gives an overview of the different intensional views which group source-code entities that belong to the implementation of the filtering mechanism in DelfSTof. We consider two kinds of filters: attribute filters that restrict the set of attributes considered for the concept analysis and concept filters that prune the set of identified concepts.

Name: Filters

Attributes: class, method
Parent view: DelfSTof entities

Description:
The source-code entities responsible for the filtering of attributes and concepts. This intensional view is defined as the set of all classes/methods in the category *Filters*.

Intension:(SOUL)

```
classInCategory(?class,Filters)
```

Alternative intensions: /

Name: Attribute Filters

Attributes: class, method
Parent view: Filters

Description:
This intensional view groups the entities implementing the filtering of attributes which are considered for the concept analysis. It is defined by two alternative views: 1. all the methods and classes in the hierarchy of `AttributeFilter` and 2. all the classes which implement the correct interface.

Alternative intensions:(SOUL)

```
1.classInHierarchyOf(?class,[AttributeFilter])
2.methodWithNameInClass(?,[#'validAttribute:'],?class),
  methodWithNameInClass(?,[#'generateAttributesFor:'],?class)
```

Name: Concept Filters

Attributes: class, method
Parent view: Filters

Description:
The tuples belonging to the extension of this view represent all the classes and methods which perform filtering on the resulting concepts outputted by the concept analysis. This view is defined as either 1. all the classes/methods in the hierarchy of `ConceptFilter` or either as 2. all the classes and their corresponding methods which implement a method `shouldInclude:`.

Alternative intensions:(SOUL)

```
1.classInHierarchyOf(?class,[ConceptFilter])
2.methodWithNameInClass(?,[#'shouldInclude:'],?class)
```

D.1.3 Concept Analyzers

The following two intensional views group the source-code entities which are responsible for the analysis of the concepts returned by the formal concept analysis algorithm. Each concept analyzer provides a means to pre-classify a concept based on a given property. E.g. DelfSTof contains analyzers for identifying concepts that consist out of polymorphic methods, and so on.

Name: Predefined Analyzers

Attributes: class, method
Parent view: DelfSTof entities

Description:
The *Predefined Analyzers* intensional view groups all the concrete concept analyzers in DelfSTof. It is defined as 1. all the classes which are an (indirect) subclass of `ConceptAnalyzer` and 2. all the classes which understand a certain set of messages.

Alternative intensions:(SOUL)

```
1.classBelow(?class,[ConceptAnalyzer])
2.classUnderstands(?class,[#'handleConcept:']),
  classUnderstands(?class,[#'canHandleConcept:'])
```

Name: Basic Analyzers

Attributes: class, method
Parent view: Predefined Analyzers

Description:
The basic analyzers are a subset of the predefined analyzers and represent all the classes which implement an analysis which groups a number of concepts based on primitive properties like the set of methods, parse trees, ...

Intension:(SOUL)

```
classBelow(?class,[BasicAnalyzer])
```

Alternative intensions: /

D.1.4 Context Creation

The intensional views in this section group a number of tuples representing the source-code entities involved in the set-up of the experiment: for each experiment which uses DelfSTof, a context is created which selects the proper set of filters and concept analyzers.

Name: Context Creation

Attributes: class, method
Parent view: DelfSTof entities

Description:
The main intensional view grouping all the entities which are involved in the creation of a context.

Intension:(SOUL)

```
classInCategory(?class,[#'Context Creation'])
```

Alternative intensions: /

Name: Context Creators

Attributes: class, method
Parent view: Context Creation

Description:
The classes and methods which implement a different context for the formal concept analysis algorithm. This intensional view is defined by two alternative views: 1. all the classes in the `ContextCreator` hierarchy and 2. all the classes which implement a certain set of methods.

Alternative intensions:(SOUL)

```
1.classInHierarchyOf(?class,[ContextCreator])
2.methodWithNameInClass(?,attributeFilterClasses,?class),
  methodWithNameInClass(?,basicAnalyzers,?class),
  methodWithNameInClass(?,conceptFilterClasses,?class),
  methodWithNameInClass(?,classificationAnalyzers,?class),
  methodWithNameInClass(?,attributeCreatorClass,[?class class])
```

Name: Classification Analyzers Creation

Attributes: method, analyzer
Parent view: Context Creators

Description:
The set of tuples representing a method which selects the Predefined Analyzers used in a context, together with each selected analyzer.

