Contents

Acknowledgements *4*

Introduction *5*

Chapter 2 - Reuse contracts *9*
- Reuse contracts 9
- Collaboration contracts 10
- Reuser clauses 11
  - Participant extension 11
  - Participant cancellation 12
  - Participant refinement 13
  - Participant coarsening 14
  - Context extension 15
  - Context cancellation 15
  - Context refinement 15
  - Context coarsening 16
- Evolution conflicts 17
  - Interface conflicts 17
  - Dangling reference conflicts 18
  - Conflicts concerning the calling structure 18

Chapter 3 – Reuse contracts at implementation level: problem analysis *19*
- HotDraw 19
  - Patterns in HotDraw 19
  - HotDraw and Java 20
  - HotDraw and Reuse Contracts 21
- Typing features of Java 23
  - Types 23
  - Methods 24
  - Class hierarchies 24
  - Interfaces 24
  - Type casts 25

Chapter 4 - Extended collaboration contracts *26*
- Collaboration contracts 26
- Participants 27
- Participant relations 30
- Signatures 33
- Acquaintance clauses 36
- Specialisation interfaces 38
- Validation 45
  - Collaboration Contracts extraction 49

Chapter 5 - Operators on extended collaboration contracts *51*
- Reuse contracts 52
- Reuser clauses 54
  - Participant extension reuser clauses 55
  - Participant cancellation reuser clauses 57
  - Participant refinement reuser clauses 58
  - Participant coarsening reuser clauses 59
  - Context extension reuser clauses 60
  - Context cancellation reuser clauses 61
  - Context refinement reuser clauses 62
  - Context coarsening reuser clauses 64
- Reuse contracts well-formedness 66

Chapter 6 – Evolution conflicts with extended reuse contracts *68*
A classification of conflicts 68
Duplicate modification conflicts 71
Double operation name conflict 71
Double participant name conflict 71
Double operation invocation conflict 71
Double participant-relationship conflict 72
Dangling reference conflicts 73
Dangling operation conflict 73
Dangling participant conflict 74
Dangling relationship conflict 74
Conflicts concerning the calling structure 76
Operation capture 76
Inconsistent operations 81
Unanticipated recursion 83
Hierarchy conflicts 85
Invalid acquaintance redefinition conflict 85
Inconsistent return-type conflict 85
Cyclic inheritance hierarchy conflict 86
Double inherited interface conflict 86
Masked operation conflict 87
Unanticipated redirection conflict 89
Implementation conflicts 93
Missing implementation conflict 93
Redundant specification conflict 93
Redundant implementation conflict 94
Type casts conflicts 96
Invalid type cast conflict 96
Scaling up 96

Chapter 8 - Conclusion *98*
Contribution 98
Evaluation 99
Related work 100
Future work 103
Formal extensions 103
Other extensions 104

Bibliography *105*

Appendix A – Formal Model *107*
Important notation conventions 107
Collaboration Contracts model 107
Name domains 107
Collaboration contracts 107
Participants 108
Classes & Interfaces 108
Acquaintances 108
Signatures 109
Specialisation interfaces 109
Hierarchies & Implementation relation 110
Well formedness 111
Mathematical definitions 113
Reuse contracts 113
Reuser clauses 115
Participant reuser clauses 115
Participant extension reuser clauses: 115
Participant cancellation reuser clauses: 116
Participant refinement reuser clauses: 117
Participant coarsening reuser clauses: 118
Context reuser clauses: 119
Context extension reuser clauses: 119

Context cancellation reuser clauses:.....120
Context refinement reuser clauses:.....121
Context coarsening reuser clauses:.....123

Appendix B – Table of conflicts *126*

Appendix C – HotDraw Model-View- Controller collaboration contract *130*

Acknowledgements

I thank Prof. Theo D'Hondt for advising this thesis and setting up the environment necessary for its development. Tom Mens worked by my side and was essential to develop and give shape to this work. I am specially thankful with Wolfgang De Meuter for his concerning about my thesis, particularly in early stages when a refocus of the subject was needed.

I thank all the people in the Programming Technology Lab for supplying a so good research environment. In particular Kris De Volder, Roel Wuyts and Patrick Steyaert for their help when I needed feedback from their experiences.

I thank Kim Mens, Bart Wouters for their academic help, but specially because together with Wolf they provided invaluable help in arranging everything for my accommodation when I moved to Brussels, and also later.

I thank Annya Romanczuk, for her setting up so many things in the courses period in Nantes.

Personally, there are few people to who I owe enormous gratitude. Mis padres fueron, como siempre, la base sólida que me dio seguridad y confianza para hacer las cosas. Mis hermanos, los más hermosos, se mantuvieron naturalmente cerca haciéndome sentir que la distancia era solamente física. La gente que tuve al lado desde que salí de Argentina fue por momentos vital para poder continuar y disfrutar esta experiencia dentro de otra cultura. A Natalia no voy a terminar nunca de agradecerle el increíble apoyo que me dio en los momentos más difíciles. María José con su comprensión y cariño incondicionales me hizo muchas veces sentir un impagable calor de familia. Además de mis 'medio hermanas', tuve la suerte de encontrarme con seres adorables. Miro, Andre, Ilse, y Thomas son los mejores ejemplos. Les agradezco a ellos todo el apoyo y afecto que me dieron. En especial a Thomas por darle una luz tan diferente a las cosas.

Introduction

The management of software change is one of the remaining big challenges to be faced by software development technology. The software evolves due to reasons of different nature, such as the inclusion of new or extended functional requirements, bug fixes and the need to achieve non-functional requirements, such as efficiency, reusability, maintainability, flexibility, adaptability, etc. Coping with different kinds of changes is indispensable because it extends the useful lifetime of software components, with the consequent increase in productivity. And also because, as has been largely discussed, it is impossible to predict at the moment of conception of a software component all its future uses and the possible changes that it will go through.

Software evolution is a complex issue, and the problems it yields are not always easy to solve. When a software component is evolved into a new version that has a problem to be integrated in the context of the original version, the original version is kept and the new one is left for new applications. This is an example where *version proliferation* problem appears when evolving software. Another difficulty is that the changes made to a certain software component in an evolution step need to be analysed in relation to the effect they may have in other interacting components. This problem is called *change propagation*, and the techniques that deal with it are known as impact analysis [BA96]. Another difficult problem is *architectural drift* [PW92] [DeHondt98], which arises when several evolution steps mainly at the source code level perform changes that make it drift away from the original architecture. This often occurs because the architectural documentation is insufficient, not easily available, or not explicitly linked to the code. This is in relation with another problem concerning documentation upon evolution. Update of documentation when a software component is changed is often neglected leading to *inconsistent documentation*.

Besides evolution, we distinguish reuse as another kind of change that software can be subject to. When concerned with reusable systems evolution becomes a key issue. When a component that has been reused evolves, it is necessary to ensure that the new version of the component still works in all the contexts where it was reused. Evolution occurs frequently in early stages of reusable systems, for as a component is reused it receives feedback coming from new insights in the domain and new insights in the design structure. These changes often need to be factorised in the base component, yielding upgrade versions of the base reusable system.

The schema in Figure 1.1 depicts the two kinds of change to which reusable systems and its component parts are exposed, in a simplified version where only one step of evolution and one reuse context is considered.

Figure 1.1

Evolution is applied to a software component that was conceived to be reused and consequently has some depending applications that customised the component to reuse it. Both reuse and evolution changes should be handled in a way that allows the evolved version of the component to be integrated with the customisations performed by the reuser.

Most classical approaches to reuse provide not enough flexibility to deal with unforeseen uses of the reusable components. The alternative proposed by object-oriented approaches to reuse (based on the inheritance mechanism) represents an interesting more flexible solution. On the other side, the inheritance mechanism alone is sometimes too open and does not impose sufficient discipline in its use. The single-inheritance mechanism provided by most frequently used object-oriented programming languages nowadays is not enough to support evolution in an acceptable way. For instance, when a change needs to be performed on a class, for efficiency reasons for example, this is most safely done by subclassing. A new subclass is created that specifies the differences with the parent class by defining overriding and new methods. If the need for another change to the class is detected, that has no relation with previous changes, then the process is repeated creating a new subclass that holds the new modifications. After this, problems come up when a requirement arises that involves both of the modifications performed to the class. The combination of both subclasses is not provided by the language, and some hand-made solution, frequently involving code replication, has to be carried out.

So far, we have briefly described the problems that arise in evolution in general and particularly in evolution of reusable systems, mentioning some general lines that could be used to tackle those problems. In this scenario, reuse contracts [Lucas97] are proposed as a new general approach aimed at managing reuse and evolution in a flexible yet structured way. The approach relies on explicit documentation of a component's functionality and its reuses. It defines a clear notation to be used by the component builder to provide information of the component he delivers, as well as for the component reuser to specify explicitly the way in which the component is reused.

The component's documentation specifies a *Collaboration Contract* that describes structural acquaintances of components as well as behavioural dependencies between the operations provided by those components. These component descriptions can be reused through a set of *reuse operators*. Reuse operators specify which assumptions are made over a Collaboration Contract to be reused, and which are the modifications made to it by the reuser. In this way it is possible to detect when a conflict in the composition occurs due to the breach of one of those assumptions.

Reuse contracts are intended to give support for a full-fledged methodology for disciplined reuse and evolution. After their whole presentation in [Lucas97], several topics such as formalisation of the model and tool support were developed in order to validate and further support the original formulation of the approach [DeHondt98] [Mens99a] [Mens99b]

An issue that still needs to be studied further within the reuse contract approach is the evolution of software at the implementation level. Evolution problems at the implementation level arise when an application is

written (in Java for example), using a framework that is later on upgraded by a new version, or when two software developers independently make modifications to the same code, and want to combine their changes afterwards. In practice, finding these conflicts manually is very time-consuming and error-prone, mainly because of the large amount of details present in source code.

When trying to apply the original definition of reuse contracts as defined in [Lucas97] to manage evolution at the implementation level, we encounter some restrictions that prevent the direct use of the model as it is.

For example, a property required by reuse contracts is that the code to which it is applied should respect the *Law of Demeter* [LH89]. This restriction basically states that the class structure should not be hard-wired inside methods, which concretely implies in the reuse contract model that messages can not be sent to the result of previous message sends. While this is a requirement for good design, in practice it turns out to be too restrictive. By inspecting some 'good quality' sample code such as HotDraw, the framework for drawing editors developed at UIUC [HotDraw] and source code of some running system [Wuyts99], we found that cascaded method invocations are frequently employed. Furthermore, the employment of other recommendable design techniques such as the definition of accessor functions to inspect and modify the attributes of a class clearly conflicts with the *Law of Demeter*, since a message send to the result of an accessor necessarily introduces a cascaded method invocation.

There are also other features proper of implementation-level software artefacts that are not expressible in the current definition of reuse contracts. The reason for this is that the general model is intended to be generally applicable to several stages of the life cycle of software development. This determines a formulation that sometimes happens to be too abstract, neglecting specific issues needed when dealing with evolution of source code.

In order to provide support for implementation-level characteristics it is necessary to have more expressiveness in the description of Collaboration Contracts so that elements of the implementation phase can be represented, and also a richer set of modification operators so that evolution of these added elements can be expressed.

Additionally, most of the work developed up to now for reuse contracts does not deal with typing features. Type systems are useful to characterise precisely and concisely the behaviour of software components to be reused. On the other hand, the approach of behavioural subtyping for customisation of software components proposed by formal approaches [LW94] [DL96] is not flexible enough to deal with reuse and evolution as it is needed in real practice. Reuse contracts are proposed as an alternative way to cope with reuse and evolution. They do not exhibit the restrictions of type systems, and at the same time remain a rigorous method. Nevertheless, for representing evolution of source code in typed languages it would be useful to have a way to represent some abstraction of the type system provided by the programming language.

The intention of this work is therefore to analyse if an approach applicable to typed code in the style of Java programs can be defined that is as useful and intuitive as the reuse contracts. The goal is to extend the reuse contracts model with features necessary to appropriately express and manage mechanisms present at the implementation level of object-oriented programs. In order to do that we need to incorporate in the model the possibility to represent cascaded method invocations, the explicit handling of types in method results and acquaintances, and the representation of inheritance hierarchies. These and other features added to the Collaboration Contracts yield an added expressive power in the abstraction model, that results in the detection of interesting source code evolution conflicts.

We aim at providing new features indispensable to deal with source code, but without loosing any of the features supplied already by the original definition of reuse contract. For that reason we perform a conservative extension to the original model. This means that all the features that can be expressed in the original reuse contracts can still be modelled in our extended formulation, plus some new added characteristics.

To ensure a clear and precise understanding of the semantics of the features we represent in Collaboration Contracts, we define a formal model of it. We use a function domain system to describe mathematically all the elements in the Collaboration Contracts as well as the modification operators we define. This allows to express concisely the result of operator application. Consequently, it helps in defining precisely the conditions for evolution conflict identification.

To validate the expressive power of the Collaboration Contracts model, we use a case study. We choose the Java version of HotDraw. It is a well-structured framework with extensive use of design patterns, that has been widely analysed and referred to in the literature [Richner99] [Brant98]. We select a set of HotDraw classes and interfaces that implement the Model-View-Controller architectural pattern. Then, we apply our Collaboration Contract definition to that piece of code to check if the structure of the pattern it implements is still present in the resulting abstraction.

The work we present here represents a step towards the application and validation of reuse contracts in real evolving systems.

The structure of the dissertation is as follows. In chapter 2 we present the basic original formulation of reuse contracts, describing the collaboration contracts, the modification operators and the detectable conflicts. We analyse the problems that arise when trying to apply that definition of reuse contracts to source code, in chapter 3. There we introduce and use the HotDraw framework as case study. After that we briefly describe some of the typing features present in Java, which will serve as a specification for some of the features we need to add to the model. In the three subsequent chapters we present our extended model. We define our model for Collaboration Contracts, together with the relevant parts of the formalism in chapter 4. In chapter 5 we present the modification operators we identify, and in chapter 6 we describe and characterise some of the conflicts that the presented model allows to detect automatically. Finally chapter 8 concludes, compares our work with some related approaches, and outlines some lines of further work.

Chapter 2 - Reuse contracts

In this chapter we will present the definition of reuse contracts, which is the base of our work. We will not provide a detailed description of the model, but only the main elements needed to understand the concepts we introduce.

Reuse contracts propose an extended interface description mechanism to document reusable components. They are based on the idea that making explicit the assumptions about the co-operation between software components that are usually implicit, is fundamental for change propagation and impact analysis. Reuse contracts involve two parties: *providers* of reusable components, and *reusers* of those components. Providers supply reuse information about a software component, and reusers state the way in which they modify it. By establishing predefined ways in which components can be modified, it is possible to define rules to detect when changes to a reusable component breach the assumptions made by a reuser on it.

Reuse contracts enforce discipline reuse without being too coercive. The simplicity and intuitiveness of the model are remarkable qualities that make it is easy to learn and use in real development environments. Moreover, one of the key advantages of reuse contracts is that they can be incorporated straightforward in tools because they deal mainly with static information of control flows. They do not depend on sophisticated techniques such as data-flow analysis or deadlock detection to find inconsistencies.

Reuse contracts were first introduced in [SLMD96], as a way to deal with the fragile base class problem in class hierarchies. Carine Lucas in her Ph. D. dissertation [Lucas97] established a complete definition of the model with the basic ideas, motivation and terminology. After that several works were presented that improve and apply the model to in different directions. [DeHondt98] defines a software classification model as the basis for a Classification Browser tool that aids developers in extracting and organising collaboration contracts and reuse contracts. [Mens99a] sets up a formal foundation for reuse contracts in a domain independent way, that helps in making the approach scalable. In [Mens98] and [Mens99b], reuse contracts are integrated in UML to deal with evolution of collaboration diagrams, and in [Mens99c] an extension of the UML metamodel with reuse contracts is defined, to integrate reuse and evolution in all kinds of UML models. Other research lines include the application of reuse contracts to the management of evolving requirements [D'Hondt98] and the implementation of a tool that extends Java to be able to express reuse contracts [Cornelis97].

In this work we present the reuse contracts model as defined in [Lucas97], but with the redefined terminology as in [DeHondt98].

Reuse contracts

A reuse contract is a contract between a provider and a reuser. It consists of a name that identifies it, a *contract type* and two clauses.

The two clauses are a *provider clause* and a *reuser clause*. The provider clause describes a collaboration between components co-operate in order to achieve a certain task. The provider clause is a *collaboration contract*. The reuser clause provides information of how the provider clause is modified.

The *contract type* of reuse contracts identifies the kind of modification represented by the contract. It is an annotation that describes the relationship between the provider and the reuser clauses.

The reuser clause specifies elements of a collaboration contract that take part in the modification. The interpretation of the contents of the reuser clause depends on the contract type of the reuse contract.

There are eight basic reuse contracts types, and consequently eight kinds of reuser clauses. They can be divided in two groups according to the kind of modelling elements they affect in the provider collaboration contract.

The different types of reuse contracts will be explained together with the kinds of reuser clauses. We will first describe the *collaboration contract* model.

Collaboration contracts

A collaboration contract consists of a number of participants that interact. The collaboration contract consists of a name to identify it and a number of participants. Participants can represent classes, parts of classes, modules, components, subsystems, software layers, etc. The collaboration contract should be well-formed, so that it does not reference participants or operations that are not defined in the collaboration contract.

A participant consist of

- a name, that identifies them inside the collaboration contract
- an *acquaintance clause*
- an *interface*.

The *acquaintance clause* of a participant states with which other participants in the collaboration contract the participant is acquainted with. It consists of *acquaintance relationships*, that are directed relationships between participants. An *acquaintance relationship* represent any kind of relationship in actual code, from an association or a parameter binding to the transitive closure of a series of acquaintance relationships. An acquaintance relationship has a name, that is the name with which the participant owning the acquaintance relationship refers to the associated participant.

When the acquaintance clause of a participant *p* contains *a.q* we say that *a* on *p refers* to *q* or simply that *p refers* to *q*.

The set of participants in a collaboration contract, together with the acquaintance relationships between them is called the *context* of the collaboration contract.

The *interface* of a participant is a set of operations each consisting of:

- an operation name that is unique within the interface
- a *specialisation clause*

A *specialisation clause* is a set of operation invocations *a.m*, where a is an acquaintance name and m is an operation name.

The well-formedness of a collaboration contract is defined:

Definition

A collaboration contract CC is *well-formed* if for each participant p in CC the following conditions hold:

- for each acquaintance relationship *a.q* in the acquaintance clause of p a participant with name *q* exists in CC
- for each operation invocation *a.m* in a specialisation clause in p:
  - (a) *a* is an acquaintance name in the acquaintance clause of p
  - (b) *m* is the name of an operation in the interface of the participant a refers to

Law of Demeter

The Law of Demeter [LH89] is a design guideline for object oriented systems that restrict the invocations that should be performed by a method:

'An operation O of a class C should only call operations of the classes (called preferred supplier classes) of the following objects: immediate subparts of the current object, argument objects of O (including self), objects created by O'

This principle is inherently present in the original definition of reuse contracts, since operations can only invoke operations of its direct acquaintances.

Reuser clauses

We said that a contract type is an annotation to describe the kind of modification taking place. The types are divided into two groups: participant types and context types. The former correspond to changes in the description of participants in a collaboration contract, and the latter correspond to changes in the context of the collaboration contract. The two kinds of types represent addition or removal of all of the four modelling elements described by a collaboration contract. Participant types express addition or removal of operations and operation invocations, and context types express addition or removal of participants or acquaintance relationships.

Each kind of contract type has its associated reuser clause. A reuser clause together with a contract type form a *modification operator* (also called reuse operator).

The names and meaning of the eight basic modification operators is described in the following table:

Basic modification operator	Meaning
Participant extension	adding new operations
Participant cancellation	removing operations
Participant refinement	adding new operation invocations
Participant coarsening	removing operations invocations
Context extension	adding new participants
Context cancellation	removing participants
Context refinement	adding new acquaintance relationships
Context coarsening	removing acquaintance relationships

The changes represented by a reuser clause are applied to the collaboration contract in a provider clause to obtain a new collaboration contract that describes the adapted collaboration contract. Reuser clause definition consist of three parts: the definition of the structure of the reuser clause (the elements it consists of), the conditions for applicability of the reuser clause to a given collaboration contract, and the result collaboration contract after the application of the reuser clause. The definitions ensure that the collaboration contract that results from applying a reuser clause on a well-formed collaboration contract is well-formed.

We will now present the eight basic kinds of reuser clauses.

Participant extension

A participant extension reuser clause describes how the interfaces of participants are extended with new operations.

Definition

A *participant extension reuser clause* is a set of pairs (p, int), each consisting of a participant name p and an interface int.

Definition

A collaboration contract CC is *participant extendible* by a participant extension reuser clause R if for each pair (p, int) in R:

- p is a participant name in CC
- no operation in int appears in the interface of participant p in CC
- for each operation invocation a.m in a specialisation clause in int:
    - a is an acquaintance name in the acquaintance clause of p in CC
    - if a on p refers to q then m is an operation in the interface of q in R

Definition

If a collaboration contract CC is participant extendible by a participant extension reuser clause R, then the collaboration contract CC2 is the *participant extension* of CC by R, where:

- CC2 contains all the participants of CC that are not mentioned in R
- for each (p, int) in R: CC2 contains a participant with the same name and acquaintance clause as p in CC and that contains all operations of p, plus int.

The definitions above state that a participant extension reuser clause can only add operations to existing participants in a collaboration contract, and that they can only add new operations to them. Besides that, the operation invocations added in the specialisation clause of each added operation should refer to acquaintance names in the participant, and the operation they invoke should exist in the interface of the target acquaintance.

The result of the application of a participant extension is a collaboration contract with the same participants as in the original collaboration contract, but where the participants referred in the reuser clause contains an extended interface.

Participant cancellation

A participant cancellation reuser clause describes how the interfaces of a set of participants are reduced.

Definition

A *participant cancellation reuser clause* is a set of pairs (p, int), each consisting of a participant name p and an interface int.

Definition

A collaboration contract CC is *participant cancellable* by a participant cancellation reuser clause R if for each pair (p, int) in R:

- p is a participant name in CC and each operation in int is identical to an operation in this participant in CC
- for all operations m, n and for all participants q in CC such that m on q invokes n on p: if n is an element of int, then m appears associated with q in R

Definition

If a collaboration contract CC is participant cancellable by a participant cancellation reuser clause R, then the collaboration contract CC2 is the *participant cancellation* of CC by R, where:

- CC2 contains all the participants of CC that are not mentioned in R
- for each (p, int) in R: CC2 contains a participant with the same name and acquaintance clause as p in CC and that contains all operations of p, except for those in int.

The definitions above state that a participant cancellation reuser clause is only applicable if it refers operations existing in the indicated participants. In addition operations can only be removed if they are not referred to in any specialisation clause in the collaboration contract on which the reuser clause is applied..

The result of the application of a participant cancellation is a collaboration contract with the same participants as in the original collaboration contract, but where the participants referred in the reuser clause contains a reduced interface.

Participant refinement

A participant refinement reuser clause specifies addition of operation invocations to the specialisation clause of operations.

Definition

A *participant refinement reuser clause* is a set of pairs (p, exint), each consisting of a participant name p and an extended interface exint. An extended interface is a set of operations, each consisting of an operation name and two disjoint specialisation clauses. The first is called the *repeating specialisation clause* (or context specialisation clause), and repeats the specialisation clause in the base collaboration contract. The second specialisation clause describes the operation invocations that need to be added.

Definition

A collaboration contract CC is *participant refinable* by a participant refinement reuser clause R if for each pair (p, exint) in R:

- p is a participant name in CC
- for each operation name m in exint: m appears in participant p in CC and m's repeating specialisation clause in exint is identical to the specialisation clause of m in p in CC
- for each operation invocation a.m in a second specialisation clause in exint:
    - a is an acquaintance name in the acquaintance clause of p in CC
    - m is an operation in the interface of the participant referred to by a in p in CC

Definition

If a collaboration contract CC is participant refinable by a participant refinement reuser clause R, then the collaboration contract CC2 is the *participant refinement* of CC by R, where:

- CC2 contains all the participants of CC that are not mentioned in R
- for each (p, exint) in R: CC2 contains a participant:
    - with name p and the same acquaintance clause as p in CC
    - that contains all operations of p in CC not mentioned in exint
    - that contains all operations in exint with as specialisation clause the union of their two specialisation clauses in exint.

The repeating specialisation clause in the reuser clause is used to restrict the cases in which the reuser clause is applicable: only when the participant contains a specialisation clause equal to it, will the reuser clause be applicable. This is defined in this way to allow the detection of conflicts.

The result of the application of a participant refinement is a collaboration contract with the same participants as in the original collaboration contract, and the same operations for each of them, except that the ones mentioned in the reuser clause have an extended specialisation clause for the specified operation.

Participant coarsening

A participant coarsening reuser clause specifies removals of operation invocations from the specialisation clause of operations.

Definition

A *participant coarsening reuser clause* is a set of pairs (p, exint), each consisting of a participant name p and an extended interface exint. The first specialisation clause specifies which invocations are retained, while the second denotes the invocations that are removed.

Definition

A collaboration contract CC is *participant coarsenable* by a participant coarsening reuser clause R if for each pair (p, exint) in R:

- p is a participant name in CC
- for each operation name m in exint:
    - m appears in participant p in CC
    - the union of m's specialisation clauses in exint is identical to the specialisation clause of m in CC.

Definition

If a collaboration contract CC is participant coarsenable by a participant coarsening reuser clause R, then the collaboration contract CC2 is the *participant coarsening* of CC by R, where:

- CC2 contains all the participants of CC that are not mentioned in R
- for each (p, exint) in R: CC2 contains a participant:
    - with name p and the same acquaintance clause as p in CC
    - that contains all operations of p in CC not mentioned in exint
    - that contains all operations of exint with as specialisation clause the first of the specialisation clauses in exint.

The double specialisation clause in the reuser clause has a similar purpose than it has in participant refinement reuser clauses. It makes the participant coarsening context sensitive: it is only applicable on a collaboration contract that holds operations with an expected sc. This is also used in the detection of conflicts.

Other three participant specialisation clause are defined in [Lucas97] for the application of reuse contracts to UML, that we don't describe here because they are not part of the basic operators. They are: participant specialisation, participant abstraction and participant concretisation. The first one correspond to a context insensitive refinement (used to represent a form of super invocations), that we don't consider because we will represent super and self invocations in a more homogeneous way. The other two refer to abstract/concrete annotations of operations, that we don't consider in this work because we prefer to first analyse the extension of the basic model, and the study the integration with other features.

Context extension

A context extension reuser clause specifies addition of participants to a collaboration contract.

Definition

A *context extension reuser clause* is a well-formed collaboration contract.

Definition

A collaboration contract CC is *context extendible* by a context extension reuser clause R if for each participant p in R:

- p's name is different from all participant names in CC

Definition

If a collaboration contract CC is context extendible by a context extension reuser clause R, then the collaboration contract CC2 is the *context extension* of CC by R, where:

- CC2 contains all the participants in CC and all participants in R

The only restriction for context extension is that the added participants do not exist in the original collaboration contract.

Context cancellation

A context cancellation reuser clause specifies removal of participants from a collaboration contract.

Definition

A *context cancellation reuser clause* is a well-formed collaboration contract.

Definition

A collaboration contract CC is *context cancellable* by a context cancellation reuser clause R if for each participant p in R:

- p is identical to a participant in CC
- p does not appear in the acquaintance clause of a participant in CC that is not in R

Definition

If a collaboration contract CC is context cancellable by a context cancellation reuser clause R, then the collaboration contract CC2 is the *context cancellation* of CC by R, where:

- CC2 contains all the participants in CC except for those named in R

A context cancellation can only be applied to a collaboration contract if it lists participants that are equal to participants in the collaboration contract, and if those participants are not acquainted by remaining participants.

Context refinement

A context refinement reuser clause specifies acquaintance relations that should be added to acquaintance clauses of existing participants in the collaboration contract.

Definition

A *context refinement reuser clause* contains triples of the form (p, acq1, acq2), each consisting of a participant name, and two disjoint acquaintance clauses.

Definition

A collaboration contract CC is *context refinable* by a context refinement reuser clause R if for each triple (p, acq1, acq2) in R:

- p is a participant name in CC
- acq1 is identical to the acquaintance clause of p in CC
- acq2 contains acquaintance relationships a.q, where a is different from all acquaintance names in acq1 and q is a participant name in CC

Definition

If a collaboration contract CC is context refinable by a context refinement reuser clause R, then the collaboration contract CC2 is the *context refinement* of CC by R, where:

- CC2 contains all participants of CC that are not mentioned in R
- for each triple (p, acq1, acq2) in R: CC2 contains a participant with the same name and interface as p in CC and the union of acq1 and acq2 as acquaintance clause

The context refinement is also context sensitive, it only adds acquaintance relationships to participants whose acquaintance clause is as expected.

Context coarsening

A context coarsening reuser clause specifies acquaintance relations that should be removed from acquaintance clauses of existing participants in the collaboration contract.

Definition

A *context coarsening reuser clause* contains triples of the form (p, acq1, acq2), each consisting of a participant name, and two disjoint acquaintance clauses.

Definition

A collaboration contract CC is *context coarsenable* by a context coarsening reuser clause R if for each triple (p, acq1, acq2) in R:

- p is a participant name in CC
- the union of acq1 and acq2 is identical to the acquaintance clause of p in CC
- for all a.q in acq2: no operation in p has a in its specialisation clause.

Definition

If a collaboration contract CC is context coarsenable by a context coarsening reuser clause R, then the collaboration contract CC2 is the *context coarsening* of CC by R, where:

- CC2 contains all participants of CC that are not mentioned in R
- for each triple (p, acq1, acq2) in R: CC2 contains a participant with the same name and interface as p in CC and acq1 as acquaintance clause

The context coarsening is also context sensitive, it only removes acquaintance relationships to participants whose acquaintance clause is as expected. Moreover, it requires that no operation of an affected participant contains an invocation to the removed acquaintance relationship.

Evolution conflicts

When the above described modification operators are applied simultaneously to the same collaboration contract by independent evolvers (or reusers), there can be problems. The problems may arise when trying to combine the modifications performed independently by two or more evolvers. These inconsistencies problems that arise in the combination of parallel modifications are called *evolution conflicts*.

Evolution conflicts occur when the assumptions made by one of the evolvers are breached when combining with the other modifications. Evolution conflicts can be characterised according the type of modification of each of the involved changes, and the elements described by the reuser clauses.