Intension:(SOUL)

```
methodWithName(?method,classificationAnalyzers),
methodReferencesClass(?method,?analyzer),
not(equals(?analyzer,[OrderedCollection ]))
```

Alternative intensions: /

Name: Attribute Filter Creation

Attributes: method, filter
Parent view: Context Creators

Description:
The set of tuples representing a method which selects the Attribute Filters used in a context, together with each selected filter.

Intension:(SOUL)

```
methodWithName(?method,attributeFilterClasses),
methodReferencesClass(?method,?filter),
not(equals(?filter,[OrderedCollection ]))
```

Alternative intensions: /

Name: Basic Analyzers Creation

Attributes: method, analyzer
Parent view: Context Creators

Description:
The set of tuples representing a method which selects the Basic Analyzers used in a context, together with each selected analyzer.

Intension:(SOUL)

```
methodWithName(?method,basicAnalyzers),
methodReferencesClass(?method,?analyzer),
not(equals(?analyzer,[OrderedCollection ]))
```

Alternative intensions: /

Name: Concept Filters Creation

Attributes: method, filter
Parent view: Context Creators

Description:
The set of tuples representing a method which selects the Concept Filters used in a context, together with each selected filter.

Intension:(SOUL)

```
methodWithName(?method,conceptFilterClasses),
methodReferencesClass(?method,?filter),
not(equals(?filter,[OrderedCollection ]))
```

Alternative intensions: /

Name: Predefined Context Creation

Attributes: method
Parent view: Context Creators

Description:
All methods responsible for selecting the context. This intensional view is defined as the union of the four intensional views we defined above.

Intension:(Smalltalk)

```
((Views.AttributeFilterCreation union: Views.BasicAnalyzersCreation)
  union: Views.ClassificationAnalyzersCreation)
     union: Views.ConceptFiltersCreation
```

Alternative intensions: /

D.1.5 Experiment initialization, parse tree attributes and concepts

Name: Experiment execution	**Attributes:** method **Parent view:** DelfSTof entities
Description: This intensional view consists out of all the methods which initialize the execution of an experiment using the FCA algorithm.	
Intension:(SOUL) `methodOfClassInProtocol(?method,[ContextCreator class],[#'run me'])`	
Alternative intensions: /	

Name: Concepts	**Attributes:** class, method **Parent view:** DelfSTof entities
Description: This intensional view groups all the classes and methods which represent a concept. These classes are used to represent the different types of pre-classified concepts and are created by the concept analyzers. This intensional view is defined by two alternative views: 1. all the classes in the package 'Concept Classes' and 2. all the classes in the category 'Helper Classes'.	

Alternative intensions:(SOUL)
`1. classInPackageNamed(?class,['Concept Classes'])`
`2. classInCategory(?class,[#'Helper Classes'])`	

Name: Parsetree Attribute Creator	**Attributes:** class, method **Parent view:** DelfSTof entities
Description: When using a set of methods as the objects under analysis by the FCA algorithm, often the attributes describing these methods are based on the parse tree of the methods. DelfSTof offers a set of parse tree attribute creators which extract a certain property from the parse tree of a method. This intensional view contains those classes and methods which implement such a parse tree attribute creator. It groups these entities based either on 1. all the classes in the `ParseTreeAttributeCreator` hierarchy or 2. all the classes in the package 'AttributeCreation' which contain the string "ParseTreeAttribute-Creator" in their class name.	

Alternative intensions:()
`1. classInHierarchyOf(?class,[ParseTreeAttributeCreator])`
`2. classInPackageNamed(?class,['AttributeCreation']),`
`   ['*ParseTreeAttributeCreator*' match:?class name]`	

Name: Parsetree Attribute Generator	**Attributes:** class, method **Parent view:** DelfSTof entities
Description: The parse tree attribute creator uses a visitor pattern in order to traverse a parse tree of a method. The entities implementing this visitor are grouped in this intensional view. It is defined as: 1. all the classes in the hierarchy of `AttributeGeneratorVisitor` or 2. all the classes in the package 'AttributeCreation' which contain the string "AttributeGenerator" in the class name.	