In [Lucas97] a classification of conflicts is done identifying three categories of conflicts: *interface conflicts*, *dangling reference conflicts* and *conflicts concerning the calling structure*. All of them except one are detectable only by checking the type of the modification operator. For one conflict it is necessary to also inspect the original collaboration contract on which both modifications are performed.

In this section we will only describe briefly and informally each of the categories, a deeper description of each of the conflicts is found in [Lucas97]. We will analyse further the evolution conflicts detectable with the original reuse contracts in chapter 6.

Each conflict refers to two modification operators M1 and M2 that are applied independently on the same base collaboration contract.

Interface conflicts

Interface conflicts are conflicts concerning the interface i.e. conflicts of operation names, participant names, etc. They occur when two operators adding the same kind of information to the base collaboration contract. Since there are four kinds of informations in a reuse contract (participants, acquaintances, operations and operation invocations) there can be four kinds of interface conflicts. Since the four kinds of informations are added by four different basic modification operators, the rules for interface conflict detection are straightforward.

Operation name conflict

Occurs when M1 is a participant extension adding an operation m to a participant p, and M2 is also a participant extension adding the same operation to the same participant.

Participant name conflict

Occurs when M1 is a context extension adding a participant p, and M2 is also a context extension the same participant.

Double operation invocation conflict

Occurs when M1 is a participant refinement (coarsening) extending (reducing) the specialisation clause of an operation m of a participant p, and M2 is also a participant refinement (coarsening) also extending (reducing) the specialisation clause of the same operation of the same participant.

Acquaintance relationship conflict

Occurs when both M1 and M2 represent context refinements or context coarsenings of the acquaintance clause of a participant p, and the repeating acquaintance clause of p in M2 is not identical to the resulting acquaintance clause of p in M1.

Interface conflicts are detectable thanks to the repeating clauses in the reuser clauses (repeating specialisation clause or repeating acquaintance clause, depending on the reuser clause). Except for acquaintance relationship conflict, for the other interface conflicts it holds that if both operators are applicable on a collaboration contract, its because their repeating clauses match the ones in the collaboration

contract. After applying one of the operators, the corresponding clause will be extended and the second operator will not be applicable any more.

Dangling reference conflicts

These conflicts occur when an operator removes an item from the interface, while the other operator continues to refer to it.

Dangling operation conflict

Occurs when an operation is removed by M1 and M2 refers to this removed operation. This can only happen when M1 is a participant cancellation and M2 is a participant cancellation, refinement or coarsening.

Dangling participant conflict

Occurs when a participant is removed by M1 and M2 refers to this removed participant. This can only happen when M1 is a context cancellation. M2 can be any possible operator except for context extension, because context extensions do not refer to existing participants.

Dangling acquaintance conflict

Occurs when an acquaintance relationships is removed by M1 and M2 refers to this removed acquaintance relationship. This can only happen when M1 is a context coarsening and M2 is a participant extension, refinement or coarsening.

Conflicts concerning the calling structure

Operation capture

Occurs when M1 changes the specialisation clause of an operation m in a participant p (i.e. it is a participant refinement or participant coarsening), and M2 adds to the specialisation clause of an operation n, an invocation to operation m in p (i.e. M2 is a participant refinement). This kind of operation capture is called *regular operation capture*. Another kind of operation capture is *accidental operation capture*, that occurs when the operation m did not exist in the base collaboration contract, and both operators introduce it (i.e. they are both participant extensions). Accidental operation capture always involves an operation name conflict, since both operators are adding operation m.

Inconsistent operations

This conflict represents the counterpart of operation capture conflict. It occurs when M1 changes the specialisation clause of an operation m in a participant p i.e. it is a participant refinement or coarsening, while M2 removes an invocation to m in p i.e. it is a participant coarsening.

Unanticipated recursion

Unanticipated recursion occurs when after two separate augmentations of the specialisation clauses of two separate operations i.e. M1 and M2 are participant refinements, these operations show mutually recursive behaviour. In the collaboration contract resulting from combining the two participant refinements there is a cycle formed by operation invocations links, this cycle could be of any length. It could be the case that the two participant refinements add the links that were missing in the base collaboration contract to close the cycle. Therefore, to determine an unanticipated recursion conflict it does not suffice to consider only the two operators, the base collaboration contract need also to be taken into account.

Chapter 3 – Reuse contracts at implementation level: problem analysis

In this chapter we investigate how reuse contracts can be applied at implementation level. First we present our case study, HotDraw for Java, a framework for drawing editors written in Java. We describe the main characteristics of HotDraw and then describe in detail a particular pattern implemented in the code that will be used throughout this dissertation. After that we analyse the application of reuse contracts as they are defined in [Lucas97] to the HotDraw code. We find some restrictions that we organise in a list of requirements that need to be fulfilled by reuse contracts in order to be applicable to source code. Finally, we will describe some technical issues about some type features of Java, since it will serve as a specification for some of the previously identified requirements.

HotDraw

HotDraw is a two-dimensional graphics framework for structured drawing editors. It can be used to build specialised drawing tools for schematic diagrams, blueprints or program design. It allows to define new figures and special manipulation tools for the drawings. The drawing editors that are created can be stand-alone applications or they can be part of larger systems. HotDraw has been used to create many different editors, from CASE tools to a HyperCard clone [Brant98].

HotDraw has been a very popular framework for a variety of reasons: the availability of its source code, and the existence of several existent implementations that have been studied by many developers. It was used as starting point to write specific applications, benefiting from having the source code available.

The first version of the framework was implemented by Kent Beck and Ward Cunningham in VisualWorks Smalltalk. Some years later a prototype version was written in Java, and currently there exist more refined implementations in that language. The first name of the Java version was HotDraw for Java and then it was renamed as Drawlets. In this work we will refer to it as HotDraw for Java (or simply HotDraw) because it is the most well known name. We use the Java version that is available in [RoleModel]

Patterns in HotDraw

The use of patterns in developing frameworks in Java encourage extendible and understandable frameworks. HotDraw was used as an illustrative example of how patterns can be used for the documentation of frameworks [Johnson92]. In HotDraw for Java an intensive use of design patterns [GHJV94] and implementation patterns [Beck97] is made. It is claimed that a lot of code documentation in frameworks can be reduced through the use of patterns.

As frameworks are reusable design solutions to be used in a number of different applications inside a given domain, they are more abstract than most software. This, together with the fact that there are different kinds of users of the framework, makes the documentation of frameworks a complex issue. Patterns aid in all the

kinds of documentation that are needed for a framework, that is: documentation of the purpose of the framework, documentation of how to use the framework, and documentation of the detailed design of the framework.

According to [Johnson92] each pattern documenting a framework provides a description of the purpose of the pattern, i.e. the problem for which it was conceived. Several examples of uses of the pattern are then described in order to guide the user in what kind of situations the pattern is applicable, and what are the concrete actions to carry on. Finally the design parts of the framework and more details about how that part of the framework works are provided.

Model-View-Controller

HotDraw is based on the Model-View-Controller triad [KP88]. We decided to take the kernel classes and interfaces implementing the roles in the Model-View-Controller as case study for our work. The fact that it is good quality code makes it ideal to study the application of reuse contracts to source code.

We will use the classes and interfaces implementing the Model-View-Controller in HotDraw mainly to validate the expressiveness of the collaboration contract model we define in chapter 4. We will apply our extended collaboration contracts model to document the source code implementing the hierarchies that play each of those three fundamental roles. Then we analyse if the structure and interactions of the pattern are perceivable in the collaboration contract modelling that we obtain. We consider this as a validation that the model for collaboration contracts we propose is expressive enough for documenting statically typed source code.

The basic HotDraw roles participating in the Model-View-Controller in the Java version are: Drawing, Figure, EventHandler, Handle and DrawingCanvas. A Drawing is a complete picture, and is composed of a number of Figures. The DrawingCanvas acts as the “main program” of HotDraw [Brant98] and specifies the Drawing being edited and a palette of tools to work on it. EventHandlers are responsible for modifying the drawing being edited. EventHandler serves as role for specifying the tools grouped in the tool palette of the DrawingCanvas. There is always an active tool, that represents the mode of the DrawingCanvas. For example, a possible tool playing the role of EventHandler is a SelectionTool. Selecting a Figure causes the Figure to present a set of Handles, that can be manipulated by the tool to change the Figure. Interface Handle is an EventHandler’s child.

When a DrawingCanvas receives an event it delegates it to its active tool. The active tool interacts with a Figure in the Drawing to modify it (or perform the action corresponding to the tool). After the Figure is modified, it notifies the DrawingCanvas of the change to enable to repaint the picture. The DrawingCanvas plays the role of the *view*, the Figure is the *model* and the EventHandler is the *controller*.

As the main structure of the pattern is explained in terms of interfaces, it is necessary to take a group of classes implementing those interfaces in order to see the collaborations that take place. We chose classes SimpleDrawingCanvas, ConnectingLineTool, and LinearShape as implementors of interfaces DrawingCanvas, EventHandler and Figure respectively. A collaboration contract documenting the code representing the explained patterns in HotDraw is presented in chapter 4.

HotDraw and Java

When implementing the Java version of HotDraw some useful features that Smalltalk lacks were used to improve the way to structure the framework.

The problem is that, as powerful as Smalltalk is [LP90], it does not provide an explicit difference between implementation and design. Class hierarchies of frameworks implemented in Smalltalk do not always reveal the design behind them, and often are confusing for users inspecting them. Abstract classes are used both to define the interface of objects (which is a design issue), and as basis for concrete classes of objects (which is an implementation concern). There is no explicit way in Smalltalk to differentiate those roles at design level, nor are there many mechanisms to specify that classes in a hierarchy can play several roles, or that roles can be played by several classes in different hierarchies.

Java [GJS96] lacks some of the flexibility features provided in Smalltalk, but it introduces other powerful features that help in tackling the problem described above. The most powerful in that sense is the concept of interfaces. Interfaces are essentially a group of method signatures which imply certain behaviour. With interfaces Java encourages the separation between design and implementation concerns, because it enables developers to separate design inheritance hierarchies from implementation inheritance hierarchies.

When working with Java, framework developers should concentrate on the identification of the fundamental roles that form the kernel of the framework [Lea95]. This is a significant change with respect to more classical approach that defines classes as the basic units of object-oriented design. Fundamental roles of a framework should be just a few (around seven), while there are normally a higher number of classes. Roles will be implemented by interfaces, and different classes that fulfil the roles can be defined. When using the framework for a particular application, additional classes will fulfil the same roles. Whenever possible classes should collaborate with other objects through interfaces, references to classes should be limited as much as possible (only when constructing new objects, for example). In this way any class implementing the interface could play the role without the need for changes.

In a framework in Java, abstract classes should be implemented to provide as much concrete behaviour as possible for a group of concrete classes. They must define very few abstract methods that will be specified by subclasses, and all the other methods should directly or indirectly depend on them.

In HotDraw for Java, the framework's main structure is implemented completely with interfaces. There is an interface which defines figures. Figure has 32 public protocols. An abstract class implementing Figure is BasicFigure which defines 46 methods, most of which are protected. Only 3 of them are abstract and must be implemented by subclasses, the remaining methods rely more or less directly on these 3.

The extensive use of interfaces that HotDraw makes for the design of the framework makes them an important feature to consider at the moment of extracting structural information with our collaboration contracts documentation.

HotDraw and Reuse Contracts

We tried to use the code corresponding to the classes and interfaces in the Model-View-Controller explained above to apply the collaboration contract documentation defined in [Lucas97]. We found some hard restrictions that made it impossible to abstract the intended interactions in a collaboration contract.

A sample code is shown below that corresponds to the method `mouseDown` in class `ConnectingLineTool`.

```
public boolean mouseDown(Event evt, int x, int y) {
      super.mouseDown(evt,x,y);
      // see if the initial point was on top of a figure
      LineFigure myShape = (LineFigure)figure;
      if (myShape.getNumberOfPoints() != 2)
            return true;
      Figure target = canvas.otherFigureAt(figure,x,y);
      if (target != null) {
            Locator newLocator = target.requestConnection(myShape,x,y);
            if (newLocator != null) {
                  myShape.setLocator(0,newLocator);
                  canvas.moveFigureBehind(myShape,target);
                  canvas.repaint(myShape.getBounds());
            }
      }
```

```
        return true;
    }
```

The first problem we found was the super invocation in the first line of this method. Even when in collaboration contracts it is possible to indirectly represent super sends by embodying the specialisation clause method in the superclass, this has some restrictions. It is not possible to express the fact that a class redefines an operation of a superclass and uses the implementation provided by the superclass through a super call. “Each participant in a collaboration contract should be self-contained, without needing knowledge of the superclass” [Lucas97]. The important restriction here is that it is not possible to represent the fact that a class inherits from another. In HotDraw all classes are very well-factored, and the behaviour of a class is spread along the hierarchy of its ancestors. For example several ancestors of `ConnectingLineTool` implement method `mouseDown`, performing super calls as well. It is not possible to represent this design/implementation in the collaboration contracts.

Another issue concerning inheritance are instance variables. The canvas referenced in the code above is an instance variable inherited from class `CanvasTool`. The corresponding declaration is the following

```
public abstract class CanvasTool extends SimpleEventHandler {
/**
 * The canvas upon which to "operate"
 */
protected DrawingCanvas canvas;
```

There is no way to represent in the collaboration contracts the inheritance of variable `canvas` from `CanvasTool` in class `DrawingCanvas`. Moreover `DrawingCanvas` is an interface, and as in a collaboration contract all participants are of the same kind, interfaces have to be represented as participants in the same way as classes are represented as participants. The problem of this is that, as interfaces have methods but not implementations for those methods, the participant associated to acquaintance canvas will have methods without any specialisation clause. Hence, the behaviour performed in response to the invocation

```
canvas.otherFigureAt(figure,x,y);
```

in the body of `mouseDown` cannot be expressed in the extracted collaboration contract.

Another problem is that the original definition of reuse contracts enforces the Law of Demeter. We see in the above code that this principle is not followed, since methods are invoked on objects referenced by local variables, like `myShape`, `target` or `newLocator`. With the original definition of collaboration contract these invocations cannot be represented in the specialisation clause corresponding to `mouseDown`. The breach of this law is also visible in the following method of class `BasicFigure` (ancestor of `LinearShape`)

```
public boolean contains(int x, int y) {
return bounds().contains(x,y);
}
```

Here a self call of `bounds` is performed, and the result is used to invoke method `contains`. This cascaded method invocation cannot be represented in collaboration contracts either because there is no way to represent the results of methods.

Another feature very used in the code above and that we cannot express in a collaboration contract are type casts (see next section for a description). We observe that throughout the HotDraw framework (and in any Java program in general) type casts are extensively used. In general they are used to cast from a class (interface) to a descendant class (interface), so that the invocation of a method defined in the descendant is valid, or to cast from classes to interfaces they implement, or vice versa.

Another feature that cannot be represented directly in collaboration contracts are the arguments of methods. This is a problem when invocations to actual arguments are performed in the body of a method, because those invocations cannot be reflected in the collaboration contract.

We also found that static attributes like class variables or class methods cannot be distinguished in the notation of collaboration contracts.

After considering the restrictions to the extraction of collaboration contract out of the Model-View-Controller implementation in HotDraw, we selected some of the features that need to be added in order to allow the application of the model to real source code. We decided that gradual extensions to the model are better than trying to integrate a great number of different features all at once. We identified two kinds of extensions to perform first: one concerning the inheritance hierarchies, and another concerning type features.

Inheritance features:

- representation of class inheritance hierarchies: inheritance of methods and instance variables
- representation of super sends and self sends with late binding
- representation of interfaces
- representation of interface inheritance hierarchies: inheritance of methods
- representation of implementation links between classes and interfaces

Type features:

- representation of results of methods (method return types), to allow cascaded invocations
- representation of type casts

There are other features that could be integrated in the model as well (like parameter passing, and class features), but we selected the ones above because they form two cohesive groups of requirements for a not so large, but still meaningful, extension.

Typing features of Java

In this section we will present some technical specifications of how Java provides some of the features analysed in the previous section. We don't intend to make a comprehensive presentation of the language specification or its type system. Our purpose here is just to describe some technical details of some features of the Java semantics, that will serve as specification for our implementation in the reuse contracts model.

Types

Java is strongly typed language, which means every expression has a type that is known at compile time [GJS96]. There are two kinds of types in Java: *primitive types* and *reference types*. There is also a null type (without name) whose only value is `null`.

Primitive types are boolean and numerical types. *Reference types* are class types, reference types and array types. Primitive values are boolean or numerical values. Reference values are pointers to *objects*. *Objects* are instances of class or arrays.

Methods

Method declarations are divided into *method header* and *method body*, with the following structure

MethodDeclaration:

MethodHeader MethodBody

MethodHeader:

MethodModifiers$_{opt}$ *ResultType MethodDeclarator Throws*$_{opt}$

ResultType

Type

`void`

MethodDeclarator:

Identifier (*FormalParameterList*$_{opt}$)

The method header contains the modifiers, the return type of the method, the name, the parameters and the throws clause. The modifiers can be one or more of: `public`, `protected`, `private`, `abstract`, `static`, `final`, `synchronized` and `native`.

The method declaration either specifies the type of the values that the method returns, or the keyword `void` to indicate that no value is returned.

The return type of a method indicate the type of value that will be obtained each time the method is invoked. When the return type of a method is a reference type, cascaded invocations can be expressed. A cascaded invocation is like the following

```
bounds().contains(x,y)
```

In this base the receiver is self, but the keyword `this` can be omitted. The method contains is invoked on the value returned by the invocation of `bounds()` on the receiver.

The *signature* of a method consists of the name of the method and the number and types of its formal parameters.

Class hierarchies

The class hierarchy in Java has a single root, it is class Object. All classes inherit directly or indirectly from Object.

The methods and fields declared by a class are inherited by its subclasses (except if they are `private` or overridden in the subclass).

Java allows to define in a subclass an instance variable (field) with the same name as an inherited one. They must have a different type, and the variable in the subclass is said to *hide* the declaration in its superclass, this mechanism is also called *variable shadowing*.

Instance methods in a class are said to *override* all the inherited methods that have the same interface and the same return type. An overridden method can be accessed from its subclasses using `super` invocations.

Java provides also *overloading* of methods, when a subclass declares a method with the same name, but different signature and different return type than another inherited (or own defined) method.

Interfaces

An interface in Java is a reference type with no implementation. Interfaces are used in Java to allow objects to support common behaviour without sharing any implementation.

An interface is inherently abstract. All the methods it defines are abstract. An interface can have one or more direct superinterfaces. The set of methods inherited by an interface is the union of the methods defined by all its superinterfaces. Hence, the interface hierarchy implements multiple inheritance, and there is no single root such as Object for class inheritance hierarchy. Overriding and overloading are valid for interface methods the same as for classes, they apply also for methods inherited from different superinterfaces.

Interfaces can also declare static variables. Moreover these variables are also `public` and `final`. In contrast to variables, methods in interfaces cannot be declared `static`.

A class may be declared to implement an interface, meaning that instances of the class will implement all the methods specified by the interface. As an interface inherits the methods of its superinterfaces, if a class implements an interface it also indirectly implements all its superinterfaces. The subclasses of a class also indirectly implement all the interfaces that the superclass implements.

A variable whose type is an interface type may hold as value a reference to any instance of a class declared to implement the interface

Type casts

Casting conversions can be applied to any primitive type or reference type to convert values at run-time, or to check that a value at run-time is compatible with a certain reference type.

A cast is for example

```
(LineFigure)figure
```

where the value of the variable `figure` is cast to type `LineFigure`. The type to which the value is converted is always surrounded by parenthesis, and is called the *target* type.

Casts from a primitive type to a reference types, or vice versa, are never allowed.

Casts between two reference types (without considering array types) obey the following rules. A cast from a value that at compile-time has type S (from *source* type) to a target type T

- is valid at compile-time if:
  - T is an interface type and S is an interface type that is direct or indirect subinterface or superinterface of T.
  - T is a class type and S is a class type that is a direct or indirect subclass or superclass of T
- is invalid at compile-time if:
  - T is an interface type, S is a class type declared final, and S does not implement T
  - S is an interface type, T is a class type declared final, and S does not implement S

Observe that a cast from a interface type to a class type declared final (or vice versa) that does not implement the interface is certainly incorrect, since no subclasses could be defined that implement the interface. On the other hand, if the class is not final then no further check can be performed and the cast is considered correct at compile time. Because even if the class does not implement the interface, a subclass might do it.

Chapter 4 - Extended collaboration contracts

We have stressed the importance of reuse contracts in dealing with reuse and evolution. We have also described the problems that arise when trying to apply them to source code. In this chapter we will present the first part of our extension to the Reuse Contracts model in order to tackle the problems described previously.

We propose a formal model using functions and function domains.

We extend the original definition of reuse contracts presented in [Lucas97] by introducing variants to the four basic constructs in collaboration contracts (participants, acquaintances, operations and operation invocations). The reuse contracts operate by modifying, adding or removing these constructs. Consequently there will be variants to the *modification operators* in the original formulation of reuse contracts. The combination of these new modifiers yield to different kinds of conflicts that can be detected by our extension of reuse contracts. Some of these conflicts correspond to variants in the original definition and others are new conflicts that can only be detected due to the new features added in the collaboration contracts descriptions.

Our model is a *conservative extension* to the original definition. All the constructs in the original collaboration contracts are present in our model, all the original modifiers are present and further refined in our model, and most important, all the conflicts detected before can still be detected, plus new extra conflicts.

In this chapter we will introduce our definition of collaboration contracts, and in the two subsequent the new modification operators definitions and detectable conflicts are presented. In this chapter we will first describe informally the added features, then present their graphic notation and finally introduce the formal definition for them. In this way both the graphic notation and the formalism supporting it will be described incrementally as its parts are introduced. We will show only the relevant formal definitions, to give a precise description of the introduced elements. For the complete definition of the formalism the reader is referred to the Appendix A, where the whole formal description as well as the needed notational conventions and mathematical preliminaries are presented.

Collaboration contracts

Collaboration contracts are abstractions of certain portions of code (e.g. packages or parts of packages in Java programs), representing a certain behaviour that is documented for reuse and evolution. A collaboration contract has a name that identifies it, and consists of a set of participants and relations between them. We will distinguish between two kinds of participants, and we define a hierarchical relations for each of them, as well as a third relation that relates participants of the two kinds. All of these components will be further described in the following subsections.

Graphically, we represent collaboration contracts as rectangles containing the graphical description of the structure of the collaboration contract, and a shadowed upper box that holds the name of the collaboration contract. Figure 4.1 shows an (empty) collaboration contract for the Model-View-Controller collaboration in

the HotDraw code presented before. For the moment it is left empty, but we will represent elements inside it as we gradually define those element in the extended collaboration contract model.

Figure 4.1: HotDraw Model-View-Controller collaboration contract

Formally, the set of all the collaboration contracts is defined using a function that maps collaboration contracts names to collaboration contracts structures. This function is called **collContract**, and the domains it maps are **CCNam** and **CCStructure**. Functions are defined using "**:**", and function domains are defined using "**=**". The structure of collaboration contracts, i.e. **CCStructure** is represented by a tuple which contains the four elements mentioned before: participants, and the three kinds of relations between them.

CCStructure = Participants x ClassHierarchy x InterfHierarchy x ImplemRelation
collContract : CCNam → CCStructure

The expression

collContract(HotDrawMVC**)**

represents the collaboration contract for the Model-View-Controller represented graphically in Figure 4.1.

In the following section we will give an initial description of **Participants**. After that we will describe the three relationships connecting participants (**ClassHierarchy**, **InterfHierarchy** and **ImplemRelation**) and later on we will further present the internal structure of the different kinds of participants.

Participants

Participants in our model basically represent abstractions of classes in the source code. They exhibit some behaviour and interactions with other participants in the same collaboration contract. We already saw the importance of interfaces as provided by some programming languages. We will also represent interfaces as participants inside collaboration contracts. This is the first important extension in our model. We distinguish between two kinds of participants: *Class participants* and *Interface participants*. Class participants implement the behaviour and interactions with acquainted participants. Interfaces are simply specifications of protocols, in the sense that they don't describe any behaviour but only specify a set of operations that should be provided by some class participants.

In the following we use the word *participant* to refer to any kind of participant. When a distinction needs to be done we will use either *class participant* (or *class*, when it is clear from the context that we are referring to a participant) or *interface participant* (or simply *interface*).

The same as in the original definition of reuse contract, each participant has a name that univocally identifies it inside a collaboration contract. The set of class names and interface names have to be disjoint. Participants are graphically represented as rectangle boxes inside a collaboration contract box. The participant box holds the name of the participant. To distinguish between class participant boxes and interface participant boxes, we annotate the latter with the label *«Interface»* (similar to the UML notation for interface stereotype [Booch99]). Figure 4.2 shows some participants that belong to the HotDrawMVC collaboration contract.

Figure 4.2: Participants inside HotDrawMVC

Formally, we define the domain of the participants inside a collaboration contract as a name space mapping participant names (**PartNam**) to a **Participant** domain. The **Participant** domain represents the structure of all the participants and is formally defined as the disjoint union of two domains: **Class** and **Interface**.

$$\mathbf{Participants = PartNam \rightarrow Participant}$$
$$\mathbf{Participant = Class + Interface}$$
$$\mathbf{PartNam = ClassNam + InterfNam}$$

These domains are manipulated with functions to obtain the participant corresponding to a certain participant name inside a collaboration contract (function **participant**), to obtain the names of all the participants inside a given collaboration contract (function **participantNames**), the names of all its classes (function **classNames**) and the names of all its interfaces (function **interfaceNames**).

$$\mathbf{participant: CCNam \rightarrow Participants:}$$
$$\mathbf{cc \rightarrow p \rightarrow (parts(cc))\ (p)}$$
$$\mathbf{participantNames: CCNam \rightarrow P(PartNam)\ :}$$
$$\mathbf{cc \rightarrow Dom(parts(cc))}$$
$$\mathbf{classNames: CCNam \rightarrow P(ClassNam):}$$
$$\mathbf{cc \rightarrow participantNames(cc) \cap ClassNam}$$
$$\mathbf{interfaceNames: CCNam \rightarrow P(InterfNam):}$$
$$\mathbf{cc \rightarrow participantNames(cc) \cap InterfNam}$$

The expression **Dom(f)** is used to denote the set of elements for which the partial function f is defined. The function **parts** whose definition is not presented here, given a collaboration contract name returns its participant component.

The function **participant** is used to obtain the participant corresponding to a given participant name in a certain collaboration contract. It uses the function **parts** to get the participants component of the collaboration contract, which belongs to the **Participants** domain. Then it applies the obtained function to the name of the participant and the actual participant is obtained. For example, to get the participant corresponding to the CanvasTool name in the HotDrawMVC collaboration contract graphically introduced in Figure 4.2, one should use the expression

participant(HotDrawMVC**,** CanvasTool**)**

To make our formal expressions simpler to read, we make an abuse of notation and write the application of curried functions as if they were multiple parameter functions. In order to know the names of all the participants inside a given collaboration contract, the function **participantNames** takes the domain of the **Participants** function in the collaboration contract, that is obtained using the **parts** function. Function **participantNames** is used by subsequent definitions that filter the names of the class participants (**classNames**), or the interface participants (**interfaceNames**).

The defined functions can be applied to the HotDrawMVC as defined in Figure 4.2, obtaining the following expressions

participantNames(HotDrawMVC**) = {** CanvasTool, Figure, ConstructionTool, DrawingCanvas, SequenceOfFigures **}**
classNames(HotDrawMVC**) = {** CanvasTool, ConstructionTool **}**
interfaceNames(HotDrawMVC**) = {** Figure, DrawingCanvas, SequenceOfFigures **}**

Thus far we presented the general organisation of participants inside collaboration contracts. We will now describe the internal structure of each kind of participant. Class participants have basically the same structure as the participants in the original reuse contracts definition[1]. In our model the structure of class participants is described in terms of a *signature*, a *specialisation interface* and an *acquaintance clause*. The structure of interface participants is simpler, it consists solely of a *signature*.

This is formally defined by the following expressions

Class = Signature x Spec_Int x Acq_clause
Interface = Signature

We will see how these internal components are defined and used in later sections. First it is necessary to understand how the participants relate between them.

[1] This cannot be seen directly in the formalism because we will present the structure of participants in a slightly different way than it is presented in [Lucas97]. This restructuring enables us to deal in a more uniform way with class and interface structures.

Participant relations

We have already seen in the previous chapter that classes and interfaces at the implementation level are organised in hierarchical structures. This is valid not only for implementation level but also for any level of software design (architectural design, detailed design, etc). The hierarchical organisation of classes and interfaces highly determines their behaviour and interactions. The behaviour of a certain class is often spread over its ancestor classes.

When reusing or evolving object-oriented software, changes to the structure in which classes are organised are expected to occur. When reusing a framework for example, one frequently introduces new subclasses of framework classes in order to customise their functionality. When new versions of components come up they usually reorganise the way in which classes are structured, as is proposed by refactorings [OJ93], [JO93].

In order to provide documentation for changes in the structural organisation of classes, it is necessary first to be able to describe such structures. In the reuse contract model this implies that the collaboration contracts should provide a way to express those structural relations between participants. Once such hierarchies are expressed in a collaboration contract, changes to them can be expressed through modification operators that precisely specify such changes.

In early publications of the reuse contract model [SLMD96] the subclassing mechanism was proposed as steps of reuse. However this is not enough to express evolution of class hierarchies, because the inheritance relation is not explicitly represented in the abstraction of the software artefact to be reused. In [Lucas97] collaboration contracts are defined as containing classes only related through acquaintance relationships.

In this work we introduce inheritance structures as a new kind of relation between participants. We will enrich the collaboration contracts with class hierarchies and interface hierarchies. Moreover, we will provide an implementation relationship that relates class participants to interface participants. This is a substantial extension to the original definition that allows detecting new kind of conflicts in reuse and evolution.

The class hierarchy resembles that of languages like Java or Smalltalk. It is a single inheritance relation, and the way in which class features are shared is similar to that of Java, except for some restrictions that will be pointed out as they are introduced in the text. In our model, there is not a single root class for each collaboration. As collaboration contracts represent abstractions of pieces of code, it may be the case that the parent class of a given class is not represented in the collaboration contract because it is not relevant for the intended modelling. We say that a class has (at most) one *superclass*, which is its direct parent class in the inheritance relation. The set of indirect ancestor of a class is called its *ancestors*. Complementary, the set of direct children of a class is called its *subclasses*, and the indirect children are referred to as its *descendants*. A class ‘inherits’ from its ancestor classes all the behaviour (namely signature and specialisation interface) and structure they define, and can add new own behaviour and structure.

The graphical notation we use for class inheritance is the same as in UML [Booch99]. In the HotDraw example, class participant CanvasTool is the superclass of ConstructionTool. This is represented in Figure 4.3.

Figure 4.3: Class hierarchy relationship

The interface hierarchy in our work models that of Java. It is a multiple inheritance hierarchy, where a subinterface relation between two interfaces has a set inclusion semantics (with respect to the set of operations specified). An interface's protocol specification is the union of that in all its ancestor interfaces plus any new operation specification that it may define by itself. The set of direct parents of an interface in the interface hierarchy relation is called its *superinterfaces*, the set of indirect parents is called its *ancestors*. For the direct children of an interface, we refer to as its *subinterfaces*, and the indirect children are its *descendants*.

The graphical notation for the interface hierarchy relation is the same as for class participants. In the example of Figure 4.2 we could add information stating that DrawingCanvas is a subinterface of SequenceOfFigures, as is shown in Figure 4.4.

Figure 4.4: Interface hierarchy relationship

The third relation between participants is called *Implementation relation*, and it associates participant classes to interface classes. An implementation link between a class and an interface means that the class is an *implementor* of the interface (resp. the interface is a *specification* of the class) i.e. it provides the whole behaviour for the operations specified by the interface.

The implementation relation has the same meaning as the relation determined by the 'implements' keyword of class declarations in Java programs, as described in chapter 3 and defined in [GJS96]. A class participant can be implementor of several interface participants, and the same interface participant can be specification of different class participants. Also, due to the semantics of interface inheritance, the fact that a class implements an interface implies that it is also an implementor of all the interface's ancestors.

If in the collaboration contract of Figure 4.2 we model the class SimpleDrawingCanvas of the Model-View-Controller in HotDraw with a class participant, we could express the fact that it is an implementor of the interface participant DrawingCanvas. This is graphically represented as in Figure 4.5. For the implementation link we use the graphical dependency symbol introduced by UML [Booch99], except that we omit the stereotype *«implement»*, since it is considered as default in this work.

Figure 4.5: Implementation relationship link

Now we will describe how these relationships between participants are represented in our formalism. Recall that the collaboration contracts structure domain is a tuple whose three last components are **ClassHierarchy**, **InterfHierarchy** and **ImplemRelation**. The definition of these domains is

ClassHierarchy = ClassNam → ClassNam
InterfHierarchy = InterfNam → P(InterfNam)
ImplemRelation = ClassNam → P(InterfNam)

ClassHierarchy is a mapping from class names to class names; elements of this domain will be mappings that, given a class, return its unique superclass in the collaboration contract where they are defined. Elements in the **InterfaceHierarchy** domain are mappings that retrieve the set of names of the superinterfaces of a given interface name. The **ImplemRel** domain maps the class names domain into the powerset of interface names. For each element of this domain, given a class name it will retrieve the set of names of all the interfaces the class implements.

To handle the participant relations in a useful way, we define the following functions.

superclass : CCNam → ClassNam → ClassNam
cc → c → class_hier(cc, c)
superInts : CCNam → InterfNam → P(InterfNam)
cc → i → interf_hier(cc, i) , if i ∈ Dom(interf_hier(cc))
{ } , otherwise
specifications : CCNam → ClassNam → P(InterfNam)
cc → c → implem_rel(cc, c) , if c ∈ Dom(implem_rel(cc))
{ } , otherwise

The first two functions are used to obtain the direct parent participants in the class and interface hierarchies respectively. **superclass** uses a function **class_hier** that given a collaboration contract retrieves its class hierarchy component. The third function retrieves the set of all the interfaces that are specifications of a given class in the collaboration contract.

The counterparts of the functions above, that retrieve the direct children in the hierarchies and the implementors of a given class are

subclasses: CCNam → ClassNam → P(ClassNam):
cc → c → {c1 | superclass(cc, c1)=c}
subInts: CCNam → InterfNam → P(InterfNam):
cc → i → {i1 | i ∈ superInts(cc, i1)}
implementors: CCNam → InterfNam → P(ClassNam):
cc → i → {c | i ∈ specifications(cc, i)}

We also define functions to obtain the ancestors an descendants of a participant in its corresponding hierarchy, that are computed as the transitive closure of the functions defined above.

descendants: CCNam → PartNam → P(PartNam)
ancestors: CCNam → PartNam → P(PartNam)

For a complete definition of these and other functions refer to the Appendix A.

All the elements in the participant relation domains should satisfy some constraints, such as the single inheritance of class participants (the relation must represent a tree) and the absence of cycles in both hierarchies. We formally represent these constraints in well-formedness conditions that are defined for each relation. A well-formedness condition for a participant relation states when an element in the corresponding domain (**ClassHierarchy**, **InterfHierarchy** or **ImplemRelation**) is a valid hierarchy or implementation relation inside the collaboration contract where it is defined.

These well-formedness conditions are expressed as boolean functions in our formalism.

wf_classHier : CCNam → Bool
wf_interfHier : CCNam → Bool
wf_implemRel : CCNam → Bool

All these well-formedness functions receive a collaboration contract name and checks that the corresponding relation in the corresponding collaboration contract structure is well-formed. The first thing they check is that the referenced class participant names and interface participant names correspond to participant inside the collaboration contract. **wf_classHier** and **wf_interfHier** check that there are no cycles in the hierarchies, and that some semantic restrictions concerning the structure of the related participants are respected. **wf_implemRelation** checks that each class in the relation has defined the required behaviour to implement the operations specified by its interface. The well-formedness functions are used to define the applicability of a modification operator on a collaboration contract. For the complete definition of the functions the reader is referred to the Appendix A.

Signatures

In order to specify the behaviour of the collaboration contract, participants provide a number of operations. Operations represent methods in the implementation level. In our model of collaboration contracts we do not distinguish between operations and methods and henceforth use any of the two words as synonyms.

At this point we introduce the notion of *type*. A *type* in our model is the specification of a participant inside a collaboration contract. A type is a participant name, it can be either a class name or an interface name.

Types in our model are used, amongst others, to specify operation *return types*. In object-oriented programming languages, an object is always returned as the result of the invocation to a method[2]. In our model the *return type* of an operation is characterises the returned object by specifying its type, in the same way as Java defines method return types.

Operations in class participants have a name, a return type and a specialisation clause. Operations in interface participants have a name and a return type. The *signature* of a participant is composed of the names of all the operations it defines, together with their corresponding return type. This is different from the definition of signature in the Java language where, as explained in chapter 3, references the method names and the number and types of the method arguments.

When organised in hierarchies, participants share operations defined by their ancestors. This means that the methods the participant understands are the *own-defined* ones (those in its signature) plus the *inherited* ones (those understood by its ancestors). If a participant has no ancestors, then the methods it understand are only the *own-defined* ones.

The same as in programming languages, classes and interface participants can redefine operations defined by its ancestors. They do so by providing an operation with the same name and return type as one defined by an

[2] In the last subsection of this chapter we show how we deal with methods that don't have a return type (specified in Java with the `void` keyword).

ancestor. This means that inside a participant's own-defined methods we distinguish between new methods (those that no ancestor defines already) and overriding methods.

In contrast to the overloading mechanism of Java, we don't allow overloading of method return types. That is, a participant cannot define a method with the same name as a method in an ancestor, if it has a different return type. Either it must have the same return type as the method of its ancestor, and then it is a overriding method, or it must have a different method name, and then it is a new method definition.

For a class participant, the set of inherited methods are the methods understood by its unique superclass. For an interface participant, the set of inherited methods is the union of the methods understood by all its superinterfaces. A restriction is imposed in our model, that states that if a method is defined in more than one superinterface, then it should have the same return type in all the definitions, otherwise the interface hierarchy is not well-formed (this is part of the definition of the **wf_interfHier** presented above).

In our graphic notation we denote the return type of an operation preceding the name of the operation, as illustrated by Figure 4.6. The operations defined by a participant are included in a lower box inside the participant square. Only the own-defined methods are located inside a participant box. The inherited methods will be included inside the participant that defines it.

Figure 4.6: Method return type notation

In Figure 4.6, a detail of interface participants SequenceOfFigures and DrawingCanvas in the HotDrawMVC collaboration contract is shown. SequenceOfFigures has an operation otherFigureAt, with return type Figure, which is another participant in the same collaboration contract. In the same way DrawingCanvas defines the method repaint. The inheritance link between these two classes implies that method otherFigureAt is also understood by DrawingCanvas.

Note that as return type of the repaint method there is the word 'Void'. This is our modelisation for the specification of 'Void' for method return types in Java. We consider Void as a trivial participant with an empty signature, no ancestors and no descendants. This allows us to keep a uniform treatment of method return types as provided by programming languages like Java.

Formally we define the **Signature** domain as a mapping from method names to method return types.

Signature = MethNam → Type
Type = PartNam

The domain for the method return types is the same as the one for participant names.

We define the following functions to deal with the signature of participants

part_sign: CCNam → PartNam → Signature :
cc → p → sign(participant(cc, p))
part_ownMethNames: CCNam → PartNam → P(MethNam):
cc → p → Dom(part_sign(cc, p))
part_ownMethRetType: CCNam → PartNam → MethNam → Type :
cc → p → m → part_sign(cc, p, m)

In these function definitions the auxiliary function **sign** is used, that given a participant structure returns its signature component. The function **part_sign** is defined in terms of it and returns the signature corresponding to a given participant name inside a given collaboration contract name. It should be here again the abuse of notation that we mentioned before. As function **sign** returns a signature (that is a function itself), this result can be further applied to a method name, as in the definition of **part_ownMethRetType**.

We will refer to the *context* of a feature in a collaboration contract, as all the information necessary to univocally identify it, without considering the name of the feature. In the definitions above, the context of a participant signature is the name of the collaboration contract and the name of the participant where it is defined (**cc** and **p**). We will call this a *participant attribute context*, because it is the context for any attribute defined inside a participant. Note that it is necessary to provide the name of the participant and the name of a collaboration contract where the participant is defined because the same participant name can appear inside different collaboration contracts.

The above functions are used to obtain the signature of a participant (function **part_sign**), or to get all the names of method it defines (function **part_ownMethNames**), or the return type of one of its own methods (function **part_ownMethRetType**). These three functions work only over the set of own-defined methods. There are also functions that deal with the whole set of methods understood by a participant:

part_methRetType: CCNam → PartNam → MethNam → Type :
cc → p → m →
part_ ownMethRetType(cc, p, m) , if m ∈ part_ownMethNames(cc, p)
part_ownMethRetType(cc, sp, m), , if m ∉ part_ownMethNames(cc, p)
where sp ∈ ancestors(cc, p) ∧ m ∈ part_ownMethNames(cc, sp)
part_methNames: CCNam → PartNam → P(MethNam):
cc → p → map(part_ownMethNames, ancestors(cc, p))

Given a participant attribute context and a method name, function **part_methRetType** retrieves the return type of a method in a participant, no matter if the method is own defined or inherited. It does so by applying function **part_ownMethRetType** on the received participant, if the method is own defined, or in an ancestor participant defining the method, if it is inherited. It should be noted that because of the restriction on non-overloading return types that we mentioned above, it does not matter from which ancestor the return method type will be obtained, since in a well-formed hierarchy all the ancestors defining that method will have the same return type for it.

Function **part_methNames** receives a participant attribute context and returns the set of names of all the methods understood by the participant. It proceeds collecting the method names in all the ancestors of the received participant.

With these two sets of functions it is possible to obtain, through set subtraction, the information about inherited methods of a participant.

The information about the signature of the interfaces illustrated in Figure 4.6 can be expressed in the formalism with the expressions

part_sign(HotDrawMVC**,** SequenceOfFigures**) = { (**otherFigureAt**,** Figure**) }**
part_sign(HotDrawMVC**,** DrawingCanvas**) = { (**repaint**,** Void**) }**
part_ownMethNames(HotDrawMVC**,** SequenceOfFigures**) = {** otherFigureAt **}**
part_ownMethNames(HotDrawMVC**,** DrawingCanvas**) = {** repaint **}**
part_methNames(HotDrawMVC**,** SequenceOfFigures**) = {** otherFigureAt **}**
part_methNames(HotDrawMVC**,** DrawingCanvas**) = {** otherFigureAt, repaint **}**
part_methRetType(HotDrawMVC**,** SequenceOfFigures**,** otherFigureAt**) =** Figure
part_methRetType(HotDrawMVC**,** DrawingCanvas**,** otherFigureAt**) =** Figure
part_methRetType(HotDrawMVC**,** DrawingCanvas**,** repaint**) =** Void

We will defer the description of *specialisation interfaces*, and introduce first the *acquaintance clause* definition, which is needed for better understanding the structure and functioning of the specialisation interfaces.

Acquaintance clauses

Participants are acquainted with each other in different ways. In [Lucas97] an abstraction is made of what an acquaintance relationship corresponds to in actual code. In that model an acquaintance relationship can represent a diversity of different things. In this work we intend to stay closer to source code, so acquaintance relationships will not represent such a variety of ways to be acquainted, but will be restricted to some of them.

In our model the only participants that have acquaintances relationships are classes, not interfaces. We will represent as acquaintance relationships of a class only its direct acquainted participants that are identified by a name inside the participant, e.g. instance variables. We will not represent as acquaintance relationships the indirect acquainted participants that may result as the transitive closure of a series of acquaintance relationships. Method return types will also not be represented as acquaintance relationships, for we have seen that they are explicitly managed in the participants signature. In principle it suffices to assume that the acquaintance relationships of a class participant represent the instance variables of its corresponding class in the source code[3]. The name of the acquaintance relationships represents the name of the instance variable, and the participant associated represents the type of the instance variable. We will also refer to the associated participant as the type of the acquaintance relationship. It should be noted that acquainted participants of a class can be either other classes or interfaces.

The acquaintance clause of a participant is composed of all the acquaintance relationships it defines.

The same as operations, acquaintances are shared by the descendants of a class in the same way as instance variable declarations of a class are inherited by subclasses. The descendant class participants will share the definition of the acquaintance relationship, but of course this models the fact that in the represented source code the corresponding descendant classes has their own value for the inherited instance variable. As in our model we do not deal with values, this fact is abstracted and not explicitly shown.

The inheritance of acquaintances between classes is reflected in the fact that acquaintances defined in an ancestor of a class can be referenced inside the class itself. We will see further how this is done in the definition of specialisation interfaces.

[3] In the last subsection of this chapter we will see what other direct acquainted participants are also represented by the acquaintance relationships in our model. Note also that as interface participants cannot have acquaintances, we don't model here the constant fields that can be defined in interfaces in Java.

In a similar way as with methods, we refer to *own-defined acquaintances* as the acquaintances defined in the acquaintance clause of the class, and to *inherited acquaintances* as those defined by an ancestor and inherited by the class.

I contrast to the shadowing mechanism of Java, we don't allow a class to define an acquaintance with the same name as an inherited acquaintance.

The graphical notation for an acquaintance relationship is the same as in the original definition of reuse contracts and is illustrated in Figure 4.7, where class CanvasTool defines an acquaintance canvas of type DrawingCanvas.

Figure 4.7: acquaintance relationship

In general we do not explicitly represent in the graphic notation the links corresponding to inherited acquaintances (unless an operation invocation to it is performed, which will be seen in next section).

In our formalism, we define the acquaintance clause domain **Acq_clause** and some useful functions that handle this and other domains. **Acq_clause** is the set of all the mappings from acquaintance names to their respective types i.e. participant names.

Acq_clause = AcqNam → Type

class_acq: CCNam → ClassNam → Acq_clause:
cc → c → acq(participant(cc, c))
class_ownAcqNames: CCNam → ClassNam → P(AcqNam):
cc → c → Dom(acq(participant(cc, c)))
class_ownAcqType : CCNam → ClassNam → AcqNam → Type:
cc → c → ac → acq(participant(cc, c), ac)

We define functions to obtain the acquaintance clause of a given class participant in a given collaboration contract (function **class_acq**), to obtain all its own-defined acquaintances names (function **class_ownAcqNames**), and the type of a given own-defined acquaintance relationship (function **class_ownAcqType**). All of these functions take a class attribute context and use an auxiliary **acq** function (which works in a similar way as auxiliary function **sign** used in the previous section) that given a class structure retrieve its acquaintance clause component.

We also define functions to get the names and type of all the acquaintances of a class, the own-defined plus the inherited ones.

class_acqNames: CCNam → ClassNam → P(AcqNam):
class_acqType : CCNam → ClassNam → AcqNam → Type:

These functions are implemented in a similar way as the functions to get names and type of all the methods understood by a participant, that was presented in previous section. The complete definition is shown in the Appendix A.

The acquaintance relationship inside HotDrawMVC as presented in Figure 4.7, is formally described by the following expressions

class_acq(HotDrawMVC**,** CanvasTool**) = { (**canvas**,** DrawingCanvas**) }**
class_ownAcqNames(HotDrawMVC**,** CanvasTool**) = {** canvas **}**
class_acqType(HotDrawMVC**,** CanvasTool, canvas**) = {** DrawingCanvas **}**

The last component of a class participant structure that still needs to be presented is the specialisation interface of class participants.

Specialisation interfaces

The *specialisation interface* of a class participant specifies the dependencies of each of its operations with other operations in the same class or in other participants. It represents an abstraction of the operation invocations performed in the body of the corresponding method in the corresponding class in the source code.

Specialisation interfaces are the feature in our model that abstracts method bodies at source code level. Each class participant defines a specialisation interface which is formed of a *specialisation clause* for each of the methods in its signature. Interface participants do not have specialisation interface, since methods in interfaces at the source code level do not have an implementation. Each *specialisation clause* consists of a set of *operation invocations*. One can decide to include in the specialisation clause of a method all the operations invocation present in the corresponding method body implementation, or to select just the subset of them that is relevant for the modelling. Examples of specialisation clause extraction from source code is shown in the last subsection of this chapter.

An *operation invocation* (also named *method invocation* or *message send*) in our model can have the same form as in the original reuse contracts definition, or it can have other new forms. And this is another essential extension of our model compared to the original one. The operation invocations are different from those in [Lucas97] in three main points.

The first point of difference is the kind of receiver that invocations can have. An operation invocation can be

- a *regular method invocation*, where the receiver of the method is an acquaintance of the class
- a *self method invocation*, where the receiver is the class itself
- a *super method invocation*, where the receiver is the class itself but the method implementation is inherited from an ancestor class

Regular operation invocation is already provided by reuse contracts as defined in [Lucas97]. Self operation invocations are included also, but the late binding of self is not represented in the original definition of reuse contracts. Super operation invocations are not represented in the collaboration contracts model of [Lucas97], although a modification operator to include some form of super invocation is later included. In our extended collaboration contracts, super operation invocations model super method calls as provided by programming languages. It is possible to represent such kinds of operation invocations in this model because we provide a way to represent the class hierarchy. The class hierarchy of the class containing a super invocation determines which ancestor's implementation[4] of the invoked operation will be used in the super call. The method lookup we model is the same as in programming languages, that is, the implementation bound to a

[4] When we refer generically to 'implementation' of a method in our model we are speaking of the abstraction we make of the method implementation at the source code level that is, the specialisation clause. Anyway, sometimes we choose to use the word 'implementation' to ease the understanding of the explanations by associating specialisation clauses it to method implementations at source code level.

super invocation is the one provided by the closest class in the ancestor chain that defines the operation. The implementation of the bound operation is considered in the context of the original receiver, that is the class containing the super invocation. So for example self operation invocations contained in the implementation will bind to the original receiver.

We keep the graphical notation of regular and self operation invocations that were defined in [Lucas97]. Super operation invocations are modelled with an arrow from the caller class to its superclass, labelled with the name of the operation containing the invocation and the keyword super between curly braces.

In the HotDraw Model-View-Controller example, the operation mouseDown in class ConnectingLineTool performs a super invocation (of the same operation). This is depicted in Figure 4.8.

Figure 4.8

The second point of difference of invocations compared to the original definition of reuse contracts is that we provide a more general form of operation invocations. We don't restrict operations to be invoked only on an acquaintance, self or super. Due to the representation of the results of method invocations, we can use those results to further perform operation invocations on them. This results in the representation of *cascaded method invocations* as found in actual source code and explained in the previous chapter.

An operation invocation inside the specialisation clause of an operation *op* inside a class participant *C* can have for example the following structure

acq.op2.op3

provided that

- *acq* is an acquaintance name of class *C* (own-defined or inherited) , of a type *C2*
- *op2* is an operation in the signature of participant *C2*, with a return type *C3*
- *op3* is an operation in the signature of participant *C3*

Cascaded method invocations can also have self or super as the initial receiver. In this case it is required that the first method invoked is understood by the class containing the invocation.

We propose a special graphical notation for cascaded method invocations, that is not intended to be a final one, but just an approach to express this kind of invocations graphically. For each cascaded operation invocation inside the specialisation clause corresponding to a method in a class we annotate it with a greek letter between curly braces next to the name of the method in the box inside the class square. Each greek letter uniquely identifies a cascaded invocation in the entire collaboration contract. Each method referenced in the cascaded invocation can be referenced by the greek letter of the cascaded invocation followed by the order that it has in that invocation (and separated with a dot).

For example the cascaded operation invocation

canvas.otherFigureAt.requestConnection

in the specialisation interface of method mouseDown in class ConnectingLineTool is labelled with letter α. So inside the method box of that class, method mouseDown will be noted with

Bool mouseDown {α}

If there were more cascaded invocations in the same method's specialisation interface, they would be listed inside the curly braces also. Invocation of otherFigureAt is referenced with α.1, and invocation of requestConnection with α.2.

We will draw arrows labelled with the reference of each single operation invocation and between curly braces the method being invoked. The invocation of otherFigureAt will be drawn as an arrow from ConnectingLineTool (the original sender) to DrawingCanvas (the type of the canvas acquaintance) labelled with 'α.1 {otherFigureAt}'. Next, invocation of requestConnection will be drawn as an arrow from DrawingCanvas to Figure (the return type of the previous method) labelled with 'α.2 {requestConnection}'. If there were more invocations cascaded (α.3, α.4, …), the procedure to draw it would remain the same.

Figure 4.9 shows the complete graphical representation of the described cascaded invocation.

Figure 4.9

I should be noted that in the *canvas* acquaintance of class ConnectingLineTool, the notation we use for inherited instance variables is a link drawn from the class to the acquaintance type, where the label is preceded by a '/' indicating that it is inherited.

Also for the second operation of the cascaded invocation, α.2, a dotted arrow is used. This resembles UML notation [Booch99] for dependencies.

The third point of difference between the operation invocations in this model and those in the original reuse contracts, is that we allow *type casts* to the receiver of a method. It is important to be able to express type casts in our formalism because when dealing with typed languages, casting is a mechanism used to a great

extent. Also, if we don't represent type casts in our model, then the abstraction of source code level operation invocations that involve a down-cast would not have meaning. This is because normally after a down-cast an operation is invoked that is not understood by the source type, and is defined by the descendant target type.

Besides it is necessary to represent them, in order to be able to reason on method invocations when they involve casts (specially down-casts, since the method invoked after that kind of casting is frequently not provided by the source type).

We model type casts as they are provided by Java and were explained in the previous chapter. The receiver of a method (be it a simple receiver or the result of one or more method invocation) can be cast to another type. The result of the cast is a participant, that can be used to further invoke messages on it.

The rules for type casts are the same as in Java. Inside a casting we will also call *source type* the type of the expression that is cast, and *target type* the type to which the expression is converted. A participant interface can be cast to any class participant and vice versa, there are no restrictions for these cases. If both the source and target types are of the same kind (both are classes or both are interfaces), then the target type should be an ancestor of the source type (an 'up-cast'), or it should be a descendant of the source type (a 'down-cast').

The graphical notation we use for type casts is similar to the one used for cascaded invocation. The type cast is numbered according to the order it has inside the invocation (which could also be cascaded). The reference to the cast is a label that goes together with the reference of operation invocation performed on the target type, in the corresponding arrow. The label also specifies between square brackets the source and target types separated by an arrow.

For example, if the invocation

```
((LineFigure) figure).setLocator
```

were represented in the HotDraw Model-View-Controller (it is an invocation inside mouseDown in ConnectingLineTool), the graphic in Figure 4.9 would be extended as is depicted in Figure 4.10.

Figure 4.10

In Figure 4.10 we see how the type casting of acquaintance *figure* to class LineFigure has the reference β.1 [Figure→LineFigure] and accompanies the subsequent method invocation (to otherFigureAt) which has reference β.2.

In the formalism, we represent specialisation interfaces with the domains **Spec_Int** and **SC**

Spec_Int = MethNam → SC
SC = P(OpInvoc)

Spec_Int associates a specialisation clause to each of the methods defined by a class. A specialisation clause is a set of operation invocations. Operation invocations can be of any of two kinds: casting operation invocations or non-casting operation invocations. This is expressed with the disjoint union (**+**)

OpInvoc = NCastInvoc + CastInvoc
CastInvoc = Receiver x Type
NCastInvoc = Receiver x MethNam

receiver: **NCastInvoc → Receiver**
method: **NCastInvoc → MethNam**
invoc: **CastInvoc → Receiver**
targetCastType: **CastInvoc → Type**

A non-casting operation invocation is formed of a receiver and a method that is invoked on that receiver. A casting operation invocation is composed of an invocation (the first component of the pair, of type **Receiver**) together with a target type to which the invocation is cast (the second **Type** component of the pair). The auxiliary functions whose type is given above retrieve each of the described components in casting and non-casting operation invocations.

A **Receiver** can itself be an operation invocation or it can be a simple receiver. Domain **ReceiverNam** represents all the possible simple receivers, that can be the name of an acquaintance, or 'self' or 'super'.

Receiver = OpInvoc + ReceiverNam
ReceiverNam = AcqNam + 'self' + 'super'

The specialisation interface of a class participant and the specialisation clause corresponding to a method in a class are retrieved by the following functions

class_specInt: CCNam → ClassNam → Spec_Int :
cc → c → specInt (participant(cc, c))
class_methSC: CCNam → ClassNam → MethNam → SC :
cc → c → m → class_specInt(cc, c, m)

which use the auxiliary function **specInt** that gives the specialisation interface component of a class structure.

To express in the formalism the operation invocations presented in Figure 4.8, we can use the following expressions

class_methSC(HotDrawMVC**,** ConnectingLineTool**,** mouseDown**) = { (**'super'**,** mouseDown**) }**
class_methSC(HotDrawMVC**,** ShapeTool**,** mouseDown**) = { }**

Note that, as we assume a correspondence between methods and the specialisation clauses in class participants, but we define them in two separate components (**Signature** and **Spec_Int**) inside the class structure, we must ensure that the correspondence effectively holds. This check is done in a well-formedness function **wf_class**, that checks if a class inside a given collaboration contract is well-formed.

We also define a well-formedness function for operation invocations inside the specialisation clause of a method of a participant in a collaboration contract

wf_opInv: CCNam → ClassNam → OpInvoc → Bool:
cc → c → op →
(wf_receiver(cc, c, receiver(op))
∧ understands(cc, type(cc, c, receiver(op)), method(op))) **, if op ∈ NCastInvoc**
(wf_receiver(cc, c, invoc(op))

$\wedge$ wf_casting(cc, type(cc, c, invoc(op)), targetCastType(op)) , if op $\in$ CastInvoc
false , otherwise

If the operation invocation is non-casting one (first branch of the definition), the function checks first that the receiver component is well-formed and then that the participant corresponding to the type of the receiver understands the method being invoked. If the operation is a non-casting one (second branch of the definition), it checks first that its invocation component is well-formed and then that the specified casting is well-formed. Function **wf_casting** receives a collaboration contract name, a source type and a target type and performs the check by implementing the rules for type casting described informally above.

Function **wf_receiver** proceeds according to the structure of the **Receiver** recursively calling **wf_opInv** if it is not a receiver name (i.e. it is a complex receiver, containing type casting or cascaded invocations), and checking that the receiver corresponds to an acquaintance name if it is a simple receiver name in **AcqNam** domain.

wf_receiver: CCNam $\rightarrow$ ClassNam $\rightarrow$ Receiver $\rightarrow$ Bool:
cc $\rightarrow$ c $\rightarrow$ rec $\rightarrow$
true , if rec $\in$ {'self'} $\subseteq$ ReceiverNam
has_ancestors(cc, c)) , if rec $\in$ {'super'} $\subseteq$ ReceiverNam
rec $\in$ class_acqNames(cc, c) , if rec $\in$ AcqNam $\subseteq$ ReceiverNam
wf_opInv(cc, c, rec) , if rec $\in$ OpInvoc
false , otherwise

The other two more relevant functions used are **type,** that returns the type of a receiver expression, and **understands**, which returns true if a participant understands a given method. Function **type** proceeds according to the structure of the receiver if it is a receiver name: it retrieves the type of the acquaintance (if it is an acquaintance name) the name of the class received as context (if it is 'self'), or the name of the superclass of the class received as context (if it is 'super'). If the receiver is a casting expression then the type of the whole expression is clearly the target type of the casting, and if it is an non-casting invocation it retrieves the return type of the method in the participant corresponding to the type of the receiver component of the non-casting invocation.

type : CCNam $\rightarrow$ ClassNam $\rightarrow$ Receiver $\rightarrow$ Type :
(cc $\rightarrow$ c) $\rightarrow$ rec $\rightarrow$
c , if rec $\in$ {'self'} $\subseteq$ ReceiverNam
superclass(cc, c) , if rec $\in$ {'super'} $\subseteq$ ReceiverNam
class_acqType(cc, c rec) , if rec $\in$ AcqNam $\subseteq$ ReceiverNam
targetCastType(rec) , if rec $\in$ CastInvoc $\subseteq$ OpInvoc
part_methRetType(cc, type(cc, c, receiver(rec), method(rec))
, if rec $\in$ NCastInvoc $\subseteq$ OpInvoc

understands : CCNam $\rightarrow$ PartNam $\rightarrow$ MethNam $\rightarrow$ Bool :
cc $\rightarrow$ p $\rightarrow$ m $\rightarrow$ (m $\in$ part_methNames(cc, p))

Function **understands** simply checks if the received method name is in the set of names of methods understood by the participant. It uses the function part_methNames introduced in previous sections.

The well-formedness functions presented in this and previous sections are used by a general well-formedness function that checks if a collaboration contract is well-formed. This function proceeds

recursively through the structure of the collaboration contract checking if all the components are well-formed and consistent among themselves.

The definition of this function is

$$\mathbf{wf_collContract : CCNam \rightarrow Bool}$$
$$\mathbf{cc \rightarrow wf_participant(cc) \wedge wf_classHier(cc) \wedge wf_interfHier(cc) \wedge wf_implemRel(cc)}$$

The complete definition of these and other functions is in the Appendix A.