Alternative intensions:(SOUL)
`1. classInHierarchyOf(?class,[AttributeGeneratorVisitor])`
`2. classInPackageNamed(?class,['AttributeCreation']),`
`   ['*AttributeGenerator*' match:?class name]`	

Name: Chain Building Blocks	**Attributes:** class **Parent view:** DelfSTof entities
Description: The entities in the context creation make use of a chain in order to analyze/filter the concept/attributes. This intensional view captures all the classes representing a core building block of such a chain.	

Intension:(SOUL)
`or(equals(?class, [DefaultAnalyzer]),`
`   equals(?class, [NullConceptFilter]),`
`   equals(?class, [NullAttributeFilter]))`	

Alternative intensions: /	

D.2 Intensional Constraints

D.2.1 Implementation patterns and regularities

Description:
All methods initializing an experiment must be implemented in the same way.

Quantification:
∀ experiment ∈ *Experiment execution*:

Predicate:
```
statementsOfMethod(statements(<?statement>),?experiment.method),
equals(?statement,send(variable(self),[#runAnalysisOnObjects:forCase:],
     <send(variable(self),?,<>),literal(?project)>)),
[('runOn', ?project asString) match:(?experiment.method selector asString)]
```

Description:
All context creation methods must create a new `OrderedCollection` using either the `new` or `with:` messages.

Quantification:
∀ creator ∈ *Predefined Context Creation*:

Predicate:
```
or(methodWithSend(?creator.method,variable(OrderedCollection),new,?),
   methodWithSend(?creator.method,variable(OrderedCollection),[#'with:'],?))
```

Description:
All parse tree attribute creators make use of a parse tree generator (visitor).

Quantification:
∀ attribute ∈ *Parsetree Attribute Creator*: ∃ generator ∈ *Parsetree Attribute Generator*:

Predicate:
```
methodInClass(?method,?attribute.class),
methodWithSend(?method,variable([?generator.class name asSymbol]), new,?)
```

D.2.2 Regularities concerning correct use of the chain

Description: The basic building blocks of the chain must be used by the `AbstractContextCreator`. If not, the chain will not be correctly terminated.
Quantification: $\forall$ block $\in$ *Chain Building Blocks*:

Predicate:
`methodInClass(?method,[AbstractContextCreator]),`
`methodReferencesClass(?method,?block.class)`

Description: Context creators may not directly refer to a chain building block.
Quantification: $\forall$ creator $\in$ *Context Creators*: $\nexists$ block $\in$ *Chain Building Blocks*:

Predicate:
`methodInClass(?method,?creator.class),`
`methodReferencesClass(?method, ?block.class)`

Description: Basic Analyzers may not directly refer to a chain building block.
Quantification: $\forall$ analyzer $\in$ *Basic Analyzers*: $\nexists$ block $\in$ *Chain Building Blocks*:

Predicate:
`methodInClass(?method, ?analyzer.class),`
`methodReferencesClass(?method, ?block.class)`

Description: Attribute filters may not refer to the next filter in the chain.
Quantification: $\nexists$ filter $\in$ *Attribute Filters*:
Predicate: `methodWithSend(?filter.method,variable(self),nextFilter,?)`

Description: Concept filters may not refer to the next filter in the chain.
Quantification: $\nexists$ filter $\in$ *Concept Filters*:
Predicate: `methodWithSend(?filter.method, variable(self),nextFilter,?)`

D.2.3 Regularities governing context creation

The following four regularities encode the knowledge that each of the different kinds of context creation methods return a set of filters/analyzers of the correct kind.

Description:
Classification Analyzers Creation are only allowed to refer to Predefined Analyzers.