Validation

In this section we will present the resulting collaboration contract for the source code corresponding to the Model-View-Controller implementation in HotDraw for Java. We will show the graphical notation for the collaboration between the participants that play the main interactions in the pattern. These interactions allow to see the pattern structure and behaviour. We consider this as an evidence of added expressiveness of our extended collaboration contracts.

We take the formulation of the Model-View-Controller pattern as presented in [BMRSS97]. The interaction diagram used there to represent the collaboration between the roles in the pattern is shown in Figure 4.11. Later on we will see how the interaction schema shown in that figure is carried out by a group of classes in HotDraw.

Figure 4.11: Model-View-Controller interaction diagram

In HotDraw, one of the defined tools is a ConnectingLineTool, which draws a line connecting two existing figures in the drawing. We will analyse the collaboration that takes place between the mentioned classes when the ConnectingLineTool knows that the mouse was clicked (the mouse was pressed but still not released). The tool fixes the starting point of a line. The starting point for the line will be a connecting point with any figure that is drawn at the point where the mouse was clicked. To do that, the tool asks a connection to the figure located in that place. The figure returns a Locator object, which is a point to know

where to draw figures. Once the locator object is obtained, the line is drawn and the DrawingCanvas is notified of the change for the repainting of the drawing.
The most relevant code source parts corresponding to the described interaction are shown below. We include parts of the HotDraw code here to show how the source code classes and interfaces are represented in our collaboration contracts notation. For the controller role, the interface EventHandler, the class ConnectingLineTool, and relevant superclasses are:

```
public interface EventHandler {
public abstract boolean handleEvent(Event evt);
…}
public abstract class SimpleEventHandler implements EventHandler {
…}
public abstract class CanvasTool extends SimpleEventHandler {
  protected DrawingCanvas canvas;
…}
public abstract class ConstructionTool extends CanvasTool {
  protected Figure figure;
…}
public abstract class ShapeTool extends ConstructionTool {
…}
public class LineTool extends ShapeTool {
…}
public class ConnectingLineTool extends LineTool {
  public boolean mouseDown(Event evt, int x, int y) {
      super.mouseDown(evt,x,y);
      // see if the initial point was on top of a figure
      LineFigure myShape = (LineFigure)figure;
      if (myShape.getNumberOfPoints() != 2)
            return true;
      Figure target = canvas.otherFigureAt(figure,x,y);
      if (target != null) {
            Locator newLocator = target.requestConnection(myShape,x,y);
            if (newLocator != null) {
                  myShape.setLocator(0,newLocator);
                  canvas.moveFigureBehind(myShape,target);
                  canvas.repaint(myShape.getBounds());
            }
      }
      return true;
  }
…}
```

For the model role, interfaces Figure and LineFigure, and class Linear shape and relevant superclasses are:

```
public interface Figure extends Paintable, ObservableWithTransients,
Duplicatable, Serializable {
  public abstract Locator requestConnection(Figure requestor, int x, int
  y);
…}
public interface LineFigure extends Figure {
    public abstract int getNumberOfPoints();
    public abstract void setLocator(int index, Locator locator);
…}
```

```
public abstract class BasicObservable implements Observable, Serializable
{
   protected Object observerList;
   public synchronized void notifyObservers(Object arg)  {
       if (observerList != null) {
             if (observerList instanceof Vector)
                   for (Enumeration e = new
   ReverseVectorEnumerator((Vector)observerList); e.hasMoreElements();)
                         notifyObserver(e.nextElement(), arg);
             else
                 notifyObserver(observerList, arg);
   }
   }
   protected void notifyObserver(Object observer, Object arg)  {
       ((Observer)observer).update(this, arg);
   }
…}
public abstract class BasicFigure extends BasicObservableWithTrans
implements Figure {
   protected Object observerList;
   protected void notifyObserver(Object observer, Object arg)  {
       ((FigureObserver)observer).update(this, arg);
   }
   public Locator requestConnection(Figure requestor, int x, int y) {
   if (requestor == this)
       return null;
   return new FigureRelativePoint(this,0.5,0.5);
   }
…}
public abstract class Shape extends BasicFigure {
   protected void changedShape(Rectangle rect)  {
       notifyObservers(rect);
   }
…}
public abstract class LinearShape extends Shape implements LineFigure {
   public abstract int getNumberOfPoints();
   public void setLocator(int index, Locator locator)  {
             Rectangle oldBounds = getBounds();
             resetBoundsCache();
             basicSetLocator(index,locator);
             changedShape(oldBounds);
       }
…}
```

Finally, the relevant parts of the implementation of the view role by class SimpleDrawingCanvas

```
public class SimpleDrawingCanvas extends java.awt.Canvas implements
DrawingCanvas, FigureObserver, ComponentListener, MouseListener,
MouseMotionListener, KeyListener {
protected Drawing drawing = defaultDrawing();
protected EventHandler tool = defaultTool();
…
   public void update(Figure figure, Object arg)  {
```

```
            Rectangle area = figure.getBounds();
            …
            repaint(area);
        }
    …}
```

We represented the interactions performed when a mouse is pressed and a ConnectingLineTool is active, in the collaboration contract whose description is presented in Figure 4.12. In this graphical notation we omitted some invocations links, to make the drawing more clear. Some of those omitted invocations are expressed in the specialisation clause between curly braces next to the corresponding method. The cascaded invocations are also included in the specialisation clause, with their corresponding reference (α, β, etc.) next to them.

In Figure 4.12 we see what happens when a ConnectingLineTool receives a mouse click event. We highlighted in bold the main method invocations, corresponding to the interactions of the pattern as depicted in Figure 4.11. handleEvent method inherited from SimpleEventHandler calls method mouseDown, that is defined by ConnectingLineTool. Super calls are performed, and the figure located at the place where the mouse was pressed is requested a connection. The figure is obtained by asking the canvas associated to the tool through the canvas instance variable (which has type Figure, and in the case of the connecting line tool). Once the locator corresponding to the requested connection is obtained, the figure of the tool (variable figure, inherited from class ConstructionTool) is set a locator, as a starting point to draw it. This last invocation corresponds to the *service* invocation performed to the *Model* in Figure 4.11.

The setLocator method in class LinearShape performs, among others, an invocation of changedShape on itself. This corresponds to the *notify* invocation of *Model* in Figure 4.11. In response to that, method changedShape (inherited from Shape) performs a self invocation of notifyObservers, to inform all the dependant objects that a change took place.

Class LinearShape inherits from BasicObservable an instance variable observerList of type Object, to hold references to all the observers that depend on it. Method notifyObserver in LinearShape (inherited from class BasicObservable) performs an invocation of update on its observer object previously cast to type Observer. This corresponds to the *update* invocation to *View* in Figure 4.11.

When the SimpleDrawingCanvas (referenced by variable observerList in LinearShape) is invoked method update, it performs a repaint. This corresponds to the *display* invocation in *View* in Figure 4.11.

Figure 4.12

In Figure 4.12 we shown only the mainstream of the collaboration. The complete collaboration contract in a usual syntax is shown in Appendix C.

Collaboration Contracts extraction

We will describe here some considerations that should be made when extracting our extended collaboration contracts from Java code.

Method local variable:

The method local variables in the code are represented in the collaboration contract with the invocations that are assigned to them. Consequently, each time a method is invoked on a local variable, a cascaded invocation is represented in the collaboration contract. For example in method mouseDown a local variable myShape is used:

```
…
LineFigure myShape = (LineFigure)figure;
```

```
if (myShape.getNumberOfPoints() != 2)
      return true;
```

The invocation of getNumberOfPoints on myShape is represented in the collaboration contract with the cast cascaded invocation

(LineFigure)figure.getNumberOfPoints

As collaboration contract only deal with control flow information, this represents no change in the obtained semantic.

Method arguments

We did not have the need in the HotDrawMVC collaboration contract extraction, but if it were necessary to explicitly represent arguments of methods, acquaintances can be used. Tags can be used on such acquaintances to distinguish them from acquaintances representing instance variables.

Primitive types

When primitive types are used in Java code that need to be documented in a collaboration contract, *trivial* participants can be used. Trivial participants are participants without structure: no methods, no acquaintances. Even void keyword could be represented in a collaboration contract with a trivial participant. This is done just to keep the well-formedness of the collaboration contract, since all types should have a participant implementation in the collaboration contract.

Classes implementing interfaces

In the HotDrawMVC, interfaces are implemented by at most one class. But it may be the case that several classes implementing an interface need to be documented in a collaboration contract. Which classes implement a certain interface plays an important role at the moment of conflict detection (see chapter 6). Certain conflicts (like conflicts concerning the calling structure) require the examination of the specialisation clause associated to the invocation of a method on an acquaintance. When that acquaintance has an interface type, all the classes implementing that interface in the collaboration contract should be checked. For a more flexible conflict detection procedure, the user could specify which implementor classes should be checked for each interface type.

Chapter 5 - Operators on extended collaboration contracts

In the previous chapter we defined our extended model for collaboration contracts. We took the original definition from [Lucas97] an extended it with new modelling elements needed to express features at the source code level.

But this is not enough to define a complete extension to the reuse contracts model that deals with reuse and evolution at the implementation level. We need to specify which will be the modification operators used to express modifications to the defined collaboration contracts. It is not sufficient to use the existent modification operators defined in [Lucas97] because they do not manage the new features that we introduced in the collaboration contracts.

It is necessary to analyse which set of modification operators is best suited to express all the possible changes that can be made to the extended collaboration contracts. The set of modification operators needs to be complete, to allow us to represent any kind of modification that may occur when evolving or reusing source code. On the one hand the operators themselves have to be simple, with not too complex side effects, to make conflicts easy to understand and identify. If an operator involves changes to different parts of a collaboration contract, then its interactions when combined with other operators applied independently on the same collaboration contract are more difficult to determine. On the other hand, it is also necessary that the operators are general to be meaningful enough to represent changes to the modelled elements.

In works dealing with *schema modifications* such as [Barbedette91], the possible changes are classified in class contents modifications and inheritance graph modifications. This is a possible approach to be taken for the organisation of modifiers in this work, but we prefer to stay close to the definition of modifiers as present in the original definition of reuse contracts. This is consistent with our goal of defining a conservative extension of reuse contracts.

In the extended collaboration contracts we introduced new modelling elements as variants of the existing ones. We distinguished two kinds of participants, new kinds of relations between participants, and new kinds of operation invocations. In the same way we will introduce the modifications to these new modelling elements as variants of the existing modification operators. Later on, other new modifications can be incorporated into the model extended in this way.

As explained in chapter 2, in the original definition of reuse contracts there are four kinds of operators: extension, cancellation, refinement and coarsening. When these kinds of operators are applied at the context and participant level, they define eight different basic operators. We will respect this structural organisation for the modification operators of the extended model we define.

As we identify two kinds of participants, we will consequently have two variants for the context extension and cancellation operators. The four kinds of relations we define between participants will determine four variants for the context refinement and coarsening operators. At the participant level we distinguish two kinds of extension and cancellation, depending on the kind of participant which the operation is added to or removed from (class or interface participant), and three kinds of refinement and coarsening due to the three kinds of invocations we define (regular, self and super invocation).

For the sake of simplicity in the definition of modifications of the new modelling elements, we try to keep the operators as independent as possible from each other. For example, it is not desirable that a context extension (cancellation) involves a context refinement (coarsening). Even when in the original definition of reuse contract this is sometimes the case (for example, when a class is added all its acquaintances relations are added too), we try not to add new dependencies among operators. Thus, classes and interfaces for example must be added without any inheritance link. The same holds for cancellation: a context cancellation does not involve context coarsening; therefore inheritance links should be removed before a participant can be removed from a collaboration contract.

In the following sections we will explain the different modification operators, characterising the conditions that must hold in order for each modification to be valid, and the collaboration contract that results from the application of each operator. Although some insights into the formal characterisation of modification operators will be shown in this chapter, their complete formal definition can be found in the Appendix A.

Reuse contracts

A reuse contract has the same structure as in the original definition of [Lucas97]. Basically, it consists of a provider clause and a reuser clause. The provider clause is a collaboration contract that specifies how components interact to implement a certain behaviour. The reuser clause specifies how the collaboration contract is modified.

The complete definition of a reuse contract consists of:

- a name that identifies it
- a provider clause, that is a collaboration contract
- a contract type
- a contract type variant
- an extended reuser clause

We will describe the new elements in this definition of reuse contracts.

The type of a reuse contract can be any of the eight basic reuse contract types defined in [Lucas97]. In our definition the reuse contract also has a *variant*, that depends on the type of the reuse contract. The different reuse contract types with their associated variants are described in the following table.

Contract type	Variants	Meaning
Context extension	Extension with a class	Add a new class participant
	Extension with an interface	Add a new interface participant
Context cancellation	Remove a class	Remove a class participant
	Remove an interface	Remove an interface participant
Context refinement	Acquaintance refinement	Add an acquaintance relationship link
	Class inheritance refinement	Add a class inheritance link between 2 class participants
	Interface inheritance refinement	Add an interface inheritance link between 2 interface participants
	Implementation refinement	Add an implementation relation link from a class participant to an interface participant
Context coarsening	Acquaintance coarsening	Remove an acquaintance relation link
	Class inheritance coarsening	Remove a class inheritance link
	Interface inheritance coarsening	Remove an interface inheritance link
	Implementation coarsening	Remove an implementation link
Participant extension	Class extension	Add a new method to a class participant
	Interface extension	Add a new method to an interface participant
Participant cancellation	Cancellation in a class	Remove a method from a class participant
	Cancellation in an interface	Remove a method from an interface participant
Participant refinement	Regular refinement	Add a regular invocation to a specialisation clause
	Self refinement	Add a self invocation to a specialisation clause
	Super refinement	Add a super invocation to a specialisation clause
Participant coarsening	Regular coarsening	Remove a regular invocation from a specialisation clause
	Self coarsening	Remove a self invocation from a specialisation clause
	Super coarsening	Remove a super invocation from a specialisation clause

Reuser clauses depend on the type and variant of the reuser clause. They are also classified in context reuser clauses and participant reuser clauses. The former are those performing changes to the context of a collaboration contract (set of participants and relations between them). The latter are the ones that perform changes to the participant operations and specialisation interfaces.

Reuser clauses

We call the reuser clauses we define for our reuse contract *extended reuser clauses* because they consist of a reuser clause, as defined in [Lucas97], plus an additional *applicability condition*.

The *applicability condition* is a new feature that we introduce in the definition in order to provide more fine tuning in the specification of the evolution conflicts that are detected automatically. This comes from a change in the way of defining the applicability of reuser clauses.

As we described in chapter 2, for each reuser clause definition in the original definition of reuse contracts consists of three parts: the definition of the structure of the reuser clause, the conditions for applicability and the resulting collaboration contract after the application (provided that the conditions for applicability are valid). The conditions for applicability are defined for each kind of reuser class in order to ensure that the resulting collaboration contract is well- formed.

In this model we prefer to use the well-formedness checks already defined for collaboration contracts to establish if the result of applying a reuser clause to a collaboration contract yields a well-formed collaboration contract. We do this by defining a well-formedness check on reuse contracts, that checks that each of its components are well-formed, and that the result of applying the reuser clause on the provider clause yields a well-formed collaboration contract. The details are explained in the next section.

Due to the approach we take for stating the validity of the application of modification operators, we don't need to define complex conditions for the applicability of each reuser clause. It suffices with the definition of simple *compatibility* specifications that ensure basic things like, for example, that the elements referenced in the reuser clause exist in the collaboration clause. These are minimal conditions to be able to apply the reuser clause to a collaboration contract, but not to ensure that the result will be well-formed. The fact that the result is well-formed is established by applying it the collaboration contract's well-formedness check. The simpler requirements asked by *compatibility* specifications compared to the conditions for applicability in original reuse contracts will be illustrated in the definition of each reuser clause. The application definition of each reuser clause is based on the assumption that the reuser clause and the collaboration contract on which it is applied are compatible. If this is not the case, the application will give an error (returning for example a simple ill-formed resulting collaboration contract).

In this setting, the *applicability condition* that we define for a reuser clause is independent of the type and a variant of reuser clause (in contrast to the conditions for applicability defined in original reuser clauses). It specifies particular semantic requirements that should be fulfilled by a collaboration contract in order for the reuser clause to be semantically applicable. This applicability condition does not specify any well-formedness condition, but things that should be present –or absent- in the collaboration contract. The applicability condition for each instance of a reuser clause is defined by the reuser that is documenting the change to the base collaboration contract.

For example, the applicability condition of a reuser clause aimed at deleting old versions of methods would specify that the participant from where the method *oldMeth* is removed must have a certain new version method *newMeth*. Applicability conditions are useful because they can be used to specify any kind of condition, requiring positive or negative information in the original collaboration contract.

To make this clearer we will introduce the formal domains representing our extended reuser clauses.

ExtReuserClause = ReuserClause x ApplicabilityCond
ApplicabilityCond = CCStructure → Bool

The applicability condition is a function, defined for each element in the domain, that given a collaboration contract returns true if it fulfils the semantic condition. As a general function, any requirement concerning the received collaboration contract can be specified with it.

The reuser clauses are defined according to the type of reuse contract they correspond to. We classify them in two big groups, one is represented by participant reuser clauses and the other by the context reuser

classes. In each of these groups we identify the four basic kinds of operations: extension, cancellation, refinement and coarsening, yielding the eight kinds of basic reuser clauses.

ReuserClause = Participant_RClause + Context_RClause
Participant_RClause = PExt_RClause + PCanc_RClause + PRef_RClause + PCoar_RClause
Context_RClause = CExt_RClause + CCanc_RClause + CRef_Rclause + CCoar_Rclause

The structure for each of these reuser clauses and their variants will be presented in the following subsections. The first four explain participant reuser clauses and the last four explain context reuser clauses. For all of them, an intuitive informal definition will be given. For the formal definition of each kind of reuser clauses the reader is referred to the Appendix A.

Participant extension reuser clauses

Participant extension reuser clauses specify operations to be added to participants. As we define two kinds of participants, classes and interfaces, there will be two kinds (variants) of participant extension reuser clauses.

PExt_RClause = ClassExt_RClause + InterfExt_RClause

ClassExt_RClause and **InterfExt_RClause** have a different structure and consequently different compatibility specifications and the result of their application is obviously different.

Class extension reuser clauses

This kind of reuser clause adds operations to classes. Its structure specifies new operations with their corresponding return type and specialisation clauses, for one or more classes. The formal structure is as follows

ClassExt_RClause = ClassNam → (Signature x Spec_int)

Compatibility specification

A class extension reuser clause is compatible with a collaboration contract if

- all the classes it specifies are classes in the collaboration contract,
- the operations it specifies for each class are not already defined for that class in the collaboration contract.

Application definition

The *class extension* of a well-formed collaboration contract CC with a compatible class extension reuser clause R is another collaboration contract CC2, where everything except the classes specified in R remains the same as in CC. For each class c specified in R, c is defined in CC2 with:

- the same acquaintances as those it has defined in CC
- a signature equal to the union of the signature it has in CC and the signature it has in R
- a specialisation interface equal to the union of the specialisation interface it has in CC and the one it has in R

The compatibility specification states the minimal condition needed to ensure that the class extension can be performed. If the classes do not already exist then the result would in fact be a context extension, and this is not the goal of a class extension reuser clause. If the operations to be added to a class already exist, then their method return types and specialisation clauses in the resulting collaboration contract would be ambiguously determined (should the ones specified in the reuser clause or the pre-existent in the collaboration contract remain?).

It should be noted that the result of applying a class extension reuser clause to a collaboration contract can be an ill-formed collaboration contract. For example it can have one of the added methods in a class that inherits a method m but with a different return type, which means it has an ill-formed class hierarchy. This will be detected by applying the collaboration contract's well-formedness check on the result, so it doesn't need to be specified in applicability nor compatibility restrictions.

Interface extension reuser clauses

This kind of reuser clause add operations to interface participants. Its structure specifies new methods with their corresponding return types for one or more classes. The formal structure is as follows

InterfExt_RClause = InterfNam → Signature

Compatibility specification

An interface extension reuser clause is compatible with a collaboration contract if

- all the interfaces it specifies are interfaces in the collaboration contract,
- the methods it specifies for each interface are not already defined in that interface in the collaboration contract.

This compatibility condition is similar to the one for class extension reuser clause, but it compares the set of interfaces instead of the set of classes in the collaboration contract.

Application definition

The *interface extension* of a well-formed collaboration contract CC with a compatible class extension reuser clause R is another collaboration contract CC2, where everything except the interfaces specified in R remains the same as in CC. For each interface i specified in R, i is defined in CC2 with:

- a signature equal to the union of the signature it has in CC and the signature it has in R

The application definition is simpler than in class extension, because interface participants do not have a specialisation interface.

An example of a participant extension reuser clause compatible with the HotDrawMVC collaboration contract defined in the previous chapter adding a method hide with Void as a return type to interface LineFigure is depicted in Figure 5.1. We keep the graphical notation defined in [Lucas97] for the application of reuse contracts.

Figure 5.1

Participant cancellation reuser clauses

Participant cancellation reuser clauses specify operations to be removed from participants. The same as for participant extension, there will be two variants of participant cancellation reuser clauses.

PCanc_RClause = ClassCanc_RClause + InterfCanc_RClause
ClassCanc_RClause = ClassNam → Signature
InterfCanc_RClause = InterfNam → Signature

The class cancellation reuser clause and interface cancellation reuser clause have different structure since the former references classes from which operations should be removed and the latter references interfaces from which operations should be removed.

In contrast to the participant cancellation reuse contracts, here we only provide the minimal information necessary to identify the operations that should be removed, i.e. name of the participant and name of the

operation. We don't specify which return type nor which specialisation clause the operations to be removed should have. These elements are defined in the original formulation of reuse contracts as a means for determine the applicability of the reuser clause and consequently as a means for defining applicability conflicts. In our work we don't need to model those kinds of restrictions for the applicability of the reuser clause because, as explained before, this can be independently specified for each reuser clause by means of an applicability condition.

We define the compatibility specification and application definition of both kinds of participant cancellation reuser clauses together, because they have the same general structure and they can be defined for both of them referring to participants in general.

Compatibility specification

A participant cancellation reuser clause is compatible with a collaboration contract if

- all the participants it specifies are participants in the collaboration contract,

Application definition

The *participant cancellation* of a well-formed collaboration contract CC with a compatible participant cancellation reuser clause R is another collaboration contract CC2, where everything except the participants specified in R remains the same as in CC. For each participant p specified in R, p is defined in CC2 with:

- the same acquaintances as those it has defined in CC
- a signature equal to the signature it has in CC, where the operations it has associated in R have been removed
- in the case where p is a class participant, its specialisation interface is equal to the specialisation interface it has in CC, where the operations it has associated in R have been removed.

Again, the compatibility specification states the minimal condition needed to ensure that the class cancellation can be performed, but they are not requirements for well-formedness of the resulting collaboration contract. It should be noted that the result of applying a participant cancellation reuser clause to a collaboration contract can be an ill-formed collaboration contract. For example it can have operations whose specialisation clause includes operation invocations to removed operations. This will be detected by applying the collaboration contracts well-formedness check on the result, so it doesn't need to be specified in applicability nor compatibility restrictions. Similar ill-formedness collaboration contracts may result from every reuser clause application, and they can all be detected in the same way.

Participant refinement reuser clauses

Participant refinement and coarsening add and remove invocations to the specialisation clauses of operations. In our model the only kinds of participant that have specialisation interfaces are classes, so participant refinement and coarsening will always refer to class participants.

We distinguish between regular, self and super variants of participant refinement and participant coarsening reuser clauses, in order to better define the different conflicts that may appear when introducing or removing each kind of invocation.

In the formal model of collaboration contracts, for simplicity, we don't differentiate between the domains of regular, self and super operation invocations. As a consequence, the reuser clauses adding or removing the different kinds of operation invocations will all have the same form. In order to distinguish between the different kinds, we define functions that check the structure of the added or removed operation invocations to see if they are regular, self or super invocations. These functions are not used in the reuse contracts definitions but are needed in the identification of evolution conflicts, in order to distinguish between different variants of a conflict, as will be explained in next chapter. The complete definition of these functions can be found in the Appendix A, here we only show their domains

PRef_RClause = ClassNam → Spec_Int

is_regRef_RClause : PRef_RClause → Bool
is_selfRef_RClause : PRef_RClause → Bool
is_superRef_RClause : PRef_RClause → Bool

A difference between this structure and the structure defined in [Lucas97] for participant refinement and participant coarsening reuser clauses is that we don't specify a 'repeating' specialisation clause to be matched with the specialisation clause of the refined operation in the original collaboration contract. We define it in this way for the same reasons that we don't need to specify the return type nor specialisation clause of cancelled operations in participant cancellation reuser clauses. All the applicability restrictions can be defined, if needed, in the applicability condition of the reuser clause.

The same as in participant cancellation, here we define the compatibility specification and application definition in general for all kinds of participant refinement reuser clauses.

Compatibility specification

A participant refinement reuser clause is compatible with a collaboration contract if

- all the classes it specifies are classes in the collaboration contract,
- all the operations in the specialisation interface for each class c in the reuser clause are operations defined for class c in the collaboration contract.

Application definition

The *class refinement* of a well-formed collaboration contract CC with a compatible class refinement reuser clause R is another collaboration contract CC2, where everything except the classes specified in R remains the same as in CC. For each class c specified in R, c is defined in CC2 with:

- the same acquaintances as those that were defined in CC
- the same signature as it has defined in CC
- for the operations not referenced in R, a specialisation clause equal to the one it has defined in CC
- for the operations referenced in R, a specialisation clause equal to the union of the one it has defined in CC and the one specified in R.

The semantics of participant refinement is the same as in the original definition of reuse contracts. Regular participant refinements add regular invocations, self participant refinements add self invocations, and super refinement add super invocations. The procedure for computing all these modifications is the same. The well-formedness checking applied on the resulting collaboration contract afterwards will perform different checks on the different kinds of invocations. The checks performed on operation invocations according to their kind (regular, self or super) were explained in the previous chapter.

Participant coarsening reuser clauses

As explained before, we distinguish between regular participant coarsening, self participant coarsening and super participant coarsening reuser clauses. We do that through functions that check if the structure of the added operation invocations is of the corresponding kind.

PCoar_RClause = ClassNam → Spec_Int

is_regCoar_RClause : RegCoar_RClause → Bool :
is_selfCoar_RClause : SelfCoar_RClause → Bool :
is_superCoar_RClause : SuperCoar_RClause → Bool :

Also here we define the compatibility specification and application definition in general for all kinds of participant coarsening reuser clauses.

Compatibility specification

A participant coarsening reuser clause is compatible with a collaboration contract if

- all the participants it specifies are participants in the collaboration contract,
- all the operations in the specialisation interface for each participant p in the reuser clause are operations defined for participant p in the collaboration contract.

Application definition

The *participant coarsening* of a well-formed collaboration contract CC with a compatible participant coarsening reuser clause R is another collaboration contract CC2, where everything except the participants specified in R remains the same as in CC. For each participant c specified in R, c is defined in CC2 with:

- the same acquaintances as those it has defined in CC
- the same signature as it has defined in CC
- for the operations not referenced in R, a specialisation clause equal to the one it has defined in CC
- for the operations referenced in R, a specialisation clause equal to the one it has defined in CC where the operation invocations specified in R for that operation have been removed.

Note that even when the name of participant refinement and coarsening reuser clauses (and their compatibility and application definitions) refer to participants, the only kind of participants for which they make sense is for classes since interface participants do not have a specialisation clause. This is reflected in the domains of both reuser clauses, that specify class names.

Context extension reuser clauses

Context extension reuser classes add participants to a collaboration contract. As we define two kinds of participants in our collaboration contract model, this determines two variants of context extensions in our reuse contract model: context extensions adding classes and context extensions adding interfaces. The domain of context extension reuser clauses is shown below

CExt_RClause = CExtClass_RClause + CExtInterf_RClause
CExtClass_RClause = ClassNam → Class
CExtInterf_RClause =InterfNam → Interface

Context extensions adding classes specify a number of class names with their corresponding class structures that should be added to a collaboration contract. Context extensions adding interfaces instead, specify interface names with their corresponding structures (that are simply signatures) that should be added to a collaboration contract.

Thanks to the approach we take of defining the well-formedness of a reuser clause application in an independent check done after the application, this allows us to also define a general compatibility

specification and application definition for both variants of context extension reuser clauses (we refer generically to participants that are added).

Compatibility specification

A context extension reuser clause is compatible with a collaboration contract if

- all the participants it specifies are not already defined in the collaboration contract,

Application definition

The *context extension* of a well-formed collaboration contract CC with a compatible context extension reuser clause R is another collaboration contract CC2, where everything except the definition of the participants remains the same as in CC. The participants definition of CC2 is equal to the union of the participants of CC and the participants defined in R.

The result of applying a context extension to a collaboration contract is another collaboration contract where the relationships between existent participants remain the same, and the set of participants is extended with the added ones. Observe that the compatibility specification stating that the added participants are different from the existent ones is necessary to avoid ambiguity in the definition of the resulting set of participants.

Note the atomicity of context extensions, that only add participants without any hierarchy or implementation relation. The addition of such relations should be specified separately with a context refinement.

Context cancellation reuser clauses

Context cancellation reuser classes remove participants from a collaboration contract. The same as for context extension, we distinguish between two variants of context cancellations in our reuse contract model: context cancellations removing classes and context cancellations removing interfaces. The domain of context cancellation reuser clauses is shown below

CCanc_RClause = CCancClass_RClause + CCancInterf_RClause
CCancClass_RClause = P(ClassNam)
CCancInterf_RClause = P(InterfNam)

Context cancellations removing classes specify a set of class names to be removed from a collaboration contract. Context cancellations removing interfaces instead, specify a set of interface names that should be added to a collaboration contract. In contrast to the original definition of reuse contracts, and with the same arguments given for participant cancellation reuser clauses, here we only need to specify the minimal information needed to identify the participants to be removed from the collaboration contract that is, the names of those participants. Further restrictions for the application of each context extensions can be specified in the applicability condition.

In these reuser clauses we can also generalise the definitions of compatibility and application for both variants.

Compatibility specification

A context cancellation reuser clause is compatible with a collaboration contract if

- all the participants it specifies are participants defined in the collaboration contract

Application definition

The *context cancellation* of a well-formed collaboration contract CC with a compatible context cancellation reuser clause R is another collaboration contract CC2, where everything except the definitions of the participants remains the same as in CC. The participants definition of CC2 is equal to the participants component of CC where the participants whose name appear in R have been removed.