Quantification:
$\forall$ creator $\in$ *Classification Analyzers Creation*: $\exists$ analyzer $\in$ *Predefined Analyzers*:

Predicate:
```
(creation valueFor:#analyzer) = (analyzer valueFor:#class)
```

Description:
Attribute Filter Creation are only allowed to refer to Attribute Filters.

Quantification:
$\forall$ creator $\in$ *Attribute Filter Creation*: $\exists$ filter $\in$ *Attribute Filters*:

Predicate:
```
(creation valueFor:#filter) = (filter valueFor:#class)
```

Description:
Basic Analyzers Creation are only allowed to refer to Basic Analyzers.

Quantification:
$\forall$ creator $\in$ *Basic Analyzers Creation*: $\exists$ analyzer $\in$ *Basic Analyzers*:

Predicate:
```
(creation valueFor:#analyzer) = (analyzer valueFor:#class)
```

Description:
Concept Filters Creation are only allowed to refer to Concept Filters.

Quantification:
$\forall$ creator $\in$ *Concept Filters Creation*: $\exists$ filter $\in$ *Concept Filters*:

Predicate:
```
(creation valueFor:#filter) = (filter valueFor:#class)
```

Bibliography

[Ald05] J. Aldrich. Open modules: Modular reasoning about advice. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 144–168. Springer, 2005.

[BE] C. Bockisch and M. Eichberg. Bat (java bytecode analysis and manipulation library). http://www.st.informatik.tu-darmstadt.de/BAT.

[BE03] G. Back and D. Engler. MJ - a system for constructing bug-finding analyses for Java. Technical report, Stanford University, September 2003.

[Bec97] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[Bec99] K. Beck. *eXtreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[BG95] F. Bergadano and D. Gunetti. *Inductive Logic Programming: From machine learning to software engineering*. MIT Press., 1995.

[BKG$^+$06] J. Brichau, A. Kellens, K. Gybels, K. Mens, R. Hirschfeld, and T. D'Hondt. Application-specific models and pointcuts using a logic meta language. In *International Smalltalk Conference (ESUG)*, LNCS, pages 1–25. Spring-Verlag, 2006.

[BM98] E. Baniassad and G. Murphy. Conceptual module querying for software reengineering. In *International Conference on Software Engineering (ICSE)*, pages 64–73, 1998.

[BMMM98] W. Brown, R. Malveau, H. McCormick, and J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

[BMR$^+$96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[BMT99] W. Brown, H. McCormick, and S. Thomas. *AntiPatterns and Patterns in Software Configuration Management*. John Wiley & Sons, 1999.

[Bok99] B. Bokowski. Coffeestrainer: statically-checked constraints on the definition and use of types in java. In *European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 355–374. Springer-Verlag, 1999.

[Bra92] J. Brant. Hotdraw. Master's thesis, University of Illinois, 1992.

[Bri05] J. Brichau. *Integrative Composition of Program Generators*. PhD thesis, Vrije Universiteit Brussel, 2005.

[Bro87] F.P. Brooks. No silver bullet - essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[BvvT05] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwè. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL77)*, pages 238–252, 1977.

[CD99] J. Clark and S. DeRose. *XML Path Language (XPath)*. W3C Recommendation, 1999.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison Wesley, 2000.

[Che06] Checkstyle, December 2006. http://checkstyle.sourceforge.net.

[CHK$^+$05] W. Chung, W. Harrison, V. Kruskal, H. Ossher, S. Sutton, P. Tarr, M. Chapman, A. Clement, H. Hawkins, and S. January. The concern manipulation environment. In *International Conference on Software engineering (ICSE)*, pages 666–667, 2005.

[Cin07] Cincom. Visualworks smalltalk environment, 2007. http://www.cincom.com/smalltalk.

[CM93] A. Chowdhurry and S. Meyers. Facilitating software maintenance by automated detection of constraint violations. In *1993 Conference on Software Maintenance*, pages 262–271, 1993.

[Cod70] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cop92] J. Coplien. *Advance C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[DBD06] C. De Roover, J. Brichau, and T. D'Hondt. Combining fuzzy logic and behavioral similarity for non-strict program validation. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP06)*, pages 15–26. ACM Press, 2006.

[DBN$^+$07] C. De Roover, J. Brichau, C. Noguera, T. D'Hondt, and L. Duchien. Behavioral similarity matching using concrete source code templates in logic queries. In *ACM-SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM07)*, pages 92–102, 2007.

[DC03] C. Depradine and P. Chaudhuri. P^3: a code and design conventions preprocessor for java. *Software - Practice and Experience*, 33(1):61–76, 2003.

[DDMW00] T. D'Hondt, Kris De Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In *TACT Symposium*. Kluwer Academic Publishers, 2000.

[De 98] K. De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1998.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard Reference Manual*. Springer-Verlag, 1996.

[DFL$^+$05] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura, and M. Südholt. An expressive aspect language for system applications with arachne. In *Aspect-Oriented Software Development (AOSD)*, pages 27–38, 2005.

[DH98] K. De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Vrije Universiteit Brussel, 1998.

[DL05] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31:75–90, 2005.

[DMR92] C. Duby, S. Meyers, and S. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Technical Conference Proceedings*, pages 99–115. USENIX Assoc., 10-13 1992.

[DRW05] S. Ducasse, L. Renggli, and R. Wuyts. Smallwiki – a meta-described collaborative content management system. In *ACM International Symposium on Wikis (WikiSym)*, pages 75–82, 2005.

[Ede02] A. Eden. Lepus: A visual formalism for object-oriented architectures. In *Conference on Integrated Design and Process Technology (IDT)*, 2002.

[EGHYM94] D. Evans, J. Guttag, J. Horning, and T. Yang Meng. Lclint: A tool for using specifications to check code. In *SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96, December 1994.

[EH99] A. Eden and Y. Hirshfeld. Lepus – symbolic-logic modelling of object oriented architectures: A case study. In *Nordic Workshop on Software Architecture (NOSA)*, 1999.

[EKF03] A. Eden, R. Kazman, and J. Fox. Two-tier programming. Technical Report CSM-387, University of Essex, 2003.

[EMS$^+$04] M. Eichberg, M. Mezini, T. Schäfer, C. Beringer, and K.M. Hamel. Enforcing system-wide properties. In *Australian Software Engineering Conference*, pages 158–167. IEEE Computer Society Press, 2004.

[FBB$^+$99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Flo02] G. Florijn. Revjava - design critiques and architectural conformance checking for java software. Technical report, Software Engineering Research Centre (SERC), 2002.

[FM04] J. Fabry and T. Mens. Language independent detection of object-oriented design patterns. *Computer Languages, Systems and Structures*, 30(1-2):21–33, April-July 2004.

[FMv97] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 472–495, 1997.

[For82] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[GAA01] Y. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Conference on the Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 296–305, 2001.

[GB03] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Aspect-Oriented Software Development (AOSD)*, pages 60–69, 2003.

[GDN02] Y. Guéhéneuc, R. Douence, and J. Narenda. No java without caffeine – a tool for dynamic analysis of java programs. In *Automated Software Engineering (ASE)*, pages 117–126. IEEE Computer Society Press, 2002.

[GHG$^+$93] J. Guttag, J. Horning, K. Garland, A. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification*. Springer Verlag, 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80, The Language*. Addison-Wesley, 1989.

[Gu2] Y. Guéhéneuc. Three musketeers to the rescue – meta-modeling, logic programming, and explanation-based constraint programming for pattern description and detection. In *Workshop on Declarative Meta-Programming at ASE 2002*, 2002.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Spring-Verlag, 1999.

[HBZH05] L. He, H. Bai, J. Zhang, and C. Hu. Amuca algorithm for aspect mining. In *Software Engineering and Knowledge Engineering (SEKE)*, 2005.

[Hed97a] G. Hedin. Attribute extension - a technique for enforcing programming conventions. *Nordic Journal of Computing*, 4:93–122, 1997.

[Hed97b] G. Hedin. Language support for design patterns using attribute extension. In *Workshop on Language Support for Design Patterns and Frameworks (LSDF)*, 1997.

[HH06] D. Hou and J. Hoover. Using scl to specify and check design intent in source code. *IEEE Transcactions on Software Engineering*, 32(6):404–423, 2006.