The atomicity of context cancellation reuser clauses is the same as for context extension. They only remove participants leaving untouched the relationships between them. The removal of hierarchy links or implementation relation links should be specified separately with a context coarsening reuser clause. It should be noted that the relationships should be removed before the participants are removed, otherwise the result of applying the context cancellation will not be well-formed.

Context refinement reuser clauses

Context refinement reuser classes add participant relations to a collaboration contract. In our model for collaboration contracts we have four kinds of relationships between participants: acquaintance relations, class inheritance relations, interface inheritance relations and implementation relations. This determines corresponding variants in context refinement reuser clauses.

CRef_RClause = AcqRef _RClause + ClassHierRef_RClause + InterfHierRef _RClause
+ ImplRelRef _RClause
AcqRef _RClause = ClassNam → Acq_Clause
ClassHierRef_RClause = ClassHierarchy
InterfHierRef _RClause = InterfHierarchy
ImplRelRef _RClause = ImplemRelation

For the formal structure of class inheritance refinement reuser clauses, we can use the domain that defines the inheritance hierarchy in the collaboration contract (**ClassHierarchy**). The same can be done with interface inheritance and implementation relation refinement reuser clause, because the three of them introduce relations that are explicitly represented in the collaboration contract structure. On the contrary, this cannot be done for acquaintance refinement reuser clause. As we respect the representation of acquaintances defined in the original definition of reuse contracts, they remain part of the participant's structure. Therefore acquaintance refinement reuser clauses specify acquaintance clauses that should be added to one or more classes in the collaboration contract.

As the different variants of context refinement reuser clauses affect different aspects of the collaboration contract structure, their compatibility specification and application definition should be specified separately.

Acquaintance refinement reuser clauses

Compatibility specification

An acquaintance refinement reuser clause is compatible with a collaboration contract if

- all the classes it specifies are classes defined in the collaboration contract,
- the acquaintance names it specifies for each class c are not already acquaintance names for class c in the collaboration contract.

Application definition

The *acquaintance refinement* of a well-formed collaboration contract CC with a compatible acquaintance refinement reuser clause R is another collaboration contract CC2, where everything except the classes specified in R remains the same as in CC. For each class c specified in R, c is defined in CC2 with:

- the same signature as the one it has defined in CC
- the same specialisation interface as the one it has defined in CC
- an acquaintance clause equal to the union of the acquaintance clause it has in CC and the acquaintance clause it has in R.

The result of applying an acquaintance refinement reuser clause, e.g. by adding acquaintances whose acquaintance types are not participants of the collaboration contract, will result in an ill-formed collaboration contract detected by the well-formedness check made afterwards.

Class inheritance refinement reuser clauses

Compatibility specification

A class inheritance refinement reuser clause is compatible with a collaboration contract if

- the classes to which it adds a superclass link do not already have a superclass in the collaboration contract.

The requirement imposed by the compatibility specification ensures that the class hierarchy of the resulting collaboration contract is a function i.e. there is at most one superclass for each class.

Application definition

The *class inheritance refinement* of a well-formed collaboration contract CC with a compatible class inheritance refinement reuser clause R is another collaboration contract CC2, where everything except the class hierarchy specified in R remains the same as in CC. The class hierarchy relation of CC2 is equal to the union of the class hierarchy relation of CC and R.

Also here, if the class hierarchy specified by the reuser clause refers for example to classes that are not defined in the collaboration contract, the problem will be detected by the well-formedness check applied to the resulting collaboration contract.

Interface inheritance refinement reuser clauses

Compatibility specification

An interface inheritance refinement reuser clause is compatible with a collaboration contract if

- the superinterface links it adds are do not already exist in the collaboration contract.

Application definition

The *interface inheritance refinement* of a well-formed collaboration contract CC with a compatible interface inheritance refinement reuser clause R is another collaboration contract CC2, where everything except the interface hierarchy specified in R remains the same as in CC. The interface hierarchy relation of CC2 is equal to the union of the interface hierarchy relation of CC and R.

Also here, any problem introduced by the reuser clause in the interface hierarchy of the collaboration contract will be detected in the well-formedness check.

Implementation relation refinement reuser clauses

Compatibility specification

An implementation relation refinement reuser clause is compatible with a collaboration contract if

- the implementation links it adds are do not already exist in the collaboration contract.

Application definition

The *implementation relation refinement* of a well-formed collaboration contract CC with a compatible implementation relation refinement reuser clause R is another collaboration contract CC2, where everything except the implementation relation specified in R remains the same as in CC. The implementation relation of CC2 is equal to the union of the implementation relation of CC and R.

Also here, any problem introduced by the reuser clause in the implementation relation of the collaboration contract will be detected in the well-formedness check.

Context coarsening reuser clauses

Context coarsening reuser clauses remove participant relation links from a collaboration contract. The same as for context refinement, the four kinds of relationships between participants in our collaboration contracts model determine corresponding variants in context coarsening reuser clauses.

CCoar_RClause = AcqCoar _RClause + ClassHierCoar_RClause
+ InterfHierCoar _RClause + ImplRelCoar _RClause
AcqCoar _RClause = ClassNam → P(AcqNam)
ClassHierCoar_RClause = P(ClassNam)
InterfHierCoar_RClause = InterfHierarchy
ImplRelCoar_RClause = ImplemRelation

Context reuser clauses specify the minimal information needed to identify the relation links to be removed. In the case of acquaintance removal they only specify the names of the acquaintances that should be removed from each class. In the case of class inheritance link removal, only a set of classes whose superclass link should be removed are specified. In the case of interface hierarchy it is possible to remove several superinterfaces of an interface, so the corresponding reuser clause is an interface inheritance relation specifying all the interface parents to be removed from a number of interfaces. A similar argument justifies the fact that the structure of an implementation coarsening reuser clause is an implementation relation, it specifies the implementation links to be removed.

As the different variants of context coarsening reuser clauses affect different aspects of the collaboration contract structure, their compatibility specification and application definition should be specified separately.

Acquaintance coarsening reuser clauses

Compatibility specification

An acquaintance coarsening reuser clause is compatible with a collaboration contract if

- all the classes it specifies are classes defined in the collaboration contract,

Application definition

The *acquaintance coarsening* of a well-formed collaboration contract CC with a compatible acquaintance coarsening reuser clause R is another collaboration contract CC2, where everything except the classes specified in R remains the same as in CC. For each class c specified in R, c is defined in CC2 with:

- the same signature as the one were defined in CC
- the same specialisation interface as the one it has defined in CC
- an acquaintance clause equal to the acquaintance clause it has in CC where the acquaintances whose names are specified in R for c have been removed.

Ill-formedness problems in the resulting collaboration contract, due to for example the deletion of an acquaintance a in a class for which some operations specialisation clauses contain operation invocations on a, are detected as well in the well-formedness check.

Class inheritance coarsening reuser clauses

Compatibility specification

A class inheritance coarsening reuser clause is always compatible with a collaboration contract.

Application definition

The *class inheritance coarsening* of a well-formed collaboration contract CC with a compatible class inheritance coarsening reuser clause R is another collaboration contract CC2, where everything except the class inheritance hierarchy specified in R remains the same as in CC. The class inheritance hierarchy relation of CC2 is equal the class inheritance hierarchy relation of CC, where all the class names specified in R do not have any superclass.

In this case the well-formedness check performed afterwards over the resulting collaboration contract will detect problems for example caused by the reuser clause removing a superclass link from a class that contains some operations whose specialisation clause contains a super invocation. Other well-formedness problems that may be introduced by class inheritance coarsening reuser clauses, and that are also detected by the checking, are for example when the deletion of a superclass link causes a class to stop inheriting an operation that was required for it to implement a given interface. In this case the modification will cause an ill-formedness in the implementation relation.

Interface inheritance coarsening reuser clauses

Compatibility specification

An interface inheritance coarsening reuser clause is compatible with a collaboration contract if:

- the subinterfaces it specifies (those in its domain) are subinterfaces in the interface hierarchy of the collaboration contract.

Application definition

The *interface inheritance coarsening* of a well-formed collaboration contract CC with a compatible interface inheritance coarsening reuser clause R is another collaboration contract CC2, where everything except the interface inheritance hierarchy specified in R remains the same as in CC. The implementation relation in CC2 is formed as follows:

- for every subinterface i in CC that is not specified as subinterface in the domain of R, i's superinterface links remain the same in CC2

- for every subinterface i in the domain of R:
  - i has superinterface links equal to the superinterface links specified in CC2 where the superinterface links specified for R have been removed.

    (if the resulting set of superinterface links for i is empty then it is not in the domain of the superinterface hierarchy of CC2)

Implementation relation coarsening reuser clauses

Compatibility specification

An implementation relation coarsening reuser clause is compatible with a collaboration contract if:

- the implementors it specifies have implementation links in the collaboration contract.

Application definition

The *implementation relation coarsening* of a well-formed collaboration contract CC with a compatible implementation relation coarsening reuser clause R is another collaboration contract CC2, where everything except the implementation relation specified in R remains the same as in CC. The implementation relation in CC2 is formed as follows:

- for every implementor class c in CC that is not specified as implementor in the domain of R, c's implementation links remain the same in CC2
- for every implementor class c in the domain of R:
  - i has an implementation links set equal to the implementation links specified in CC2 where the implementation links specified for R have been removed.

    (if the resulting set of implementation links for i is empty then it is not in the domain of the implementation relation of CC2)

Also here, any problem introduced by the reuser clause in the implementation relation of the collaboration contract will be detected in the well-formedness check.

Reuse contracts well-formedness

As we stated in beginning of the previous section and we described for each kind of reuser clause, the validity of applying a reuser clause to a collaboration contract is determined by a well-formedness check on the resulting collaboration contract.

In the same way as in the previous chapter we defined a well-formedness function for collaboration contracts that proceeds by checking the well-formedness of each element inside it, and also by checking the consistency among elements. In a similar way we define a well-formedness check on reuse contracts that checks the well-formedness of each of its component and then the consistency among them.

The well-formedness function checks that the provider clause is a well-formed collaboration contract. After that it checks that contract type and variants of the reuse contract and the reuser clause are consistent. Then it checks that the provider and reuser clause are consistent. This consists of two basic checks:

- check that the applicability condition of the reuser clause is valid on the provider clause: as we define the applicability condition as a function that given a collaboration contract returns a boolean value, this step is trivial
- check that the result of applying the reuser clause on the provider collaboration contract is a well-formed collaboration contract: this is done by applying the reuser clause on the provider clause (using the application definition of the reuser clause, as defined in the previous section), and then applying to that result the collaboration contract well-formedness function described in the previous chapter.

Note that because of the compatibility specifications that we defined for each reuser clause and because of the well-formedness check performed on the resulting collaboration contract, we do not need to define well-formedness checks for reuse clauses independently. Ill-formedness in reuser clauses will be detected by compatibility specifications or well-formedness check on the resulting collaboration contract.

Chapter 6 – Evolution conflicts with extended reuse contracts

The problems of making parallel modifications to software artefacts discussed in [Lucas97] and [Mens99a] are particularly evident at the implementation level. In collaborative development contexts where source code is modified by different teams simultaneously, the risk of inconsistencies when combining parallel changes to the same component are higher. Modifications usually occur for bug fixing, new insights in the design (like refactoring changes [OJ93]), or for extensions in the functionality of the component. When working with frameworks, the changes introduced to the components of the base framework have a potential conflicting impact in the different reuser applications that instantiate or customise the framework. Rigorous models that can handle the evolution problems arising with code changes in these contexts are very much valuable. Our work proposes a way to adapt the general reuse contracts model to be able to apply them to source code evolution.

In chapter 4 we defined our model of collaboration contracts, that provide a structured documentation of source code written in typed object-oriented languages. In chapter 5 we identified operators that are used to document modifications performed on the defined collaboration contracts. In this chapter we will present a characterisation of the problems of evolution at the implementation level that can be detected by using the previously presented documentation.

We find that the new elements introduced in our collaboration contract model give an added expressive power that serves to identify and detect new kinds of evolution conflicts that may come up when independently evolving software components. Most of these new conflicts are related to the inheritance hierarchy structure that we model between participants. The possibility to represent such inheritance hierarchies inside collaboration contracts, and the structured documentation of changes to participants in those inheritance hierarchies allows to automatically identify problems arising with conflicting modifications. These problems are not detectable by compilers of statically-typed programming languages because they appear in hierarchies that are structurally correct, and they are hard to detect manually due to the complex control-flow dependencies usually present in inheritance hierarchies, specially if they are well-factored. One of the main contribution of this work is the definition of a way to detect these kind of conflicts.

We will begin by defining what we consider a conflict in terms of the previously defined elements. Then we differentiate between applicability and evolution conflicts. After that we propose a classification for all the conflicts found that extends the classification defined in [Lucas97]. Finally we will describe the different kinds of conflicts we distinguish.

A classification of conflicts

A conflict occurs when two (or more) valid modifications made to the same collaboration contract interact. An example of this is depicted in Figure 1.1, where the two modifications are a reuse step and an evolution step. In terms of the reuse contracts we defined, this means that both reuse contracts have the same provider clause and their reuser clauses affect common elements in that collaboration contract.

There are two general sorts of conflicts (following the definition of [Mens99a]):

- applicability conflicts (or structural conflicts)
- evolution conflicts (or behavioural conflicts)

The former represents conflicts that affect the well-formedness of a collaboration contract. Applicability conflicts detect "structural or syntactic inconsistencies" [Mens99a] when the combination of two valid operators yields a problem in the structure of the result. Applicability conflict occur when two independent evolution steps cannot be merged because the items needed by one of them are modified by the other. Thus, after the application of one of the operators, the other one cannot be applied. Applicability conflicts can in general be detected by compilers of statically typed languages or other kinds of checkers provided by tools and languages. A simple example is when one operator removes a participant from the collaboration contract while the other adds an acquaintance with that type to another participant. In this case, the independent application of the two modifications is valid, but the combination of both is not consistent because the application of the second one is no longer valid after the first was performed (for it adds a reference to a participant that does not exist).

Evolution conflicts are more subtle conflicts that arise, they detect "behavioural or semantic inconsistencies" [Mens99a]. An evolution conflict occurs when the combination of two independent modification produces undesired effects that were not expected by the evolvers performing the modifications, but that nevertheless lead to structurally correct results. Evolution conflicts are hard to detect because they don't involve structural problems in the resulting collaboration contract; they occur when the result of the combination is well-formed but it contains interactions among the affected elements that were not expected by any of the modifications individually. Also, it is difficult to determine automatically when new interactions appearing in the combination of independent modifications are really undesired by one or both evolvers. Therefore we consider a broad variety of unintended interactions as evolution conflicts, and then the reuser will finally decide whether they correspond to real problems or not.

The same as in [Lucas97], most kinds of conflicts can be detected by only looking at the type of the applied operators and the elements it affects in the base collaboration contract, but for some of the conflicts it is necessary to also analyse the resulting collaboration contract as well.

For the analysis of the different conflicts we started by checking if the conflicts detectable by the original definition of reuse contracts are still conflicts in our model. Since our model is a conservative extension of the original reuse contracts, this is the case. But as we introduced different variants for the modelling elements of collaboration contracts, and they determined variants in the reuser operators, consequently we find also variants for some of the conflicts formulated in [Lucas97]. However there exist other conflicts in our model that are new, they cannot be defined as variants in the original reuse conflicts and signal possible problematic situations that are not automatically detectable by usual checkers.

We defined a classification for the conflicts we identify, that is based on the classification defined in [Lucas97] and adds new categories to embody the new conflicts that are not present in the original definition. The categories defined in the original formulation of reuse contracts are:

- *interface conflicts*: they refer to conflicts that may appear in the interface, i.e. conflicts of operation names, participant names, etc. occurring when two modifications introduce an extra operation or participant with the same name. To avoid confusion with interface participants we will rename interface conflicts to *duplicate modification conflicts*.
- *dangling reference conflicts*: conflicts caused when one operator removes some item that the other operator refers to
- *conflicts concerning the calling structure*: conflicts that concern the assumptions made about the calling structures of participants.

Additionally, we introduce the following new categories:

- hierarchy conflicts: conflicts affecting the class or interface inheritance relation
- implementation conflicts: conflicts signalling problems in the implementation relation

- type cast conflicts: conflicts detecting problems with the type casts. This category contains one conflict representing a problem that is not checked by the type checker of Java.

The conflicts identified in each category are shown in the following table:

Duplicate modification conflicts	**Dangling reference conflicts**	**Conflicts concerning calling structure**	**Hierarchy conflicts**	**Implementation conflicts**	**Type cast conflicts**
Double operation name	Dangling operation	Operation capture	Invalid acquaintance redefinition	Missing implementation	Invalid type cast
Double participant name	Dangling participant	Inconsistent operations	Inconsistent return-types	Redundant specification	
Double operation invocation	Dangling relationship	Unanticipated recursion	Cyclic inheritance hierarchy	Redundant implementation	
Double participant-relationship			Double inherited interface		
			Masked operation		
			Unanticipated redirection		

The most interesting new conflicts detected by our model are the evolution conflicts concerning class and interface hierarchies (specially masked operation and unanticipated redirection) and the evolution conflicts concerning the implementation relation (redundant specification and redundant implementation). There are also interesting variants of operation capture and unanticipated recursion that are detected by our model and could not be expressed in the original definition of reuse contracts. The remaining conflicts are also important for our work as they express interesting problems detected in [Lucas97], or applicability conflicts detecting structural problems.

All of the conflicts detect inconsistencies that may be unnoticeably introduced by cascaded operation invocations. Adding a cascaded invocation in one class of a collaboration contract may introduce dependencies with previously unrelated participants. By providing cascaded invocation in our model we are representing the fact that a class is potentially dependant on any participant in the transitive closure of the return types of its methods. We reformulate all the conflicts involving operation invocations, so that they detect inconsistencies that these subtly introduced dependencies may have when combined with other kinds of modifications. The issue cascaded invocations and transitive closure of dependencies of a participant is further analysed in the last subsection of this chapter.

From now on we will describe each kind of conflict we define. For each of them, we will describe the problem situation (with an example for the conflicts most relevant for this work), we will give the different combinations of operators that yield that conflict and suggest some possible solutions for it. In Appendix B we present a general table that points out for each combination of modification operators, which are the conflicts that may arise.

For defining every conflict we consider two modifiers (operators) M1 and M2 that are performed independently on a collaboration contract called the *base collaboration contract*.

Duplicate modification conflicts

For this category we identify basically the same conflicts as in the original definition of reuse contracts. They are all applicability conflicts, that could be detected by existing checking tools, so we will analyse them in deep detail.

Double operation name conflict

These conflicts signal problems that can occurs when two independent modifications are performed that add an operation with the same name to the same participant.

Definition:

A double **operation name conflict** occurs when M1 is a participant extension adding operation with name m to a participant p and M2 is a participant extension of the same kind adding an operation also named m to the same participant p.

It should be noted that the operators can be both class extensions or interface extensions. The only thing that is needed to detect a double operation name conflict is that the name of the added operation is the same, the other data (return type, specialisation clause) can be the same or different.

This is an applicability conflict because after applying M1 on the base collaboration contract, the application of M2 is no longer valid, or vice versa.

Double operation name conflicts can be solved by renaming the operation added by one or both of the operators, or by 'hiding' one or both of the operations. By hiding we refer to techniques provided by programming languages like Java that support the qualification of methods with modifiers such as 'private', to make them visible only inside the class owning the method. The introduction of these modifiers in reuse contracts is defined in [Cornelis97].

Double participant name conflict

These conflicts signal problems that can occurs when two independent modifications are performed that add a participant with the same name to a collaboration contract.

Definition:

A **double participant name conflict** occurs when M1 is a context extension adding a participant p and M2 is a context extension adding a participant also named p.

It should be noted that the operators can be context extension adding a class or context extension adding an interface, they can both belong to the same or different variants of context extensions.

This is an applicability conflict because after applying M1 on the base collaboration contract, the application of M2 is no longer valid, or vice versa, because there already exists a participant with that name.

The solution to double participant name conflicts will depend highly on the variant of context extension that the operators belong to. If they add different kinds of participant then a renaming of one of them would probably be a good solution. If they both add the same kind of participant, then renaming one of them, merging both, or not adding any more one of them could be possible solutions, depending on the purposes for which each of them were introduced.

Double operation invocation conflict

Double operation invocation conflict and participant relationship conflict are slightly different from the two previous conflicts. They don't occur because two independent modifications add the same item in a

collaboration contract, but because they alter the dependencies of the same item in the same or different ways.

Double operation invocation conflicts signal problems that can occur when two independent modifications affect the specialisation clause of the same method in a class. It is possible that this is a problem, because the changes they produce may be related.

Definition:

An **double operation invocation conflict** occurs when M1 is a participant refinement or coarsening affecting method m in class c, and M2 is a participant refinement or coarsening also affecting method m in class c.

The operators can be participant refinements or coarsening of any variant: regular, self or super.

In the original definition of reuse contract, double operation invocation conflicts (called operation invocation conflict) are detected as applicability conflict through the use of repeating specialisation clauses in participant refinement and participant coarsening reuser clauses. In our model we don't represent such repeating specialisation clauses, but provide the more flexible mechanism of applicability conditions defined by the reuser. So, double operation invocation conflicts will only be signalled when the assumptions explicitly stated by one or both evolvers (in the applicability condition) are breached in the combination with the other modification on the same specialisation clause. That is, double operation invocation conflicts will occur when the applicability condition of one operator is no longer valid after the other operator was applied to the base collaboration contract.

Double participant-relationship conflict

Double participant-relationship conflicts signal problems that can occurs when two independent modifications add or remove the same link between participants. Thus, this conflict occurs between two context refinements or two context coarsenings. It is an applicability conflict since after one operator was applied the second one is no longer applicable, either because the link does not exist any more (in the case of context coarsenings) either because the link already exist (in the case of context refinement).

Definition:

An **Double participant-relationship conflict** occurs when M1 is a context refinement (coarsening) adding (removing) a link between two participants p1 and p2, and M2 is also a context refinement (coarsening) adding (removing) the same link between p1 and p2.

The operators can be both acquaintance refinement, class inheritance refinement, interface inheritance refinement, acquaintance coarsening, class inheritance coarsening or interface inheritance coarsening.

Dangling reference conflicts

This category groups conflicts that occur when one modification removes an element while the other modification continues to refer to it. All dangling reference conflicts can be checked by only comparing the type and contents of the applied modifiers.

They are also applicability conflicts. They can be detected by compilers of typed languages, but in dynamically typed languages like Smalltalk, they remain undetected. They give rise to error messages like 'Message not understood'.

For all these conflicts, the possible solutions depend on the intention of the evolvers, without knowing this intention the only hint that can be suggested is not to perform any one of the two modifications. Further study on conflict resolution needs to be done, that analyses and characterise those possible intentions, to be able to suggest solutions to this conflict.

Dangling operation conflict

When one operator removes an operation from a participant, class or interface, while the other operator makes a modification involving that operation, like adding an invocation to it, or remove an invocation from it.

These are applicability conflicts because after any one of the modifications is applied, the other cannot be applied anymore (or it would give raise to an ill-formed collaboration contract).

Definition

A **dangling operation conflict** arises when M1 is a participant cancellation removing an operation m in a participant p and M2 is any of the following

- a participant cancellation, also removing operation m in p
- a regular participant refinement adding in any class:
  - an simple invocation to m in an acquaintance of type p, or
  - a cascaded invocation including m that binds to m in p

or, in the case where p is a class:

- a regular participant refinement adding a regular invocation in the specialisation clause of m in p
- a self participant refinement adding a self invocation to m in the specialisation clause of an operation in p, or adding a self invocation to any operation in the specialisation clause of m.
- a participant coarsening removing an invocation from the specialisation clause of m in p

It is important to clarify that every time we refer to "a cascaded invocation including m that binds to m in p" we mean the cascaded invocation contains m among the invoked methods, and that the receiver of m in is p or a descendant from p that directly inherits m from p. This counts as well for type casts to p or to a descendant that directly inherits m from p.

Dangling late bound operation conflict

A special variant of dangling operation conflicts are those where the reference to the removed operation is bound to it through the late binding mechanism. We distinguish this variant because it represents conflicts that are not easy to detect at first sight, because they deal with late binding in potentially very deep inheritance hierarchies. Also because late binding is a feature newly introduced in our model and not represented in the original definition of reuse contracts.

Definition

A **dangling late bound operation conflict** arises when M1 is a participant cancellation removing an operation m in a class participant p and M2 is any of the following

- a self participant refinement adding a self invocation to m in an operation of a descendant of p that inherits m directly from p[5],
- a super participant refinement adding a super invocation that binds to m in p[6], in a descendant of p

Dangling late bound operation conflicts concerning self refinements are quite likely to occur unnoticed, because the self invocation is added to a class different than the one whose method is cancelled.

Dangling participant conflict

These conflicts signal problems appearing when an operator removes a participant and the other operator continues referencing to it.

Definition

A **dangling participant conflict** arise when M1 is a context cancellation removing a participant p and M2 is any kind of operator referencing p, except a context extension adding p.

Any possible operator referencing p will conflict with M1 except the addition of a participant p, which would not applicable on the base collaboration contract, then it is not considered a conflict.

Dangling relationship conflict

These conflicts signal problems when one evolver removes a relationship link between two participants, and the other evolver independently performs any modification that relies on that relationship link. The name in [Lucas97] for these conflicts is 'dangling acquaintance reference', because the only kind of links between participants in the original definition of reuse contracts are acquaintance links. As in our model there are four kinds of relations between participants, we find four corresponding variants of this kind of conflicts, that are defined below.

Dangling acquaintance conflict

This conflict occurs when the relation link removed is an acquaintance link. It corresponds to the one originally defined in [Lucas97] and described in chapter 2, but our definition is slightly different. A dangling acquaintance conflict is detected when one modification removes an acquaintance relationship while the other continues to refer to it. This clearly occurs when the first operator is a context coarsening and the second is a participant extension or participant refinement adding an invocation referencing the removed acquaintance. The definition of this conflict in [Lucas97] says that the second operator can also be a participant coarsening referencing the removed acquaintance, but this cannot occur because if there exists an invocation referencing the acquaintance then the first operator would not be applicable. Consequently our definition of the dangling acquaintance conflict do not consider the participant coarsening as one of the operators.

---

[5] We say that a descendant d from participant p 'inherits an operation m directly from p' when operation m is defined in p, and nor d nor any participant in between (in the hierarchy) redefines it. This is, when any invocation to m in d is bound to the definition of m in p.

[6] Even when it is not enforced by the operators definition in our model, here we will consider that super invocations are not crossed. That is, in the specialisation clause of an operation m any super invocation invokes only the same method m.

Definition

A **dangling acquaintance conflict** occurs when M1 is a context coarsening removing an acquaintance, and M2 is a regular participant refinement or class extension referencing the removed acquaintance.

Dangling subclass conflict

This is the variant of dangling relationship conflict where the context coarsening is a class inheritance coarsening.

When an evolver removes a superclass link from a class c to a class c2, many problems may arise with parallel modifications because all the methods and acquaintances inherited from c2 in the base collaboration contract can no longer be referenced in c or descendants of c. For example, the second evolver can add an invocation to an operation m in c or any descendant of c that binds to an m in c2. The combination of both modifications will result in an operation invocation on a participant that does not understand that operation (and in the base collaboration it did because it inherited it from c2). The same can occur with a regular invocation on an instance variable inherited from c2.

Definition

A **dangling subclass conflict** occurs when M1 is a class inheritance coarsening removing an super class link from subclass c to superclass c2, and M2 is a participant refinement or extension referencing an operation m that binds to operation m in c2, or an acquaintance inherited from c2.

Dangling subinterface conflict

This is the variant of dangling relationship conflict where the context coarsening is a interface inheritance coarsening.

This conflict signals problems similar to the ones detected in dangling subclass conflict, where the participants that stop inheriting operations are interfaces. The problem arises because the first evolver removes an interface hierarchy link between subinterface i and superinterface i2, and the second adds a regular invocation to an acquaintance of type i (or any descendant of i) that inherits the invoked operation from i2.

Definition

A **dangling subinterface conflict** occurs when M1 is an interface inheritance coarsening removing an interface hierarchy link from subinterface i to superinterface i2, and M2 is a participant refinement or extension referencing an operation m that binds to operation m in i2.

Conflicts concerning the calling structure

This category groups all the conflicts concerning essentially the calling structure of participants. All conflicts concerning the calling structure are evolution conflicts, this makes them more interesting for the reuse contracts model because they cannot be detected by existent checking tools. For those conflicts we identify variants that include the hierarchy structures represented in our model of collaboration contracts.

As the name suggest, the conflicts concerning the calling structure affect the operation invocations of a participant. The only kinds of participants in our model that can contain operation invocations are class participants. Consequently, all the conflicts described in this category are devoted to class participants.

The first two kinds of conflicts in this category (operation capture and inconsistent operations conflicts) can be determined, as in previous categories, by simply examining the type of the involved operators and the elements they affect. For the last kind of conflicts concerning the calling structure (unanticipated recursion), it is necessary to examine the base collaboration contract.

Operation capture

An operation capture occurs when one evolver adds an invocation to an operation m from an operation n, and the other evolver independently performs changes in the specialisation clause of n. There are no structural problems when combining both modifications, but a potentially unexpected interaction of both can arise. The first evolver invoked an operation n that he knew performed a certain behaviour. But the second evolver alters that behaviour of n. Then it is possible that this change causes problems in the behaviour of operation m. The example presented in [Lucas97] for this conflict is shown in Figure 6.1.

Figure 6.1

The example represents an ATM contract in a banking application, where two participants are modelled: a bank and the consortium. In Figure 6.1 operator M1 adds an invocation to calcCode in operation verifyCard

of class Bank, and operator M2 add an invocation to verifyCard in verifyTrans of class Consortium. When combining both modifications it results that operation calcCode is called every time verifyTrans is called. It is said that operation verifyCard gets captured, because the changes made to it by M1 have an influence on M2. This may be intentional or may represent an unexpected interaction between the changes of both operators, as it is the case if it is not necessary to invoke calcCode for every verifyTrans.

In the original model of reuse contracts two kinds of operation capture conflicts are distinguished. One of them is 'regular operation capture', that represents the kind of problems described in the previous description. The other is 'accidental operation capture', that occur when the operation m did not exist in the base collaboration contract and is added by both operators simultaneously. Consequently, accidental operation capture always involve a double operation name conflict. As double operation name conflicts are applicability conflicts, what means they are easier to detect, we decide to only consider regular operation capture as our operation capture conflicts.