[Hin01] H. Hind. Pointer analysis: haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE01)*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[HNB05] W. Havinga, I. Nagy, and L. Bergmans. Introduction and derivation of annotations in AOP: Applying expressive pointcut languages to introductions. In *European Interactive Workshop on Aspects in Software (EIWAS)*, 2005.

[HOST05a] W. Harrison, H. Ossher, S. Sutton, and P. Tarr. Concern modeling in the concern manipulation environment. In *Workshop on Modeling and analysis of concerns in software*, pages 1–5. ACM Press, 2005.

[HOST05b] W. Harrison, H. Ossher, S. Sutton, and P. Tarr. Supporting aspect-oriented software development with the concern manipulation environment. *IBM Systems Journal*, 44(2):309–318, 2005.

[HP04] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[HVd06] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*, pages 2–27. Springer, 2006.

[IBM] IBM. Eclipse. http://www.eclipse.org.

[JD88] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[JD03] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *International Conference on Aspect Oriented Software Development (AOSD)*, pages 178–187, 2003.

[JF88] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

[Joh79] S.C. Johnson. Lint, a c program checker. In M.D. McIlroy and B.W. Kemighan, editors, *Unix Programmer's Manual*, volume 2A. AT&T Bell Laboratories, seventh edition, 1979.

[Joh92] R. Johnson. Documenting frameworks using patterns. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 63–76. ACM Press, 1992.

[JUt07] JUtils. Lint4J, 2007. http://www.jutils.com/.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Grisworld. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 327–355. Springer Verlag, 2001.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtoir, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 220–242. Springer Verlag, 1997.

[KM05a] M. Kersten and G. Murphy. Mylar: a degree-of-interest model for ides. In *Aspect-Oriented Software Development Conference (AOSD)*, pages 159–168. ACM Press, 2005.

[KM05b] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *International Conference on Software Engineering (ICSE)*, pages 49–58. ACM Press, 2005.

[KM05c] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 195–213. Springer Verlag, 2005.

[KMBG06] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *European Conference on Object-Oriented Programming (ECOOP)*, number 4067 in LNCS, pages 501–525, 2006.

[KMT07] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-oriented Development (TAOSD)*, 2007.

[KR88] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[KS04] C. Koppen and M. Stoerzer. Pcdiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[Lad03] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[Lan03] M. Lanza. *Object-Oriented Reverse Engineering – Coarse-grained, Fine-grained and Evolutionary Software Visualization*. PhD thesis, University of Berne, 2003.

[Lee72] R. Lee. Fuzzy logic and the resolution principle. *Journal of the ACM*, 19(1):109–119, 1972.

[M S07] M Squared Technologies. Software source code static quality analysis, 2007. http://msquaredtechnologies.com/.

[Men00] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, October 2000.

[MGL06] N. Moha, Y. Guéhéneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *Conference on Automated Software Engineering (ASE)*, pages 297–300. IEEE Computer Society Press, 2006.

[Min91] N.H. Minsky. Law-governed systems. *Software Engineering Journal*, 6(5):285–302, 1991.

[Min96] N.H. Minsky. Law-governed regularities in object systems; part 1: Principles. *Theory and Practice of Object Systems (TAPOS)*, 2(4), 1996.

[MK06] K. Mens and A. Kellens. Intensive, a toolsuite for documenting and checking structural source-code regularities. In *Conference on Software Maintenance and Reengineering (CSMR)*, pages 239–248, 2006.

[MKPW06] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views: A case study. *Elsevier Journal on Computer Languages, Systems & Structures*, 32(2-3):140–156, 2006.

[MMM$^+$05] C. Marinescu, R. Marinescu, F. Mihancea, D. Ratiu, and R. Wettel. iplasma:an integrated platform for quality assessment of object-oriented design. In *Industrial track at the International Conference on Software Maintenance (ICSM)*, pages 77 – 80, 2005.

[MMW01] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Elsevier Journal on Expert Systems with Applications*, 23(4):405–431, 2001.

[MNS95] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT)*, pages 18–28. ACM Press, 1995.

[MP97] N.H. Minsky and P. Pal. Law-governed regularities in object systems; part 2: A concrete implementation. *Theory and Practice of Object Systems (TAPOS)*, 3(2):87–101, 1997.

[MP00] N.H. Minsky and P. Pal. Providing multiple views for objects. *Software - Practice and Experience*, 3(7):803–823, 2000.

[MT05] K. Mens and T. Tourwé. Delving source-code with formal concept analysis. *Elsevier Journal on Computer Languages, Systems & Structures*, 31(3-4):183–197, 2005.

[MvM04] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Working Conference on Reverse Engineering (WCRE)*, pages 132–141. IEEE Computer Society, 2004.

[MWD99] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 33–45. IEEE Computer Society Press, 1999.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[OM05] K. Ostermann and C. Mezini, M. Bockisch. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 214–240, 2005.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PMD06] PMD, December 2006. http://pmd.sourceforge.net.

[Ren03] L. Renggli. Smallwiki collaborateive content management. Manual, October 2003.

[Rie96] A. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[RM02] M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *International Conference On Software Engineering (ICSE)*, pages 406–416, 2002.

[RM03] M. Robillard and G. Murphy. Feat: a tool for locating, describing, and analyzing concerns in source code. In *25th International Conference on Software Engineering*, pages 822–823, 2003.

[Rob06] M. Robillard. Tracking and assessing the evolution of scattered concerns. In *Linking Aspect Technology and Evolution (LATE)*, 2006.

[RWW05] M. Robillard and F. Weigand-Warr. Concernmapper: Simple view-based separation of scattered concerns. In *Eclipse Technology Exchange at OOPSLA*, 2005.

[SG05] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 653–656, 2005.

[SGS$^+$05] K. Sullivan, W.G. Griswold, Y. Song, Y. Chai, M. Shonle, N. Tewari, and H. Rajan. On the criteria to be used in decomposing systems into aspects. In *Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference (ESEC/FSE)*. ACM Press, 2005.

[SR02] S. Sutton and I. Rouvellou. Modeling of software concerns in cosmos. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 127–133. ACM Press, 2002.

[SSC96] M. Sefika, A. Sane, and R. Campbell. Monitoring compliance of a software system with its high-level design models. In *International Conference on Software Engineering (ICSE)*, pages 387–397. IEEE Computer Society Press, 1996.

[STP05] D. Shepherd, T. Tourwé, and L. Pollock. Using language clues to discover crosscutting concerns. In *Workshop on the Modeling and Analysis of Concerns*, 2005.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 1986.

[TBKG04] T. Tourwé, J. Brichau, A. Kellens, and K. Gybels. Induced intentional software views. *Elsevier Journal on Computer Languages, Systems & Structures*, 30(1-2):35–47, 2004.

[TCH$^+$04] P. Tarr, W. Chung, W. Harison, V. Kruskal, H. Ossher, S. Sutton, A. Clement, M. Chapman, H. Hawkins, and S. January. The concern manipulation environment. In *Conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 29–30, New York, NY, USA, 2004. ACM Press.

[TM04] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 97–106. IEEE Computer Society, 2004.

[TOHS99a] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. Degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE)*, pages 107–119. ACM, 1999.

[TOHS99b] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE)*, pages 107–119. IEEE Computer Society Press, 1999.

[VSCD05] W. Vanderperren, D. Suvee, M. Cibran, and B. De Fraine. Stateful aspects in JAsCo. In *Software Composition (SC)*, LNCS, pages 167–181, 2005.

[WD04] R. Wuyts and S. Ducasse. Unanticipated integration of development tools using the classification model. *Elsevier Journal on Computer Languages, Systems & Structures*, 30:63–77, 2004.

[WM06] R. Wuyts and K. Mens. Codifying structural regularities of object-oriented programs. Technical Report 2006-06, Université catholique de Louvain, Belgium, 2006.

[Wuy01] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, January 2001.

[Wuy02] R. Wuyts. Starbrowser. Website http://homepages.ulb.ac.be/ rowuyts/StarBrowser/, 2002.

[Zad65] L. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.