We will refer to operation capture as the kind of conflicts that in [Lucas97] is called 'regular operation capture'. In our model we use 'regular' to identify one of the variants of operation invocations we introduce, namely acquaintance operation invocations. So we rename 'regular operation capture' as simply 'operation capture'. We will call 'regular operation capture' one of the variants of operation capture conflicts.

In general we identify an operation capture conflict as occurring between a participant refinement (M2 in the example above) and a participant refinement or coarsening (M1 in the example above). When analysing the possibilities for these combinations of reuser clauses in our model, we identify three variants of operation capture conflicts, depending on the kind of refinement which the first reuser clause belongs to (regular, self or super variants). Then we identified a fourth variant of the conflict that is a combination of two of the other variants. We describe each of them below.

For a each of the variants, the participant refinement (M2) can also add to in any participant a cascaded invocation referencing the operation affected by M1. In this way operation captures conflicts also detect inconsistencies that may appear when the specialisation clause of an operation (indirectly) invoked by a cascaded invocations is changed.

For all variants of operation capture the solutions are similar, and depend on the particular situation. If the influence of the first modification on the second is a desired effect, then there is no problem and nothing needs to be changed. If that is not the case then either the implementation of the captured operation needs to be changed so that it performs the desired behaviour, or it can be encapsulated so that it is no longer captured.

Regular operation capture

This is the case where the first modification operator is a regular refinement. This variant corresponds exactly to the regular operation capture in the original reuse contract model. Its description is the one illustrated in Figure 6.1.

Definition

A **regular operation capture conflict** occurs when M2 is a refinement or coarsening of operation m in p, and M1 is a regular participant refinement of any of the following forms:

- adding in operation n of class q an invocation to operation m in an acquaintance of type p
- adding in an operation of any participant a cascaded invocation referencing m that binds to p

Super operation capture

This is the case where the first participant refinement is a super refinement. An example of this is depicted in Figure 6.2

Figure 6.2

In the above example class Point is modelled with two methods: display and logFile. Method display shows the point on the screen and method logFile writes to a log file the operations performed on a point. The subclass ColourPoint has a redefinition of method display because it needs to display a colour in the screen. Display contains an invocation to logFile for logging every time the colour is displayed in the screen, this is a self invocation since ColourPoint inherits logFile from Point. One evolver changes the displaying protocol in ColourPoint so that it can reuse the display method in class Point. This is represented by operator M1, which adds a super invocation in operation display of ColourPoint. Another evolver independently modifying the same collaboration contract decides to log every time a point is displayed on the screen also. This is represented by operator M2 which performs a refinement of display adding an invocation to logFile. The result of combining both operators is that every time a ColourPoint is displayed, it is logged twice.

In the above example the undesired effect of the combination of both modifications is the repeating log in the file.

Definition

A **super operation capture conflict** occurs when M2 is a participant refinement or coarsening of operation m in a class p, and M1 is a super participant refinement of any of the following forms:

- adding in a descendant of p, a super invocation to m that binds to m in p
- adding in any participant a super cascaded invocation that referencing m that binds to p.

Self operation capture

This is the case where the first participant refinement is a self refinement. Self operation capture conflicts signal unexpected interactions between independent modifications to the same class hierarchy, due to the late binding of self in self operation invocations. This kind of problems is newly detected in this extension of the reuse contracts model, because the original one did not represent the late binding of self invocations (nor participants hierarchies).

Definition

A **self operation capture conflict** occurs when M1 is a self participant refinement in operation m of class q adding a self invocation to n, and M2 is a refinement or coarsening of operation n in any of the two cases:

a) in a descendant of q that directly inherits m from q

b) in an ancestor of q from which q directly inherits n

Self operation capture conflict also occurs when M1 is a self participant refinement adding in any class a cascaded invocation referencing m that binds to p, and M2 is a participant refinement or coarsening of m in p.

The case a) distinguished in the previous definition can be seen graphically in the schema shown in Figure 6.3.

Figure 6.3

Figure 6.3 shows a class B inheriting operation m from A and defining its own implementation for operation p. The first modifier adds in method m of class A an invocation to p. When called on B, operation n will use the definition of p in B. An unexpected interaction may come up if a parallel modification alters the implementation of operation p in class B (label 'p{…}' is used to denote any refinement or coarsening on operation p, it could even be a super or regular refinement as well). An undesired effect may occur because each invocation of m in B will be affected by the changes performed in p by the second operator. Undesired effects can be repeated actions, as the case of the file logging in the example of Figure 6.2 or any other kind of unexpected dependency.

Combined operation capture

This variant represents conflicts that are a combination of super operation capture variants and a regular (or self) operation capture. It is a kind of transitive closure, see subsection **Scaling up**.

Definition

A **combined operation capture conflict** occurs when M1 is super participant refinement adding a super invocation that binds to m in class q, and M2 is a refinement or coarsening of an operation n that is invoked by m in q.

This represents a kind of indirect operation capture, where the captured operation is not modified, but one of the operations it invokes. A possible schema for a combined operation capture is shown in Figure 6.4, where the second modification affects an operation n of an acquaintance of q that is called from m in q. Here also, label 'n{…}' is used to denote any refinement or coarsening on operation n, it could even be a super or regular refinement as well (the arrow is drawn as a self refinement or coarsening, just to illustrate that a change is performed).

Figure 6.4

Inconsistent operations

Inconsistent operations conflicts are defined in [Lucas97], and are defined in a similar way in our model. Operation capture problems arise because unexpected invocations are added to the specialisation clauses of operations. Inconsistent operations problems are their counterpart, they occur when invocations are unexpectedly removed from the specialisation clause of operations. The same as in operation capture conflicts, there are no structural problems when combining both modifications, but an unexpected interaction of both is obtained. The first evolver performs some changes in the implementation of an operation that he knows is invoked by another operation. But the second evolver removes the invocation to the operation modified by the first modifier. Then the two operations become inconsistent because the changes made to one of them stops affecting the other.

In our extension of the reuse contracts model, as operation invocations can be of any of three different kinds (regular, self or super), there are interesting cases of inconsistent operation conflicts that are difficult to see without a careful observation of how operation invocations actually bind. An example of an inconsistent operation problem is illustrated in Figure 6.5, where the first modifier is a super coarsening.

Figure 6.5

In the example above, the first evolver changes the redefinition of operation move in class ColourPoint, so that now it doesn't reuse the implementation of move in its superclass Point because it doesn't perform a super call anymore. Independently from that, the second evolver wants to log to a file all the movements that all kinds of points make. Because in the base collaboration contract, the implementation of move in ColourPoint uses the one of its parent, it suffices to add an invocation to logFile only in operation move of class Point. In this way all the points will be logged when moved. But the combination of both modifications breaches the assumption of the second evolver, thus yielding unwanted behaviour.

Definition

An **inconsistent operations conflict** occurs when M1 is participant coarsening removing an invocation to operation n of class q, and M2 is a participant refinement or coarsening of operation n in q.

The participant coarsening can be a regular, self or super coarsening and the removed invocation can be a cascaded invocation referencing operation n that binds to q.

The solution to inconsistent operations problems highly depends on the semantics of each of the affected operations and the kind of undesired effect obtained in the combination. In general, changes need to be made in the second operator so that the two operations get consistent again. Further study on conflict resolution needs to be done to be able to give more support for solving this conflict once it is detected.

Unanticipated recursion

This kind of conflict occurs when the specialisation clauses of two independent operations are extended in such a way that when combining both modifications the two operations are mutually recursive [Lucas97].

The independent modifications can both be participant refinements adding an invocation to the operation refined by the other modification. For example, an operation m is refined to invoke n and n is refined to invoke m. But the transitive closure of this is also considered as unanticipated recursion. That is, both refined operations do not necessarily need to invoke one another in order for the combination to result in unexpected recursion. An unexpected recursion problem arises whenever in the collaboration contract that results from combining both operators there is a cycle of operation invocations. The two operators each add one link to form this cycle, and the other links may already exist in the base collaboration contract.

For reasons explained in the previous paragraph it does not suffice to consider only the contract type and elements affected by each operator, but the base collaboration contract also need to be considered to determine that the conflict exists.

In the explanation above the two operators were participant refinements, but they could also be participant extensions adding new operations that take part in the resulting cycle.

Moreover, our model of reuse contracts can detect mutually recursive operations that result from the combination of complex webs of operation invocations that may involve self and super invocations. Due to the late binding of these kinds of invocations, the cycle is not easily detected in the resulting collaboration contract without careful observation of how invocations are actually bound. The addition of two apparently independent operation invocations to two classes in the same hierarchy may lead to unanticipated recursion that is at first sight hidden by the late binding of operations. A possible schema of the combination of two such operations is shown in Figure 6.6.

Figure 6.6

In Figure 6.6 the collaboration contract resulting from the combination of two participant refinement operators is shown. The invocations added by each of them are highlighted with the name. By following the binding of operation invocations it can be seen the an invocation of operation m in class C will result in the recursive invocation of itself.

The definition of unanticipated recursion in [Lucas97] requires the definition of the transitive closure of a specialisation clause:

Definition

The **transitive closure of a specialisation clause** SC is the union of SC and the transitive closure of the specialisation clause of all operations appearing in SC.

For dealing with late bound invocations it is necessary to adapt this definition to be able to deal with the new features modelled in this extension of reuse contracts. Three adjustments to the definition need to be done. First, the specialisation clause of an operation m can contain self operation invocations, thus to calculate the transitive closure of operation m it is necessary to define the transitive closure of operations referred to through self and super invocations. To calculate the transitive closure of a self (or super) operation invocation to n it is necessary to know which is the actual receiver of the operation m. This actual receiver can be the class defining the operation m or a descendant that inherits it. In the latter case, the self invocation will bind to redefined versions (if any) of operation n.

The second point to consider when calculating the transitive closure of specialisation clauses in our model are operation invocations with type casts. In these invocations, the specialisation clause of the invoked method in the target type of the cast should be considered.

The third point to consider in the definition of transitive closure of a specialisation clause are cascaded invocations. Every operation referenced in the cascaded should be counted. The operation invocations that a cascaded invocation add are the transitive closures of every operation referenced in the cascade.

To define the transitive closure of specialisation clauses in our model it suffices to use the original definition in [Lucas97], and to manage type cast invocations and the maintaining of the actual receiver in each calculation as explained above. We won't include here an implementation for this procedure but, due to the simplicity of the model, it can be easily expressed in the function domains formalism we define.

Definition

An **unanticipated recursion conflict** occurs when M1 is participant refinement or participant extension of operation m in class p, and M2 is a participant refinement or participant extension of operation n in class p, and the following conditions hold:

- in the collaboration contract resulting from the combination of both modifications, an invocation to operation m in p appears in the transitive closure of the specialisation clause of n in q
- in the collaboration contract resulting from the combination of both modifications, an invocation to operation n in q appears in the transitive closure of the specialisation clause of m in p
- at least one of the two previous conditions was not true in the collaboration contract resulting from the single application of M1, or the single application of M2

Hierarchy conflicts

This category groups conflicts that are newly detected by our extended model of reuse contracts. All the conflicts in this category involve the new features of our extended model thus they are not variants of conflicts in the original definition of reuse contracts.

This category represents conflicts signalling problems in the class and interface hierarchies. Some of them are applicability conflicts and other are evolution conflicts. For evolution conflicts a more detailed explanation will be given since they have more relevance for this work. Applicability conflicts will only be described briefly.

The first three conflicts represent problems in the class and interface hierarchies that are caused by modifications in the elements of the participants taking part in the hierarchies. In classes there are two kinds of items that are inherited, acquaintances and methods. The first two conflicts deal with problems concerning each of them. Interface participants have only one kind of attributes: operations. The third conflict refers to inheritance of methods in interface hierarchies.

The next two conflicts detect problems when cycles are unexpectedly introduced in class or interface hierarchies, and when there is a double inheritance reachability (a 'diamond') in the interface inheritance.

The last two conflicts are evolution conflicts that signal problems appearing in hierarchies concerning the signature and calling structure of the involved participants. These Masked operation an unanticipated redirection conflicts represent a significant contribution because the detect new evolution problems that cannot be checked by existing checking tools nor the original definition of collaboration contracts.

Invalid acquaintance redefinition conflict

This kind of conflict is defined to signal when the combination of two independent modifications yields an ill-formed class hierarchy. In particular invalid acquaintance redefinition conflicts occur when an instance variable is defined twice along a class hierarchy.

Definition

An **invalid acquaintance redefinition conflict** occurs when M1 is a class inheritance refinement adding a superclass link from class c to class c2, and M2 is an acquaintance refinement in any of the following forms:

- adding an acquaintance to c2 or an ancestor of c2, and the same acquaintance is also defined in c or a descendant of c
- adding an acquaintance to c or a descendant of c, and the same acquaintance is also defined in c2 or an ancestor of c2

Inheritance acquaintance conflict can be solved by renaming one of the two acquaintances, if they were introduced for different purposes, or by removing one of them if they were defined for exactly the same purpose.

It should be noted that in Java the redefinition of instance variables is allowed, the mechanism is called shadowed variables [GJS96], and was described in chapter 3.

Inconsistent return-type conflict

This kind of conflicts comes up when the combination of two independent modifications yields a well-formedness problem in an inheritance hierarchy. In particular, inconsistent return-type conflicts signal incompatibility in the signatures of an participant and an ancestor. Incompatible signatures are those defining an operation with the same name and different return type. Overloading of operations is not allowed in our model of collaboration contract so this implies an ill-formed collaboration contract. This is an initial restriction that we take in our model in order not to introduce too much complexity in this first extension. Further extension can be defined that relax this restriction. Meanwhile the restriction is present, and

inconsistent return-type conflicts detect situations where the merging of independent modifications lead to a collaboration contract that does not respect it.

The first evolver links two participants in inheritance relation, they may be two classes or two interfaces. The second evolver adds an operation in a participant, such that when combining both modifications the added operation is inherited by a participant that also defines the operation but with a different return type. The second evolver could also add the operation in a participant that in the resulting combination inherits it with a different return type.

Definition

An **inconsistent return-type conflict** occurs when M1 is a (class or interface) participant inheritance refinement adding an inheritance link from p to p2, and M2 is a participant (resp. class or interface) extension adding an operation m of type t in one of the two forms:

- in participant p or a descendant of p, and operation m is understood by participant p2 with return type t2 different from t
- in participant p2 or an ancestor of p2, and operation m is defined by participant p or a descendant of p with return type t2 different from t

The solution to this conflict depends on the reason why the operations are named the same and have a different return type. If the purpose of both existent and added operations is really different and there was just a coincidence in the naming, then the added (or the existent) operation should be renamed. If they have common purposes, then maybe one can be introduced as a redefinition of the other, then the return types of both operation should be set to the same. If this is impossible, then rename one or both operations. If both operations have exactly the same purpose, and they could have the same return type as well, then one of them should be removed.

Cyclic inheritance hierarchy conflict

This kind of conflict detects structural problems that may appear in the combination of two independent modifications to the class hierarchy or two modifications to the interface hierarchy of a collaboration contract. In particular cyclic inheritance hierarchy conflicts signal unanticipated cycles in the class hierarchy and in the interface hierarchy.

The conflict arises when both evolvers perform addition of superclass (superinterface) links in such a way that in the combination of both, one class (interface) is a proper ancestor of itself.

Definition

A **cyclic inheritance hierarchy conflict** occurs when M1 is a class (interface) hierarchy refinement adding a superclass (superinterface) link, and M2 is a class (interface) hierarchy refinement adding another superclass (superinterface) link, such that in the collaboration contract that results from applying the two operators one after the other there is a class (interface) that is itself included in its set of ancestors.

It should be noted that the rules for detecting cyclic hierarchies requires to examine both operators and also the resulting collaboration contract.

The possible solutions for cyclic inheritance hierarchy conflict range from the removal of a superclass (superinterface) link to the complete redesign of the class (interface) hierarchy, depending on the situation.

Double inherited interface conflict

The multiple inheritance structure of interface hierarchies allows the occurrence of strange patterns in the interface inheritance graph. This kinds of conflict are raised when the combination of two independent

modifications yields a collaboration contract containing an interface for which there are two or more possible 'ancestors paths' to the same interface.

These are evolution conflicts since a double reachability interface inheritance graph is allowed in languages like Java and consequently in our collaboration contracts model. The purpose of these conflicts is to give a warning of possible errors in the design of the inheritance hierarchy. It maybe the case that the 'diamond' pattern in the interface hierarchy is a desired effect, in this case there is no real problem and the conflict should be ignored.

The conflict arises when both evolvers perform addition of superinterface links in such a way that the in combination of both, one interface has more than one path to the same ancestor.

Definition

A **double inherited interface conflict** occurs when M1 and M2 are both interface inheritance refinements such that, in the collaboration contract that results from applying the two operators one after the other, there is an interface with more than one inheritance path to the same ancestor.

The same as for the previous conflict the rules for detecting double inherited interface requires to examine both operators and also the resulting collaboration contract.

The possible solutions for these conflicts range from the removal of one or more superinterface links to the complete redesign of the interface hierarchy, depending on the situation.

Masked operation conflict

This kind of conflict signals problems that arise when the combination of two independent modifications yields an unexpected redefinition of an operation.

Masked operation conflicts are evolution conflicts that appear, for example, when two evolvers independently add an operation with the same signature to two different classes in a hierarchy. If the two class extensions are combined, one of the added operations will redefine the other, and this effect was not foreseen by any of the two evolvers.

It may be the case that both added operations have the same or different behaviour. If they have different behaviour then there may be problems if the evolver adding the operation in the higher class in the hierarchy expects that all descendant classes inherit it. When combined with the addition of an operation with the same signature in a descendant class c, performed by the other evolver, all the descendants of c will have a masking of the operation added by the first evolver (the mask is the operation added by the second evolver). The conflict is determined because there is a breach in the assumptions of the first evolver that all the descendant of c would inherit the operation (and implementation) he/she added to c.

If the two added operations have the same behaviour (they have an equivalent implementation), then the unexpected redefinition does not have a serious behavioural problem. But even in this case, replicated code is introduced in the hierarchy. This implies the addition of unexpected dependencies between the two operations, since they need to be maintained consistent with one another. And this is a problem that may be better to avoid.

An illustrating situation is shown in Figure 6.7.

ExampleContract
Point
Int shiftX
Int shiftY
Bool log
move
{log}
x
y
r
θ
Float
PolarPoint
Float shiftθ
Float shiftR
ColourPoint
Colour
c
Void setColour
MyPoint
Void any
n
{move}
M1
M2
ExampleContract1
Point
Int shiftX
Int shiftY
Bool log
Bool move
move
{log}
x
y
r
θ
Float
PolarPoint
Float shiftθ
Float shiftR
Bool move
{super, shiftX,
shiftY, shiftθ, shiftR }
ColourPoint
Colour
c
Void setColour
MyPoint
Void any
n
{move}
ExampleContract2
Point
Int shiftX
Int shiftY
Bool log
Bool move
move
{log}
x
y
r
θ
Float
PolarPoint
Float shiftθ
Float shiftR
ColourPoint
Colour
c
Void setColour
Bool move
{super, shiftX,
shiftY, setColour}
MyPoint
Void any
n
{move}

Figure 6.7

The example above illustrates a hierarchy that results from the implementation in a single inheritance programming language like Java of a design modelled with multiple inheritance. In the original design class MyPoint has PolarPoint and ColorPoint as direct superclasses, and both are subclasses of Point, which represent a model for a basic cartesian point. To implement that hierarchy with single inheritance it had to be linearised as is shown in the base collaboration contract. Operator M1 introduces operation move in class PolarPoint that, given an offset, adds the corresponding values to the polar and cartesian co-ordinates to shift the point on the screen. So the operation added by M1 performs a super call to move in class Point (which does not implement any particular behaviour except logging the move), and shifts its polar and cartesian co-ordinates. Simultaneously, operator M2 adds the same operation move to class ColourPoint, which performs a super call to move in class Point and then shifts its cartesian co-ordinates and sets the colour according to the new position on the screen. When both modifications are merged, it happens that operation move in PolarPoint gets masked by the definition of move in ColourPoint. Moreover, in this case the super invocation performed by move in class ColourPoint gets captured by the redefinition in PolarPoint and therefore the shift of the co-ordinates is performed twice: once by move in ColourPoint and once by move in PolarPoint. In this way the invocation of move in class MyPoint gets a wrong behaviour, because it moves twice the offset it is expected to move.

In this particular case, as the two added operation are themselves redefinitions of an existing operation (move, in class Point) then there already exist invocations to the operation that get the masked effect. If the two added operations were introduced for first time in the hierarchy, then the masking conflict is more potential, since the base collaboration contract does not contain invocation to that operation on a participant in the affected hierarchy because the operation did not exist before. But even in the case when the added operation is new in the hierarchy, one or both of the operators could add at the same time new operations that invoke the masked operation. In the example above, it could have been the case that operation move was not implemented in Point, and the invocation in class MyPoint was added by any of the two operators (together with the invoking operation n).

Definition

A **masked operation conflict** occurs when M1 is a participant extension adding an operation m to a participant p, and M2 is any of the following:

- a participant extension adding the same operation to a descendant of p
- a context refinement adding a hierarchy link to p or an ancestor (descendant) of p that causes p to have an ancestor (descendant) that also defines operation m

Observe that masked operation conflicts can arise in class hierarchies as well as interface hierarchies.

The possible solutions to masked operation conflicts depend on the situation. If the two operations are added for the same purpose and have the same behaviour then one of the two could be removed. If they have different purposes then they could be merged, if their behaviours are compatible then they could be adapted so that one can reuse the other through a super call. If they perform two different purposes, a possible solution can also be to rename one of the two operations.

Unanticipated redirection conflict

This kind of conflict also signals problems concerning redefining operations that arise in participant hierarchies. It is the counterpart of masked operation conflicts. Those conflicts appear when adding to a class an operation that masks the definition of another operation in an ancestor. Unanticipated redirection conflicts signal problems that arise when the removal of an operation in a participant causes invocations to that operation to be redirected to the definition of the operation in an ancestor .

Unanticipated redirection conflicts are evolution conflicts that arise when one evolver removes a redefining operation in a participant and the other evolver simultaneously adds an invocation that binds to the removed redefining operation. The second refinement could be a regular, self or super refinement. It may be the case that the second evolver performed an invocation to that redefinition of the operation because some behaviour it implemented was needed that is not provided by ancestors' versions of the same operation. The independent removal of the redefining operation causes the invocation to loose that expected behaviour.

A situation where there is an unexpected redirection conflict is shown in Figure 6.8. This is a symbolic example where a line of evolution (one or more steps of evolution) on a hierarchy conflicts with simultaneous uses of the same hierarchy. The hierarchy consists of a general FileManager class that serves as an abstract specification for subclasses, that defines a read operation that does not perform any particular action. The subclass of FileManager is MyFileManager, which provides an accessFile operation which is called from read in this class. The first line of evolution (L1) aims at improving the efficiency of the read service in MyFileManager, therefore operation read is replaced by operation fastRead that uses a cache to provide faster accesses. There are usually reasons for giving a new name to a new version of an operation in a class, for example that different parameters are required by the new version. At the same time, in another line of change (L2), an application uses the read services of the file manager by invoking operation read in class MyFileManager. When the two lines of change are combined, the invocation to read added by L2 is unexpectedly redirected to operation read in FileManager, therefore loosing the efficiency improvements introduced by L2. Worse than that, if operation read in FileManager does not implement the complete actions of a read service (because it is defined to be redefined in subclasses), then the added invocation gets an incorrect behaviour.

Even when the example shows lines of evolution instead of single steps (as in previous examples) the essential problem that appears in the merging of the two modifications can be appreciated anyway. Considering lines of changes with an underlying purpose for it all, provides more context to understand single changes that could appear meaningless when looking at them isolated.

Figure 6.8

Definition

An **unanticipated redirection conflict** occurs when M1 is a participant cancellation removing an operation m from a participant p, where m is a redefinition of m in an ancestor of p, and M2 is participant refinement that adds an invocation referencing m that binds to m in p:

Observe that unanticipated redirection conflicts can arise in class hierarchies as well as interface hierarchies. If it is a class hierarchy, then the second operator can be any kind of refinement:

- a regular refinement invoking m on an acquaintance of type p (or of type q, where q is a descendant of p that directly inherits m from p)
- a self refinement in p (or in a descendant that directly inherits m from p)
- a super refinement adding in a descendant of p a super invocation that binds to m in p.

If instead, the hierarchy is an interface hierarchy then the second operator can only be a regular refinement as in the first case.

The possible solutions to unanticipated redirection conflicts depend on the situation. If the removed redefining operation did not perform any action different from those in its ancestor's versions (at least for the needs of the invocation added by the second evolver), then there is no problem and it can be removed. Otherwise it depends on the reasons of the removal, if the behaviour is now implemented by another operation, then this operation should be invoked by the refinement operator instead of the removed one. If

the behaviour was not relocated and it is needed by the added invocation, then the operation cannot be removed.

Implementation conflicts

The same as the previous category, this one also contains conflicts that are newly detected by our extended model of reuse contracts. It groups all the conflicts signalling problems related to the implementation relation of our collaboration contracts.

The first presented conflicts are applicability conflicts that signal problems in the well-formedness of the resulting implementation relation.

The final two conflicts are evolution conflicts that are defined to help in the detection of redundancy in behaviour of classes inside a hierarchy.

Missing implementation conflict

These conflicts occur when independent modifications are made to a base collaboration contract, that when combined yield an ill-formed collaboration contract. In particular, the problem of ill-formedness in missing implementation conflict is that an implementation between a class and an interface is not valid because the class does not understand all the operations specified by the interface. Thus, missing implementation conflicts are applicability conflicts.

The problem occurs when one evolver adds an implementation link between a class and an interface in a collaboration contract, and the other evolver in parallel performs some modification in the operations understood by the class or the interface, that invalidates the implementation relation added by the first evolver. Thus, the first operator in a missing implementation conflict is a context refinement adding an implementation link between a class c and interface i. The second operator can be of any of the following four forms: (1) a class participant cancellation removing in c an operation needed for the implementation link to be valid; (2) a class inheritance coarsening removing a superclass link that causes class c to no longer inherit an operation needed for the implementation link; (3) an interface extension adding a new operation to i or an ancestor of i that is not understood by class c; and (4) an interface inheritance refinement adding a superinterface link that causes i to understand a new operation that is not understood by class c.

Definition

A **missing implementation conflict** occurs when M1 is context refinement adding an implementation link from class c to interface i, and M2 is any of the following:

- a participant cancellation removing from class c an operation m that is understood by i
- a class inheritance coarsening removing a superclass link in c or an ancestor of c, such that there exists an operation m (understood by i) that class c inherited directly from one of the removed ancestor classes
- an interface extension adding an operation m to i or to an ancestor of i, and m is not understood by class c
- an interface inheritance refinement adding a superinterface link to i or an ancestor of i, such that there exists an operation m (not understood by c) that i inherits directly from one of the new ancestors.

The solutions to missing operation conflicts depends on the type of the second operator. The two possible actions to take are add the ‘not implemented’ operation m to class c or an ancestor of c, or remove it from interface i or the ancestor from which it inherits m.

Redundant specification conflict

The problems detected by this and the next kind of conflicts are very naive, but they may be indicating more serious problems that remain undetected otherwise. The concrete problem detected is in both kinds a redundant implementation link.

In redundant specification conflict, two modifications are independently introduced that, when combined, cause a class participant to have twice the same interface as specification. For this to happen, one of the implementation links is defined by the class and the other is 'inherited' from an ancestor class. In the combined collaboration contract there will be redundant information about the specification of a class. This double specification may be innocuous. But we decide to define this (and redundant implementation) as evolution conflicts because they may help in the detection of other more serious problems of redundancy in structure or behaviour. Due to the frequent use that object-oriented programs make of polymorphism there is a potential risk, when parallelly evolving a single component, that classes (or operations) performing the same behaviour are defined twice. These redundant operations can be taken for polymorphic operations and the duplication remains undetected. It would be completely awkward to signal as conflicts every time two operations with the same name are encountered. But the specification interfaces that a class declares to implement express somehow an intention for some of the operations of the class. This information can be used as a means of detecting duplication in the behaviour of classes. This is the purpose of redundant specification and redundant implementation conflicts.

In redundant specification conflicts the problem comes about when one operator is an implementation relation refinement adding a link between a class c and an interface i, and the second operator is also an implementation refinement adding i as specification of a class that is an ancestor of c. The second operator could also be a class inheritance refinement adding a superclass link to c or an ancestor of c that causes the 'inheritance' of i as a specification interface again.

Definition

A **redundant specification conflict** occurs when M1 is a context refinement adding an implementation link from class c to interface i, and M2 is any of the following:

- a context refinement adding an implementation link from an ancestor class of c to i
- a class inheritance refinement adding a superclass link to c or an ancestor of c such that one of the new ancestors of c also defines i as a specification interface.

The solution to redundant specification problems depends on the real problem that they are detecting. If there is a redundancy problem in the behaviour of different classes then refactoring changes or other kind of design modifications should be performed to correct them [OJ93]. If there is no hidden problem underneath the double specification relation, then nothing needs to be done.

Redundant implementation conflict

The basis of redundant implementation conflict problems were already described in the definition of redundant specification conflicts. The difference here is that the double implementation link is between a class c and two interfaces i and i2 (i2 ancestor of i). The link specifying the implementation of interface i2 is not needed since it brings no new information, because all the operations specified by i2 are included in i. The same as before this can represent no problem at all or it can be the consequence of deeper redundancy in the system.

With redundant implementation conflicts the problem arises when one operator is an implementation relation refinement adding a link between a class c and an interface i, and the second operator is also an implementation refinement adding an ancestor i2 of i as specification of class c. Another combination of operations that can lead to this problem is if the first modifier adds to class c an implementation link to i, and the second operator is an interface inheritance refinement adding a superinterface link to i or an ancestor of i that makes an interface i2 (already related to c by an implementation link) an ancestor or descendant of i.

Definition

A **redundant implementation conflict** occurs when M1 and M2 are any of the following combinations:

1 :

- M1 is a context refinement adding an implementation link from class c to interface i
- M2 is a context refinement adding an implementation link from class c to interface i2 ancestor or descendant of i

2 :

- M1 is a context refinement adding an implementation link from class c to interface i
- M2 is an interface inheritance refinement adding a superinterface link to i or an ancestor of i such that there exists an interface i2 that becomes ancestor of i, and i2 is linked to class c in the implementation relation

The same as for redundant specification conflicts, the solution to redundant implementation problems depends on the real problem they are detecting. If there is a redundancy problem in the behaviour of different classes then refactoring changes or other kind of design modifications should be performed to correct them. If there is no hidden problem under the double specification relation, then nothing needs to be done.

Type casts conflicts

The same as the previous category, this one also contains conflicts that are newly detected by our extended model of reuse contracts. It is intended to group all conflicts dealing with type casts in statically-typed programming languages. Problems may arise when structural relations among participants evolve in such a way that certain type casts become invalid.

We devoted our analysis of type casts restricted to the type system provided by Java [GJS96]. The problems we detect are a consequence of the checking rules provided by Java. We identify only one such kind of conflicts described below, that we call invalid type cast conflict.

As our work was more devoted to analyse the added expressivity given by the introduction of different kinds of participants, participant hierarchies and late binding, a further analysis of type casts and problems that they may have is intended as part of immediate future work.

Invalid type cast conflict

These conflicts arise when an invocation containing a type cast is added and the source and target types of the cast do not have an appropriate relationship between them. As explained in chapter 3, the Java type checker does not force casts from classes to interfaces or vice versa to respect any relationship between the source and target type (in none of them is *final*). Our model of collaboration contract is not more restrictive than is the Java compiler and thus these restrictions are also not enforced in our model. Invalid type cast conflicts are evolution conflicts, thus they don't involve structural ill-formedness in the combination of parallel modifications.

If a class type is cast to an interface (or vice versa) such that there are no common operations in their signatures then, even when there will be no compile time errors, problems will appear at run time when a method is invoked on a class instance that does not understand it. The invalid type casts conflict aims at detecting these kinds of problems.

Definition

An **invalid type cast conflict** occurs when M1 is a participant refinement adding an invocation that contains a type cast from a class c to an interface i (from an interface i to a class c), and M2 is any operator such that after applying it on the base collaboration contract, in the resulting collaboration contract class c does not implement interface i.

It should be noted that the rules to detect an invalid type cast conflict are slightly different than those describe thus far, because they require not only examining the involved operators but also the resulting collaboration contract resulting from the application of on of them.

All the described are some conflicts that we consider most relevant to be detectable with the model we propose. But the set of conflicts that we present is by no means complete. It is likely that there are more conflicts that we didn't consider and that can be added to our classification of evolution conflicts.

Scaling up

An interesting capability of the conflicts we identified in this chapter is that they can detect inconsistencies introduced when using cascaded invocations. The possibility to express cascaded invocations is very important from a practical point of view, since they are very frequently used in usual source code. But cascaded invocations also have strong implications in our model from a more theoretical point of view. They provide a way to directly reference elements in the transitive closure of the dependencies of a participant.

In the original definition of reuse contracts, the direct dependencies of a participant are determined by its acquaintances. The transitive closure of the dependencies of a participant in that model can be defined by recursively taking the dependencies of its acquaintances.

By including method return types in the extended model of collaboration contract, new dependencies between participants can be expressed. A participant p depends on the participants specified as return types of every method it invokes, in the sense that any change to those participants may affect p. We will call these dependencies, *method-result dependencies*. We could define the transitive closure of the method-result dependencies by recursively taking the method- result dependencies of the method-result dependencies of a participant. The transitive closure of the dependencies of a participant p represents the set of all the participants that will have a potential impact on p if they are modified.

The transitive closure of the dependencies of a participant is an important notion to consider in the detection of conflicts. Changes introduced by one evolver in a participant p may breach the assumptions made by a second evolver over another, apparently unrelated, participant q. The changes introduced in p can affect q, if p is in the transitive closure of the dependants of p. A discussion of these kinds of conflicts, called *transitive conflicts* can be found in [Mens99a], where unanticipated recursion conflict is identified as a transitive closure (since the transitive closure of a specialisation clause must be considered for its detection). In this work we do not focus on the identification of new transitive conflicts. Only the combined operation capture can be considered as a kind of transitive conflict, because it detects problems that may appear when modifying a method in an indirect dependency of a participant.

By introducing cascaded invocations in the model, we allow the direct reference of indirect dependencies of a participant. A participant refinement adding in p a cascaded invocation, introduces new participants in the method-result dependencies of p: all the participants that are method return types of each method included in the cascaded invocation are brought to the set of direct method-result dependencies of p. As all the conflicts involving participant refinements or coarsenings that we define in this chapter contemplate the case where the added (or removed) invocation is cascaded, they could be classified as transitive conflicts. But this is not completely correct, because they don’t consider all kinds of indirect dependencies. The case where an indirect dependency between two participants is determined by successive simple invocations instead of being determined directly by a cascaded invocation, is not consider by the conflicts we defined.

Chapter 8 - Conclusion

Contribution

In this dissertation we defined a possible extension to the reuse contracts model of [Lucas97] to make it applicable to source code.

First we studied the reuse contracts model carefully, analysing the supported features. The model allows structured documentation of reusable components and is based on the definition of *collaboration contracts* to describe reusable evolving components, and *modification operators* that document in a precise way all the possible changes performed on the collaboration contracts. This documentation is used to detect conflicts when the same component is affected by simultaneous independent modifications.

We took a representative example of 'good-quality source code' and studied how the reuse contracts model could be applied to it. We chose the Java version of HotDraw code [HotDraw] because it is a framework that has been discussed frequently in literature, and its structure is stable and well-factored. We found some concrete problems to apply the definition of reuse contracts as it is in [Lucas97] to document collaborations in the source code.

The first visible problem was the restrictions to represent method invocations. Reuse contracts enforce the Law of Demeter which rules out cascaded method invocations, while in real source code they are frequently used. This is a serious restriction that cannot be solved with the elements provided in the original definition of reuse contracts. Also, reuse contracts do not provide any support for type information, therefore many features heavily employed in programs written in statically typed languages like Java cannot be represented in the model. Some of these are method return types, interface types and type casts.

Another important feature that is not supported in the original definition of reuse contracts are inheritance hierarchies, such as class and interface hierarchies in Java programs. The defined collaboration contracts do not represent the notion of inheritance between participants. Consequently, late binding of operation invocations cannot be modelled in the original definition of reuse contracts.

We defined a conservative extension of reuse contracts that includes modelling elements that tackle the above mentioned restrictions. We formulated our extension as a 'delta' with respect to the existing model. In this way all the representable features and detectable conflicts in the original definition are still supported in our extension.

Because Java is one of the most broadly used typed object-oriented programming languages, we took its type system the as specification for the semantics of the new elements that need to be added to reuse contracts in order to make them applicable to source code.

First we extended the model of collaboration contracts to include implementation-level features. The first structural extension to the collaboration contracts model is the definition of different kinds of participants: we represent class participants and interface participants. For each of these kinds of participants we introduced inheritance hierarchies. This allowed to express inheritance of operations and acquaintances as it is provided by programming languages like Java. Different kinds of relationships are presented in our model for collaboration contracts:

- acquaintance relations
- class inheritance relations
- interface inheritance relations
- implementation relations, which associates class participants to interface participants in the same way as they are linked in Java by the use of the 'implements' keyword in the declaration of classes.

The definition of participants is also extended. In order to be able to deal with method result types, we defined method signatures that specify the type of the returned value. The results of method invocations are used to further invoke methods on them. In this way cascaded method invocations are defined.

The structure of operation invocation is extended in three different ways:

- by allowing cascaded invocations
- by allowing invocations on self and super
- by allowing type casts

Thanks to the representation of participant inheritance hierarchies, it is possible to represent self and super invocations. Late binding of self is an important semantic feature introduced in our model. Finally, type cast is a mechanism heavily employed in current-day statically-typed languages and consequently needs to be represented in order to document collaborations at source code level. Our modelisation of type casts follows the specification of the Java type system.

We defined a formal model of the collaboration contracts using function domains. The use of a mathematical model helps in rigorously defining the new modelling elements, and helps in clearly understanding their semantics, specially their interaction with other existent elements.

After defining the extended model for collaboration contracts, we analysed the possible modification operators that could be applied on them. We defined a set of reuser clauses that allow the documentation of changes to the elements represented in our extended collaboration contracts. We followed the organisation used in [Lucas97] for operators, distinguishing new variants for some of them to deal with the new features in our collaboration contracts.

We also defined formally the set of modification operators for collaboration contracts.

Once the model of extended collaboration contracts and the operators to document modifications to them were defined, we studied the evolution conflicts that could be detected. First we analysed how the conflicts detected in the original model can still be signalled in our extension. Most evolution conflicts are detectable just by considering the two modification operators involved. As we define variants on those operators, we also have variants of the conflicts detected in [Lucas97]. Some of these variants are interesting because they are conflicts detected with the original definition of reuse contracts but where the new features we introduce are also involved. An attractive contribution in that sense is that we integrated the detection of inconsistencies generated with the indirect dependencies introduced by cascaded operation invocations. But most importantly, we identified new conflicts that were not specified in the original definition of reuse contracts. The new conflicts aid in detecting important problems with the evolution of inheritance structures at source code level. The possibility to identify new problems in evolution shows that the extension of the original model we have made has a meaningful added expressiveness.

Evaluation

Reuse contracts can be used to provide a structured documentation of the evolution of reusable components. One of the most remarkable advantages is the simplicity of the model, which makes reuse contracts intuitive to use by developers thus promoting disciplined reuse and evolution. But until now they have mainly shown their use in helping to manage evolving designs [Mens99b] (and requirements [D'Hondt98]). In this work we have demonstrated that extensions are possible to adapt them to manage evolution of implementation level components, while conserving the simplicity and intuitiveness of the approach. We provide a clean

extension of reuse contracts that continues to help with the same evolution problems identified in [Lucas97], while including the detection of new conflicts that may arise when evolving source code level components. In particular we focused on the reuse and evolution of inheritance hierarchies as implemented in Java-like programming languages.

With the extended model we built, all the detectable conflicts can help the software developer to write more reusable programs if the reuse contracts are implemented in a tool.

One of the reasons that make reuse contracts a simple approach, easy to implement in tools is that they only manage static information. This makes their implementation more straightforward than for approaches that require extensive data flow analysis, or proof checking such as deadlock detection. The model we defined deals with dynamic features of programs such as late binding of self and method result values. But we introduced these features adding the minimal indispensable extensions so that the complexity of the approach remains low and its implementation straightforward. The formal definition we provided for the extended model shows this simplicity, and serves as a precise specification for tools implementing it.

Over these basis, the general structure of case tools for the extended reuse contracts would be the same as the one discussed in [Lucas97]. A *collaboration contract extractor* would assist in the extraction of collaboration contract information out of statically typed source code that needs to be documented for reuse and evolution. The tool would need assistance of the user to document only those branches of a hierarchy that are relevant for a given collaboration. The work in [DeHondt98] includes the implementation of such a tool. When a documented component documented in that way is reused or is evolved into a new version, the modifications performed on the documented parts of the code should also be documented using reuser clauses. In this way, every time that a component is modified, conflicts with previous or simultaneous modifications can be checked. A *conflict checker* module would work over the generated documentation and would use the simple rules described in chapter 6 for conflict detection.

Related work

There are a few approaches that tackle the problem of reuse and evolution of software at the implementation level. In this chapter we briefly describe some of them and establish some relations with the work we present in this dissertation.

Mira Mezini in [Mezini97] aims at support managing problems in the evolution of class libraries at the language level. The essential view of how reuse and evolution problems should be tackled is similar to that in reuse contracts: making explicit the assumptions of the provider of a reusable component that should be respected upon modifications. The approach of Mezini uses metalevel techniques to support the internal protocols between providers and reusers.

Reuse and evolution are defined as vertical and horizontal evolution respectively. As these modifications are studied at the level of class libraries, the affected components are classes. Hence, vertical evolution takes place when an inheritor of a class reuses the definition of the base class by defining a subclass of it, and horizontal evolution occurs when the original (base) class is exchanged by a new version that introduces changes. The work is based on the fact that both kinds of evolution can be homogeneously treated as composition. A *smart composition* model is defined that takes into account the properties of the strong coupling between the object-oriented modules to be composed, as opposed to the *trivial composition* that occurs when combining procedural modules.

Co-operation contracts are defined that document the properties of the base class that should be propagated to inheritors. Co-operation contracts describe properties of the class, its methods and attributes, such as ‘dependent’ (methods that a method must invoke), ‘non-override’ (statement that a method cannot be overridden), ‘required’ (statement that a method must be implemented concrete inheritors), ‘abstract’ (statement that a class cannot be instantiated), etc. The smart composition model ensures the propagation of properties specified by co-operation contracts by automatic customisation at the inheritor’s site.

Support for horizontal evolution in the model of Mezini includes the automatic extraction of information about the calling structure of a class, to track the differences with new version of the base class. Modification operators of reuse contracts as defined in [SLMD96] are used in this step.

The model supports composition of an existing inheritor of a base class with a new version of the class. Two kinds of conflicts are detected in this composition: inconsistent operations and operation capture. The inheritor is informed about the invalidation of changes he defined and automatic transformations are proposed to avoid the inconsistencies, like renaming of a captured operation.

An implementation of the smart composition model in Smalltalk-80 is proposed that uses the metalevel features available in the language. The metalevel approach to the composition model allows the customisation of the model in a direct way.

The work of Mezini is relevant for reuse contracts from a practical point of view, since the composition procedure shows the feasibility of conflict detection implementation at the programming language level, with the interesting contribution of the semi-automatic resolution of conflicts. Our work concentrates more on a conceptual view of the problem. We study reuse and evolution at language level of complete class hierarchies, widening the focus from a class granularity to class hierarchy granularity. Moreover, our work is based on typed languages like Java, where many of the properties expressed in the co-operation contracts (like 'required', 'non-override' and 'abstract') are already provided by the language.

Another work in evolution at the programming languages level is [Cornelis97], that aims at providing an implementation for reuse contracts in Java. The feature analysed is visibility modifiers of methods as defined in Java.

Because Java field modifiers limit the visibility of modifiers, they influence reuse contracts. Problems are detected when trying to apply reuse contracts as defined in [SLMD96] to Java classes. Hence single-class reuse contracts notation is extended to allow visibility modifiers (**private**, **public**, **protected**, **private protected** and **default**) in the method descriptions of a class. Modifier **final** is also introduced.

The modification operators (called *reuse modifiers* in the initial formulation of reuse contracts) are not redefined in this work. The impact of the visibility modifiers on the evolution conflicts defined in [SLMD96] is analysed. Only operation capture and inconsistent operations conflicts are affected by the **private** visibility modifier. The remaining visibility modifiers and **final** are do not affect the generation of inconsistencies when independent modifications are performed to the same class.

A good part of the work is devoted to study the Java compiler and the Java virtual machine to introduce reuse contracts as a module in the language. The Java compiler is extended to support reuse contracts as a kind of Java interfaces extended to support specialisation clause for methods. The Java implementation relation between classes and interfaces is used as link between a class and its documenting reuse contract. The implementation is validated by applying it to the Component hierarchy of Java AWT framework. The hierarchy is documented as a base root class (Component) with subsequent reusers (the subclasses). In this way the design of the AWT hierarchy is made explicit and some flaws are identified. The reuse contracts documentation aid in hinting a possible redesign.

This approach to the implementation of reuse contracts in typed languages is in much complementary with the work we present in this dissertation. [Cornelis97] has a mainly practical point of view of the problem, aiming at building reuse contracts as a tool integrated in the Java language. Our approach to the documentation of statically typed source code introduces more structural extensions to the reuse contracts model aiming at handling cascaded method invocations, and inheritance hierarchies. Since we did not go deep in the problem of tool implementation, nor Java specifics as visibility modifiers, the integration of both approaches appears as a promising work line towards a real integration of reuse contracts at implementation level in development environments.

The two lines of work described above are the most closely related to the approach we present. A common conceptual difference with both of them is that they consider inheritance as a form of reuse, thus the evolution of inheritance hierarchies cannot be documented. We introduce inheritance relationships in the collaboration contract language, thus allowing the documentation of simultaneous changes that affect a class and its subclasses.

The work presented in [KGHWTC94] also deals with modifications performed to class hierarchies at code level. It identifies several kinds of changes that a object-oriented source code may undergo: data changes, method changes, class changes and class libraries changes, and describes further the class libraries changes. The representation of class libraries with graph notation is defined, and the changes that affect it are calculated as the difference between an initial graph and a final graph. The possible changes analysed are addition and removal of classes and class relationships.

The work aims mainly at doing change impact analysis to apply it in regression tests and maintenance tools. In contrast to our work, only one line of modification is analysed therefore there are no conflicts to deal with. Also, the abstraction they have is very low level, and lots of language dependant issues are involved (they use C^{++}).

A worth mentioning field in relation with reuse contracts is merging techniques. Merging techniques are applied by tools that support merging of parallel versions of software artefacts. Reuse contracts can be considered as a sophisticated kind of merge tool that support detection of inconsistencies.

[Mens99a] makes a distinction between two different kinds of merge tools:

- *two-way merge tool*: tools that compare two alternative revisions of the same software artefact and merges them in a single resulting version. To this end, it interactively displays the detected differences to the user who has to select the appropriate alternative. Alternatively, it may also perform an automatic merge, based on some arbitrary decision of which alternative is more appropriate.
- *three-way merging tools*: to reduce the number of decisions that have to be made by the user, a three-way merge tool consults a common base version if a difference is detected. If a change has been applied in only one revision, this change is incorporated automatically. Otherwise, a conflict is detected that can be resolved either manually or automatically. Three-way merging is more powerful than two-way merging because more information is available.

Using this distinction, it is obvious that reuse contracts belong to the three-way merge tools. As explained in chapter 6, the contract types are used to detect evolution conflicts (or merge conflicts) between parallel revisions of the same piece of source code.

Another categorisation based on the semantic level at which merging is performed (and consequently based on the kinds of detectable conflicts) is also defined:

- Textual merging is applied to text files. Almost all commercial software configuration management systems support textual merging [RBI95]. Although we can expect only an arbitrary text file as the result of the merge (instead of a well-formed software artefact) and only physical conflicts can be detected, textual merging seems to yield good results in practice [Leblang94]. In particular, it works well when small local changes to large well-structured programs are combined and changes have been co-ordinated beforehand so that semantic conflicts are unlikely to occur.
- Syntactic merging exploits the context-free (or even context-sensitive) syntax of the versions to be merged. Therefore, it can guarantee a syntactically correct result and can perform more intelligent merge decisions. However, syntactic merging has been realised only in a few research prototypes [Westfechtel91].
- Semantic merging takes the semantics of programs into account [BHR95]. Semantic merge tools perform sophisticated analysis in order to detect conflicts between changes. However, it is a hard problem to come up with a definition of semantic conflict that is neither too strong nor too weak (and is decidable). Furthermore, the merge algorithms developed so far are applicable only to simple programming languages. For these reasons, semantic merge tools have not yet made their way into practice.

Reuse contracts can be categorised in the latter category of semantic merging.

Future work

A large range of possible extensions to the work we present in this dissertation are possible. We identify two kinds of continuations: extensions to the formal model, and other kinds of remaining work more related to the implementation of the presented model.

Formal extensions

There are several extensions that could be made to the formal model we define. Here we only mention some of them, but a further analysis on how they affect the simplicity of the model should be carried out in order to keep reuse contracts an intuitive and usable approach.

More sophisticated type system

The type system we defined for the extended reuse contracts is very simple. It was not our intention to make use of the whole power that can be achieved with type systems, but to add the smallest extension needed to manage return types in cascaded invocations, while maintaining the simplicity of reuse contracts model. Nevertheless, the integration of a more sophisticated type system in the reuse contracts model may result in very useful contributions. For example the representation of genericity as provided by the *templates* mechanism in GJ [BOSW98] would be an interesting line of research continuing this work.

Arguments specification

Arguments of methods provide a feature without direct representation in our model. We proposed to specify them as a special kind of acquaintances of classes, but the direct inclusion of arguments in the signature of methods would probably be a better solution. The introduction of arguments in method invocations implies adding data flow control into the model. Since the existing model of reuse contracts only deals with control flow information, and the combination with data flow information may turn the model too much complex, the way to incorporate arguments passing should be carefully studied. The most significant information added to collaboration contracts with the representation of arguments would be about the actual receiver of a method. Method receivers in a model with arguments specification could be arguments of the method, so more information is handled about the type of method receivers.

Meta architecture information

In the model we present only static instance level information is supported. We represent instance acquaintances and instance methods. It would be interesting to include a mechanism for the representation of class level information as well. The differentiation between classes and instances in this way would allow to represent static methods (such as the *constructors* supported by Java [GJS96]) and static variables, thus providing a more complete representation model for actual source code.

Arity information in acquaintance relationships

The possibility to specify the aritiy of a given acquaintance of a class would yield a model that stays closer to implementation level features. It would provide the possibility to specify when an acquaintance of a given type is multiple, i.e. when several instances of the specified type are associated to the acquaintance name. This extension would require finer specifications in operation invocations. When a method is invoked on a multiple acquaintance, it could be possible to specify if it should be invoked on one, some or all of the associated instances.

Other information

The inclusion in the model of information such as abstract/concrete and private/protected/public modifiers for methods and final modifier for classes is an immediate following step to be taken. Formulations have been defined in [Lucas97], [DeHondt98] and [Cornelis97]. In this work we tried to focus on features that were not completely covered by those works. Now the integration with them is a key step towards a complete reuse contract model that supports most important implementation level features.

Other extensions

Composite operators

In this work we define modification operators that express very simple changes to a collaboration contract. Actual modifications in practise however, are much more complex and may include several steps of the simple changes we define. The definition of higher level modification operators in terms of the simple operators we proposed in this work is an important step to make reuse contracts scalable to large software systems. This topic is discussed in [Mens99a].

Conflicts resolution

Further work needs to be carried out concerning the possible ways to assess the user in the resolution of conflicts once they have been detected. A tool supporting such facilities would of course require a high user interaction to assess in key points concerning the intended semantics of the conflicting modifications, but a great deal of information could be inferred, and even some decisions automatically taken, from the base collaboration contract and the involved modification operators.

Tool implementation and validation

The implementation of a working case tool, or the integration of this work into an existing software development environment that supports reuse contracts along the software development process is one of the most important pending tasks. Also fundamental for the validation of this and the original formulation of reuse contracts is the application of the tool in a case study.

Bibliography

[BA96] Software Change Impact Analysis. S. Bohner, R. Arnold, IEEE Computer Society Press ISBN 0-8186-7384-2, 1996

[Barbedette91] Schema modifications in the LISPO2 Persistent Object-Oriented Language. Gilles Barbedette. ECOOP '91 Proceedings. LNCS 512. Springer Verlag. 1991.

[Beck97] Smalltalk Best Practice Patterns. K. Beck. Prentice-Hall, 1997. ISBN 0-13-476904-X

[BHR95] Program Integration for Languages with Procedure Calls. D. Binkley, S. Horwitz, T. Reps. ACM Transactions of Software Engineering Methodology 4(1): 3-35, ACM Press, January, 1995.

[BMRSS97] Pattern Oriented Software Architecture- A System of Patterns. F. Buschmann, R. Meunier, H. Rohner, P. Sommerlad, M. Stal. John Wiley & Sons (ed.). ISBN 0-471-95869-7. 1996.

[Booch99] The Unified Modelling Language User guide. G. Booch, J. Rumbaugh, I. Jacobson. Addison-Wesley, Reding, Mass., 1999.

[BOSW98] Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In OOPLSA'98.

[Brant98] Creating Tools in HotDraw by Composition. J. Brant, R. Johnson. University of Illinois, Urbana-Champaign

[Cornelis97] Reuse Contract as a Modular System in Statically Typed Object-Oriented Languages. Gerrit Cornelis. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, 1997.

[DeHondt98] A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems. Koen De Hondt. Ph. D. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, 1998.

[D'Hondt98] Managing the Evolution of Changing Software Requirements. Maja D'Hondt. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, 1998.

[DL96] Forcing Behavioral Subtyping Through Specification Inheritance. Krishna Kishore Dhara and Gary T. Leavens. In Proceedings 18th International Conference on Software Engineering, Berlin, Germany, pages 258-267. IEEE,1996.

[Flanagan97] Java in a Nutshell: A Desktop Quick Reference. D. Flanagan. O'Reilly & Associates (ed.), 1997.

[GHJV94] Design Patterns: Elements of Reusable Object-Oriented Software. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison-Wesley, 1994.

[GJS96] The Java Language Specification. J Gossling, B. Joy, G. Steele. Addison-Wesley. 1996

[Johnson92] Documenting Frameworks Using Patterns. Ralph E. Johnson. In proceedings of Conference on Object-Oriented Programming, Systems, and Applications, pages 63-76. ACM, 1992

[KGHWTC94] Change Impact Identification in Object Oriented Software Maintenance. D. Kung, J. Gao, P. Hsia, F. Wen, Y. Tokoshima, C. Chen. In [BA96], pp. 197-206. 1994.

[KP88] A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. G. E. Krasner, S. T. Pope. JOOP, pp26-49, Aug. 1998

[Lamping93] Typing the specialisation interface. J. Lamping. In Proceedings OOPSLA '93, ACM SIGPLAN Notices, pages 201-214, oct 1993.

[Lea95] Roles Before Objects. D. Lea, http://gee.cs.oswego.edu/dl/rp/roles.html

[Leblang94] The CM challenge: Configuration that Works. D. B. Leblang. In Configuration Management, W. F. Ticchy (ed.) Trends in software, Vol. 2, pp. 1-38, Wiley, 1994.

[LP90] Inside Smalltalk. W. Lalonde, J. Pugh. Prentice-Hall, 1990.

[Lucas97] Documenting Reuse and Evolution with Reuse Contracts. Carine Lucas. Ph. D. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, 1997.

[LH89] Assuring good style for object-oriented programs. K. Lieberherr, I. Holland. IEEE software, pages 38-48. 1989

[LW94] A Behavioral Notion of Subtyping. Barbara Liskov, Jeannette M. Wing. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16.

[CCHO89] Interfaces for Strongly Typed OO Programming, Canning, Cook, Hill, Olthoff. OOPSLA 89

[HotDraw] RoleModel Software, Inc. HotDraw for Java can be found at http://www.rolemodelsoft.com/products/hotdraw

[JO93] Refactoring and Aggregation. R. Johnson, W. Opdyke. Proceedings of ISOTAS ’93 (International Symposium on Object Techniques for Advanced Software). November 1993.

[OJ93] Creating abstract superclasses by refactoring. W. Opdyke, R. Johnson. Proceedings of 1993 ACM Computer Science Conference, pp 66-73, ACM Press, 1993.

[MacLennan89] Functional Programming Practice and Theory. B. MacLennan. Addison-Wesley Publishing Company ISBN 0-201-13744-5. 1989.

[Mens98] Giving Precise Semantics to Reuse in UML. T. Mens, C. Lucas, P. Steyaert. Proceedings of the International Workshop on Precise Semantics for Software Modelling Techniques, pp. 73-89, Kyoto, Japan, April 1998.

[Mens99a] A Formal Foundation for Object-Oriented Software Evolution. Tom Mens. Ph. D. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, 1998.

[Mens99b] Supporting Reuse and Evolution of UML Models. In Proceedings of <<UML>>’98 International Workshop, LNCS 1618, Springer-Verlag, 1999.

[Mens99c] Automated Support for Software Evolution in UML. T. Mens, T. D’Hondt. Submitted to the Automated Software Engineering, Kluwer Academic Publishers, 1999.

[Mezini97] Maintaining the consistency of Class Libraries During their Evolution. Mira Mezini. Proceedings of OOPSLA ’97, ACM SIGPLAN Notices, 32(10): 1-21, ACM Press, 1997.

[PW92] Dewayne E. Perry, Alexander L. Wolf: Foundations for the study of software architecture. Software Engineering Notes, vol 17, no 4,October 1992, ACM Sigsoft, 1992.

[Richner99] Using Recovered Views to Track architectural Evolution. Tamar Richner. Presented at ECOOP 99 Workshop on Object-Oriented Architectural Evolution. Tuesday June 15th 1999

[RBI95] Configuration Management Tools. W. Rigg, C. Burrows, P. Ingram. OVUM Ltd., 1995.

[SLMD96] Reuse Contracts: Managing the Evolution of Reusable Assets. P. Steyaert, C. Lucas, K. Mens, T. D’Hondt. Proceedings of OOPSLA ’96, ACM SIGPLAN Notices, 31(10): 268-286, ACM Press, 1996.

[Westfechtel91] Structured Oriented Merging of Revisions of Software Documents. B. Westfechtel. Proceedings of the 3rd International Workshop on Software Configuration Management, P. Feiler (ed.), pp. 68-79, ACM Press, 1991.

[Wuyts99] Declarative Reasoning about the Structure of Object-Oriented Systems. Roel Wuyts. In Proceedings TOOLS USA’’8, IEEE Computer Society Press, 1998.

Appendix A – Formal Model

Important notation conventions

- We note a domain definition with '**=**'. We note elements definition with '**:**'.
- We use several usual operators in domains definitions, whose meaning is:
  - **+** disjoint union
  - **x** cartesian product
  - **→** function application
  - **P(...)** powerset
- The expression **Dom(f)**, where **f: A → B** denotes the set of elements from **A**, for which f is defined.
- We will note the application of curried functions as if they were multiple parameter functions. This is an abuse of notation in order to make our definitions more readable. We follow the general functional programming languages notation conventions [MacLennan89], and we base on the isomorphism that exist between functions of the form **f: A → B →C** and functions of the form **f': (AxB) → C**. We will apply functions of the first form as if they were from the second form, because there is an isomorphism between them.

Collaboration Contracts model

Name domains

The domain for all the names used in collaboration contracts is defined by **Name.** This domain does not include 'self' and 'super'.

Name = CCNam + PartNam + AcqNam + MethNam
PartNam = ClassNam + InterfNam

Collaboration contracts

CCStructure = Participants x ClassHierarchy x InterfHierarchy x ImplemRelation
ClassHierarchy = ClassNam → ClassNam
InterfHierarchy = InterfNam → P(InterfNam)
ImplemRelation = ClassNam → P(InterfNam)

collContract : CCNam → CCStructure
parts : CCNam → Participants

cc → Π_1 (collContract(cc))
class_hier : CCNam → ClassHierarchy
cc → Π_2 (collContract(cc))
interf_hier : CCNam → InterfHierarchy
cc → Π_3 (collContract(cc))
implem_rel : CCNam → ImplemRelation
cc → Π_4 (collContract(cc))
participantNames: CCNam → P(PartNam) :
cc → Dom(parts(cc))
classNames: CCNam → P(ClassNam):
cc → participantNames(cc) ∩ ClassNam
interfaceNames: CCNam → P(InterfNam):
cc → participantNames(cc) ∩ InterfNam

Participants

Participants = PartNam → Participant
Participant = Class + Interface
participant: CCNam → Participants:
cc → p → (parts(cc)) (p)

Classes & Interfaces

Class = Signature x Spec_Int x Acq_clause
Interface = Signature
Acq_clause = AcqNam → Type
Signature = MethNam → Type
Spec_Int = MethNam → SC
Type = PartNam

sign: Participant →Signature:
p → Π_1(p)
specInt: Class → Spec_Int:
c → Π_2(c)
acq: Class → Acq_clause:
c → Π_3(c)

Acquaintances

class_acq: CCNam → ClassNam → Acq_clause:
cc → c → acq(participant(cc, c))
class_ownAcqNames: CCNam → ClassNam → P(AcqNam):
cc → c → Dom(acq(participant(cc, c)))
class_ownAcqType : CCNam → ClassNam → AcqNam → Type:
cc → c → ac → acq(participant(cc, c), ac)
class_acqNames: CCNam → ClassNam → P(AcqNam):
cc → c → class_ownAcqNames(cc, c) , if not(has_ancestor(cc, c))
class_ownAcqNames(cc, c) ∪ class_acqNames(cc, superclass(cc, c))
, otherwise

class_acqType : CCNam → ClassNam → AcqNam → Type:
cc → c → ac → class_ownAcqType , if ac ∈ class_ownAcqNames(cc, c)
class_ownAcqType(cc, sc, ac), if ac ∉ class_ownAcqNames(cc, c)
where sc ∈ ancestors(cc,cp) ∧ ac ∈ class_ownAcqNames(cc, c)

Signatures

part_sign: CCNam → PartNam → Signature :
cc → p → sign(participant(cc, p))
part_ownMethNames: CCNam → PartNam → P(MethNam):
cc → p → Dom(part_sign(cc, p))
part_ownMethRetType: CCNam → PartNam → MethNam → Type :
cc → p → m → part_sign(cc, p, m)
part_methRetType: CCNam → PartNam → MethNam → Type :
cc → p → m →
part_ ownMethRetType(cc, p, m) , if m ∈ part_ownMethNames(cc, p)
part_ownMethRetType(cc, sp, m), , if m ∉ part_ownMethNames(cc, p)
where sp ∈ ancestors(cc, p) ∧ m ∈ part_ownMethNames(cc, sp)

In the previous definition **sp** can be any participant from the set of all the ancestors that define the method. This makes no problem over well-formed collaboration contracts because all the ancestors defining method m will have the same return type.

part_methNames: CCNam → PartNam → P(MethNam):
cc → p → $\cup_{sp \in ancestors(cc, p)}$ part_methNames(cc, si) ∪ part_ownMethNames(cc, p)
, if has_ancestor(cc, p)
part_ownMethNames(cc, p) , otherwise

Specialisation interfaces

SC = P(OpInvoc)
OpInvoc = NCastInvoc + CastInvoc
CastInvoc = Receiver x Type
NCastInvoc = Receiver x MethNam
Receiver = OpInvoc + ReceiverNam
ReceiverNam = AcqNam + 'super' + 'self'

class_specInt: CCNam → ClassNam → Spec_Int :
cc → c → specInt (participant(cc, c))
class_methSC: CCNam → ClassNam → MethNam → SC :
cc → c → m → class_specInt(cc, c, m)
sc_receiversNames: SC → P(ReceiverNam):
sc → { fst_rec(i) | i ∈ sc}
fst_recv: OpInvoc → ReceiverNam:
i → aux_fst_recv (receiver(i)) , if i ∈ NCastInvoc
aux_fst_recv (invoc(i)) , if i ∈ CastInvoc
aux_fst_recv: Receiver → ReceiverNam:
rec → rec , if rec ∈ ReceiverNam
fst_recv (rec) , if rec ∈ OpInvoc

receiver: NCastInvoc → Receiver :
nci → Π_1(nci)
method: NCastInvoc → MethNam :
nci → Π_2(nci)
invoc: CastInvoc → Receiver :
ci → Π_1(ci)
targetCastType: CastInvoc → Type :
ci → Π_2(ci)
type : CCNam → ClassNam → Receiver → Type :
(cc → c) → rec → c , if rec ∈ {'self'} ⊆ ReceiverNam
superclass(cc, c) , if rec ∈ {'super'} ⊆ ReceiverNam
class_acqType(cc, c rec) , if rec ∈ AcqNam ⊆ ReceiverNam
targetCastType(rec) , if rec ∈ CastInvoc ⊆ OpInvoc
part_methRetType(cc, type(cc, c, receiver(rec), method(rec))
, if rec ∈ NCastInvoc ⊆ OpInvoc
understands : CCNam → PartNam → MethNam → Bool :
cc → p → m → (m ∈ part_methNames(cc, p))

Hierarchies & Implementation relation
superclass : CCNam → ClassNam → ClassNam
cc → c → class_hier(cc, c)
superInts : CCNam → InterfNam → P(InterfNam)
cc → i → interf_hier(cc, i) , if i ∈ Dom(interf_hier(cc))
{ } , otherwise
specifications : CCNam → ClassNam → P(InterfNam)
cc → c → implem_rel(cc, c) , if c ∈ Dom(implem_rel(cc))
{ } , otherwise
descendants: CCNam → PartNam → P(PartNam):
cc → p → subclasses+(cc, p) , if p ∈ classNames(cc)
subInts$^+$ (cc, p) , if p ∈ interfaceNames(cc)
subclasses: CCNam → ClassNam → P(ClassNam):
cc → c → {c1 | superclass(cc, c1)=c}
subInts: CCNam → InterfNam → P(InterfNam):
cc → i → {i1 | i ∈ superInts(cc, i1)}
implementers: CCNam → InterfNam → P(ClassNam):
cc → i → {c | i ∈ specifications(cc, i)}
ancestors: CCNam → PartNam → P(PartNam):
cc → p → class_ancestors(cc,p) , if p ∈ classNames(cc)
superInts$^+$(cc, p) , if p ∈ interfaceNames(cc)
class_ancestors: CCNam → ClassNam → P(ClassNam):
cc → c → superclass(cc, c) ∪ class_ancestors(cc, superclass(cc, c))
, if has_ancestors(cc, c)
{ } , otherwise
has_ancestors: CCNam → PartNam → Bool:
cc → p → (p ∈ Dom (class_hier(cc)) , if p ∈ classNames(cc)

```
                    (p ∈ Dom (interf_hier(cc))    , if p ∈ interfaceNames(cc)
has_descendants: CCNam → PartNam → Bool:
    cc → p → true, if p ∈ classNames(cc) ∧ (∃ c ∈ classNames(cc): superclass(cc, c)=p)
             true, if p ∈ interfNames(cc) ∧ (∃ i ∈interfaceNames(cc): p ∈ subInts(cc, i))
             false, otherwise
```

Well formedness

```
wf_collContract : CCNam → Bool
    cc → wf_participant(cc) ∧ wf_classHier(cc) ∧ wf_interfHier(cc) ∧ wf_implemRel(cc)
wf_participants : CCNam → Bool
    cc → true      , if ∀ p ∈ Dom(parts(cc)): wf_part(cc, p)
         false     , otherwise
wf_part : CCNam → PartNam → Bool
    cc → p → wf_class(cc, p)    , if p ∈ classNames(cc)
             wf_interf(cc, p)   , if p ∈ interfaceNames(cc)
wf_class : CCNam → ClassNam → Bool
    cc → c → wf_acqClause(cc, c) ∧ wf_specInt(cc, c, class_specInt(cc, c))
                ∧ wf_signature(cc, c) ∧ Dom(class_specInt(cc, c))=Dom(part_sign(cc, c))
wf_interf : CCNam → InterfNam → Bool
    cc → i → wf_signature(cc, i)
wf_signature : CCNam → PartNam → Bool
    cc → p →  true        , if ∀ m ∈ part_ownMethNames(cc, p):
                                  part_methReturnType(cc, p, m) ∈ participantNames(cc)
              false       , otherwise
wf_acqClause : CCNam → ClassNam → Bool
    cc → c → true         , if  ( ∀ a ∈ class_ownAcqNames(cc, c):
                                    class_acqType(cc, c, a) ∈ participantNames(cc) )
                                    ∧ ( ∀ c1 ∈ ascendants(cc, c): compatAcqs(cc, c, c1) )
             false        , otherwise
compatAcqs : ClassNam → ClassNam → Bool
    c1 → c2 →( class_ownAcqNames(cc, c1) ∩ class_ownAcqNames(cc, c1) = ∅ )
wf_classHier : CCNam → Bool
    cc →  true,   if classHierInsideCC(cc)
                  ∧ ∀ c ∈ Dom(class_hier(cc)):  ( superclass(cc, c) ∉ descendants(cc, c)
                  ∧ compatSigns(part_sign(cc, c), part_sign(cc, superclass(cc, c))) )
          false, otherwise
classHierInsideCC : CCNam → Bool
    cc → true,     if  ( Dom(class_hier(cc)) ⊆ classNames(cc) )
                    ∧ ( ∀ c ∈ Dom(interf_hier(cc)): superClass(cc, c) ∈ classNames(cc) )
wf_interfHier : CCNam → Bool
    cc → true,     if  interfHierInsideCC(cc)
                       ∧ ∀ i ∈ Dom(interf_hier(cc)):
                                (superInts(cc, i) ∩ descendants(cc, i) = ∅)
                       ∧ (∀i1 ∈ superInts(cc, i):
                                compatSigns(part_sign(cc, i), part_sign(cc, i1)))
```

```
            false,    otherwise
interfHierInsideCC : CCNam → Bool
    cc → true,     if  ( Dom(interf_hier(cc)) ⊆ interfaceNames(cc) )
                        ∧ ( ∀ i ∈ Dom(interf_hier(cc)): superInts(cc, i) ⊆ interfaceNames(cc) )
compatSigns : Signature → Signature → Bool
    s1 → s2 → true,      if ( ∀ m ∈ Dom(s1)∩Dom(s2): s1(m) = s2(m) )
                false,   otherwise
wf_implemRel : CCNam → Bool
    cc → true,     if  implRelInsideCC(cc)
                                    ∧ ( ∀ c ∈ Dom(implem_rel(cc)): implem_interfs(cc, c) )
            false,    otherwise
implRelInsideCC : CCNam → Bool
    cc → true, if  ( Dom(implem_rel(cc)) ⊆ classNames(cc) )
                ∧ (∀ c ∈ Dom(implem_rel(cc)): specifications(cc, c) ⊆ interfaceNames(cc))
implem_interfs : CCNam → ClassNam → Bool
    cc → c → true,       if ∀ i ∈ specifications(cc, c):
                                        part_methNames(cc, c) ⊆ part_methNames(cc, i)
                false,   otherwise
wf_specInt: CCNam → ClassNam → Spec_Int → Bool:
    cc → c → si → true,   if ∀m ∈ Dom(si): wf_SC(cc, c, si(m)
                    false  , otherwise
wf_SC: CCNam → ClassNam → SC → Bool:
    cc → c → sc → true,          if ∀ s ∈ sc: wf_opInv(cc, c, s)
                    false,       otherwise
wf_opInv: CCNam → ClassNam → OpInvoc → Bool:
    cc → c → op →
        ( wf_receiver(cc, c, receiver(op))
          ∧ understands(cc, type(cc, c, receiver(op)), method(op)) ),    if op∈NCastInvoc
        ( wf_receiver(cc, c, invoc(op))
          ∧ wf_casting(cc, type(cc, c, invoc(op)), targetCastType(op) ), if op ∈ CastInvoc
        false,                                                           otherwise
wf_receiver: CCNam → ClassNam → Receiver → Bool:
    cc → c → rec →
        true                          , if rec ∈ {'self'} ⊆ ReceiverNam
        has_ancestors(cc, c))         , if rec ∈ {'super'} ⊆ ReceiverNam
        rec ∈ class_acqNames(cc, c)   , if rec ∈ AcqNam ⊆ ReceiverNam
        wf_opInv(cc, c, rec)          , if rec ∈ OpInvoc
        false                         , otherwise
wf_casting: CCNam → Type → Type → Bool:
    cc → source → target →
        true      , if source ∈ classNames(cc) ∧ target ∈ classNames(cc)
                        ∧ (source ∈ ancestors(cc, target) ∨ target ∈ ancestors(cc, source))
        true      , if source ∈ interfaceNames(cc) ∧ target ∈ interfaceNames(cc)
                        ∧ (source ∈ ancestors(cc, target) ∨ target ∈ ancestors(cc, source))
        true      , if (source ∈ classNames(cc) ∧ target ∈ interfaceNames(cc))
```

$\vee$ **(source $\in$ interfaceNames(cc) $\wedge$ target $\in$ classNames(cc))**
false , otherwise

Note that the operator $\vee$ used above is the exclusive disjunction, both alternatives cannot be true at the same time.

Mathematical definitions

Transitive closure

Given a function

$$f: X \rightarrow P(X)$$

where X is any of the defined domains, f^+ is the transitive closure of **f**, where:

$$f^1(x) = f(x)$$
$$f^2(x) = \cup_{z \in f(x)} f^1(z)$$
$$\ldots$$
$$f^n(x) = \cup_{z \in f(x)} f^{n-1}(z)$$
$$f^+(cc, x) = \cup_{n \in 1..} f^n(cc, x)$$

Union of functions

Given two functions of the same domain

$$f, g : A \rightarrow B$$

such that their domains are disjoint

$$Dom(f) \cap Dom(g) = \varnothing$$

the *union function* $f \cup g$ is defined by

$$f \cup g : A \rightarrow B : a \rightarrow \begin{cases} f(a) & \text{, if } a \in Dom(f) \\ g(a) & \text{, if } a \in Dom(g) \end{cases}$$

Reuse contracts

Definition

A *reuse contract* is a tuple that has

- a name
- a provider clause: CollaborationContract
- a contract type: tag
- a reuser clause

Definition

A *contract type* is an annotation that specifies the kind of reuse performed on the provider clause

ReuseContract = RCNam x CCNam x ContractType x ContractTypeVariant x ExtReuserClause
ContractType = { ‘Participant_extension’, ‘Participant_cancellation’,

‘Participant_refinement’, ‘Participant_coarsening’,
‘Context_extension’,
‘Context_cancellation’, ‘Context_refinement’, ‘Context_coarsening’ }

ContractTypeVariant = { ‘AddMethClass’, ‘AddMethInterf’, ‘RemMethClass’,
‘RemMethInterf’, ‘RegRef’, ‘SelfRef’, ‘SuperRef’, ‘RegCoar’,
‘SelfCoar’, ‘SuperCoar’, ‘AddClass’, ‘AddInterf’, ‘RemClass’,
‘RemInterf’, ‘RefAcq’, ‘RefClassHier’, RefInterfHier’,
‘RefImplemRel’, ‘CoarAcq’ ‘CoarClassHier’, ‘CoarInterfHier’,
‘CoarImplemRel’ }

ExtReuserClause = ReuserClause x ApplicabilityCond

ReuserClause = Participant_RClause + Context_RClause

Participant_RClause = PExt_RClause + PCanc_RClause + PRef_RClause +
PCoar_RClause

Context_RClause = CExt_RClause + CCanc_RClause + CRef_Rclause + CCoar_Rclause

ApplicabilityCond = CCStructure → Bool

For each kind of reuser clause we define when it is *compatible* with a collaboration contract. Given that a reuser clause is compatible with a collaboration contract, the *application* of the reuser clause to the collaboration contract is defined. This is done by specifying the characteristics of the *resulting* collaboration contract.

The application of a reuser clause to a collaboration contract can be defined with a function of type

rc_application : CCNam → RClause → CCStructure

that first check if the compatibility specification defined for the corresponding kind of reuser clause are valid. If this is not the case then the function returns a simple ill-formed collaboration contract meaning an error. If the compatibility specifications are valid, then it applies the application specification defined for that kind of reuser clause and returns the resulting collaboration contract.

The compatibility specification and application definition is defined above for each kind of reuser clause.

The **ApplicabilityCond** component of a reuse contract is defined, instead of the repeating clauses in some reuser clause in the original definition of reuse contracts. Applicability conditions are an orthogonal way to allow reusers to have more control on the conflicts that have to be signalled as applicability or evolution conflicts. This is the same as *repeating clauses* of the original definition of reuse contracts do in a fixed fashion. The applicability condition of a reuse contract is a function defined by the reuser, that is applied to the provider clause to check if the reuser clause is applicable to the provider clause. If the applicability condition returns false when applied to a certain collaboration contract, then the reuser clause is not applicable to it. In this way it allows to fine tune the signalling of applicability or evolution conflicts for each individual reuse contract.

The well-formedness function for reuse contracts is

wf_reuseContract: ReuseContract → Bool:
rc → wf_collcontract(providerClause(rc))
∧ compatType&Variant (rc)
∧ validApplCondition(providerClause(rc), applCondition(reuserClause(rc)))
∧ wf_collContract(rc_application(providerClause(rc), reuserClause(rc))

Reuser clauses

Participant reuser clauses

Participant extension reuser clauses:

PExt_RClause = ClassExt_RClause + InterfExt_RClause
ClassExt_RClause = ClassNam → (Signature x Spec_int)
InterfExt_RClause = InterfNam → Signature

pExt_sign: PExt_RClause → PartNam → Signature:
pe → p → Π_1(pe(p))
classExt_specInt: ClassExt_RClause → ClassNam → Spec_Int:
ce → c → Π_2(ce(c))
pExt_extParts : PExt_RClause → P(PartNam) :
pe → Dom(pe)
pExt_addedMeths: PExt_RClause → PartName → P(MethNam):
pe → p → Dom(pExt_sign(pe, p))

Definition

A class extension reuse clause R ∈ ClassExt_RClause is *compatible* with a Collaboration contract CC if the two following conditions hold:

- pExt_extParts(R) ⊆ classNames(CC)
- ∀ c ∈ pExt_extParts(R): pExt_addedMeths(R, c) ∩ part_ownMethNames(CC, c) = ∅

Definition

The *class extension* of a well-formed Collaboration Contract CC by a compatible class extension reuser clause R ∈ ClassExt_RClause is the Collaboration Contract CCCE, that satisfies all the following properties:

1. class_hier(CCCE)= class_hier(CC), interf_hier(CCCE)= interf_hier(CC), implem_rel(CCCE)= implem_rel(CC)
2. participantNames(CCCE) = participantNames(CC)
3. ∀ p ∈ participantNames(CCCE) \ pExt_extParts(R): participant(CCCE, p) = participant(CC, p)
4. ∀ c ∈ pExt_extParts(R):
   - class_acq(CCCE, c) = class_acq(CC, c)
   - class_sign(CCCE, c) = class_sign(CC, c) ∪ pExt_sign(R, c)
   - class_specInt(CCCE, c) = class_specInt(CC, c) ∪ classExt_specInt(R, c)

Definition

An interface extension reuse clause R ∈ InterfExt_RClause is *compatible* with a Collaboration contract CC if the two following conditions hold:

- pExt_extParts(R) ⊆ interfaceNames(CC)
- ∀ i ∈ pExt_extParts(R): pExt_addedMeths(R, i) ∩ part_ownMethNames(CC, i) = ∅

Definition

The *interface extension* of a well-formed Collaboration Contract CC by a compatible interface extension reuser clause R ∈ InterfExt_RClause is the collaboration contract CCIE, that satisfies all the following properties:

1. class_hier(CCIE)= class_hier(CC), interf_hier(CCIE)= interf_hier(CC), implem_rel(CCIE)= implem_rel(CC)
2. participantNames(CCIE) = participantNames(CC)
3. ∀ p ∈ participantNames(CCIE) \ pExt_extParts(R): participant(CCIE, p) = participant(CC, p)
4. ∀ i ∈ pExt_extClasses(R): class_sign(CCIE, i) = class_sign(CC, i) ∪ pExt_sign(R, i)

Participant cancellation reuser clauses:

For participant cancellation reuser clauses we also distinguish between class cancellation and interface cancellation, but when defining the compatibility condition and the result of the application (the participant cancellation collaboration contract) we do it in general for both kinds of participants, thus referring to the participant cancellation reuser class and the cancelled participants.

Participant cancellation reuser clauses only specify the minimal information needed to identify the method to be removed that is, the name of the participant and the name of the method. This is a difference with the original definition of reuse contracts, where participant cancellation reuser clauses also specify the specialisation clause that the method to be removed must have. This extra constraints for determining the item to be removed (and therefore the applicability of the reuse contract) are expressed in our model in the applicability condition function as explained before.

PCanc_RClause = ClassCanc_RClause + InterfCanc_RClause
ClassCanc_RClause = ClassNam → Signature
InterfCanc_RClause = InterfNam → Signature

pCanc_sign: PCanc_RClause → PartNam → Signature:
pc → p → pc(p)
pCanc_cancParts : PCanc_RClause → P(PartNam) :
pe → Dom(pe)
pCanc_cancMeths: PCanc_Rclause → PartName → P(MethNam):
pc → p → Dom(pCanc_sign(pc, p))

Definition

A participant cancellation reuse clause R ∈ PCancRClause_RClause is *compatible* with a Collaboration contract CC if the two following conditions hold:

- pCanc_cancParts(R) ⊆ participantsNames(CC)

Definition

The *participant cancellation* of a well-formed Collaboration Contract CC by a compatible participant cancellation reuser clause R ∈ PCanc_RClause is the Collaboration Contract CCPC, that satisfies all the following properties:

1. class_hier(CCPC)= class_hier(CC), interf_hier(CCPC)= interf_hier(CC), implem_rel(CCPC)= implem_rel(CC)
2. participantNames(CCPC) = participantNames(CC)
3. ∀ p ∈ participantNames(CCPC) \ pCanc_cancParts(R): participant(CCPC, p) = participant(CC, p)
4. ∀ p ∈ pCanc_cancParts(R): ∀ m ∈ pCanc_cancMeths(R)

- $m \notin$ Dom(part_sign(CCPC, p))
- $p \in$ classNames(CC) $\Rightarrow m \notin$ Dom(class_specInt(CCPC, p))

Participant refinement reuser clauses:

For participant refinement and participant coarsening reuser clauses we want to distinguish between regular, self and super variants, in order to better define the different conflicts that may appear when introducing or removing each kind of invocations. For simplicity we don't differentiate between the domains of regular, self and super operation invocations. As a consequence, the reuser clauses adding or removing the different kinds of operation invocations will all have the same form. In order to differentiate between the different kinds, we define functions that check the structure of the added or removed operation invocations to see if they are invocations of operations in an acquaintance (regular), self or super. These functions are not used in the reuse contracts definitions but are needed in evolution conflicts identification, in order to distinguish between different variants of a conflict.

The same as in participant cancellation, here we don't distinguish between the compatibility conditions and resulting participant refinement definition for the different kinds of participant refinement reuser clauses. We define them generally for all kinds of participant refinements.

Definition

In difference with the original definition of reuse contract, our participant refinement and participant coarsening reuser clauses do not use an extended interface, because as we explained above we don't need to define a repeating specialisation clause.

PRef_RClause = ClassNam → Spec_Int

pRef_refClasses : PRef_RClause → P(PartNam) :
pr → Dom(pr)
pRef_refMeths : PRef_RClause → ClassNam → P(MethNam) :
pr → c → Dom(pr(c))
pRef_specInt : PRef_RClause → ClassNam → Spec_Int :
pr → c → pr(c)
pRef_SC: PRef_RClause → ClassNam → MethNam → SC:
pr → c → m → Π_2 (pRef_specInt(pr, c, m))
is_regRef_RClause : PRef_RClause → Bool :
rr → true , if ∀c ∈ pRef_refClasses(pr): ∀m ∈ pRef_refMeths(rr, c):
sc_receiverNames(pRef_modifying(rr, c, m)) ⊆ AcqNam
false , otherwise
is_selfRef_RClause : PRef_RClause → Bool :
sc → true , if ∀c ∈ pRef_refClasses(pr): ∀m ∈ pRef_refMeths(sc, c):
sc_receiverNames(pRef_modifying(sc, c, m)) = {'self'}
false , otherwise
is_superRef_RClause : PRef_RClause → Bool :
sc → true , if ∀c ∈ pRef_refClasses(pr): ∀m ∈ pRef_refMeths(sc, c):
sc_receiverNames(pRef_modifying(sc, c, m)) = {'super'}
false , otherwise

Definition

A participant refinement reuser clause R ∈ PRef_RClause is *compatible* with a Collaboration contract CC if the following conditions hold, for every class name c ∈ pRef_refClasses(R):

- pRef_refClasses(R) ⊆ classNames(CC)
- ∀ c ∈ pRef_refClasses(R): pRef_refMeths(R, c) ⊆ part_ownMethNames(CC, c)

Definition

The *class refinement* of a well-formed Collaboration Contract CC by a compatible class refinement reuser clause R ∈ PRef_RClause is the Collaboration Contract CCPR, that satisfies all the following properties:

1. class_hier(CCPR)= class_hier(CC), interf_hier(CCPR)= interf_hier(CC), implem_rel(CCPR)= implem_rel(CC)
2. participantNames(CCPR) = participantNames(CC)
3. ∀ p ∈ pRef_refClasses(R) \ participantNames(CCPR): participant(CCPR, p) = participant(CC, p)
4. ∀ c ∈ pRef_refClasses(R):
   - class_acq(CCPR, c) = class_acq(CC, c)
   - class_sign(CCPR, c) = class_sign(CC, c)
   - Dom (class_specInt(CCPR, c)) = Dom(class_specInt(CC, c))
   - ∀ m ∈ part_ownMethNames(CCPR, c) \ pRef_refMeths(R, c) :

     class_methSC(CCPR, c, m) = class_methSC(CC, c, m)
   - ∀ m ∈ pRef_refMeths(R, c): class_methSC(CCPR, c, m)= class_methSC(CC, c, m)∪pRef_SC(R, c, m)

Participant coarsening reuser clauses:

The same as for participant cancellation, here we also distinguish regular, self and super variants of the reuser clauses using functions that check the structure of the removed operation invocations. Also the same as with participant cancellation and refinement here we don't distinguish between the compatible conditions and resulting participant coarsening definitions, for the different kinds of participant coarsening reuser clauses. We define them generally for all kinds of participant coarsenings.

PCoar_RClause = ClassNam → Spec_Int

pCoar_coarClasses : PCoar_RClause → P(PartNam) :
pc → Dom(pc)
pCoar_coarMeths : PCoar_RClause → ClassNam → P(MethNam) :
pc → c → Dom(pc(c))
pCoar_specInt : PCoar_RClause → ClassNam → Spec_Int :
pc → c → pc(c)
pCoar_SC: PCoar_RClause → ClassNam → MethNam → SC:
pc → c → m → Π_2 (pCoar_specInt(pc, c, m))
is_regCoar_RClause : RegCoar_RClause → Bool :
rc → true , if ∀c ∈ pCoar_coarClasses(pr): ∀m ∈ pCoar_coarMeths(rc, c):
sc_receiverNames(pCoar_modifying(rc, c, m)) ⊆ AcqNam
false , otherwise
is_selfCoar_RClause : SelfCoar_RClause → Bool :

sc → true , if ∀c ∈ pCoar_coarClasses(pr): ∀m ∈ pCoar_coarMeths(sc, c):
sc_receiverNames(pCoar_modifying(sc, c, m)) = {'self'}
false , otherwise

is_superCoar_RClause : SuperCoar_RClause → Bool :
sc → true , if ∀c ∈ pCoar_coarClasses(pr): ∀m ∈ pCoar_coarMeths(sc, c):
sc_receiverNames(pCoar_modifying(sc, c, m)) = {'super'}
false , otherwise

Definition

A participant coarsening reuser clause R ∈ PCoar_RClause is *compatible* with a Collaboration contract CC if the three following conditions hold, for every class name c ∈ pCoar_coarClasses(R):

- pCoar_coarClasses(R) ⊆ classNames(CC)
- pRef_refMeths(R, c) ⊆ part_ownMethNames(CC, c)

Definition

The *class coarsening* of a well-formed Collaboration Contract CC by a compatible class coarsening reuser clause R ∈ PCoar_RClause is the Collaboration Contract CCPR, that satisfies all the following properties:

1. class_hier(CCPR)= class_hier(CC), interf_hier(CCPR)= interf_hier(CC), implem_rel(CCPR)= implem_rel(CC)
2. participantNames(CCPR) = participantNames(CC)
3. ∀ p ∈ participantNames(CCPR) \ pCoar_coarClasses(R): participant(CCPR, p) = participant(CC, p)
4. ∀ c ∈ pCoar_coarClasses(R):
   - class_acq(CCPR, c) = class_acq(CC, c)
   - class_sign(CCPR, c) = class_sign(CC, c)
   - Dom (class_specInt(CCPR, c)) = Dom(class_specInt(CC, c))
   - ∀ m ∈ part_ownMethodNames(CCPR, c)\pCoar_coarMeths(R, c): class_methSC(CCPR, c, m)= class_methSC(CC, c, m)
   - ∀ m ∈ pCoar_coarMeths(R, c): class_methSC(CCPR, c, m)= class_methSC(CC, c, m)\pCoar_SC(R, c, m)

Context reuser clauses:

Context extension reuser clauses:

For context extension reuser clauses, the same as for participant cancellation we distinguish two variants: context extensions adding classes and context extensions adding interfaces. We define different domains for each, but the compatibility and result of application is defined in general for both kinds of reuser clauses.

CExt_RClause = CExtClass_RClause + CExtInterf_RClause
CExtClass_RClause = ClassNam → Class
CExtInterf_RClause =InterfNam → Interface

cExt_addedParts : CExt_RClause → P(PartNam) :
ce → Dom(ce)

cExt_addedPartDef: CExt_RClause → PartNam → Participant
ce → p → ce(p)

Definition

A context extension reuser clause R ∈ CExt_RClause is *compatible* with a Collaboration contract CC if the following condition hold:

- CExt_addedParts(R) ∩ participantNames(cc) = ∅

Definition

The *context extension* of a well-formed Collaboration Contract CC by a compatible context extension reuser clause R ∈ CExt_RClause is the collaboration contract CCCE, that satisfies all the following properties:

1. class_hier(CCCE)= class_hier(CC), interf_hier(CCCE)= interf_hier(CC), implem_rel(CCCE)= implem_rel(CC)
2. participantNames(CCCE) = participantNames(CC) ∪ cExt_addedParts(R)
3. ∀ p ∈ participantNames(CCCE) \ cExt_addedParts(R): participant(CCCE, p) = participant(CC, p)
4. ∀ p ∈ cExt_addedClasses(R): participant(CCCE, p) = cExt_addedPartDef (R, p)

Context cancellation reuser clauses:

For context cancellation reuser clauses, the same as for participant cancellation and context extension, we distinguish two variants: context cancellation removing classes and context cancellation removing interfaces. We define different domains for each, but the compatibility and result of application is defined in general for both kinds of reuser clauses.

CCanc_RClause = CCancClass_RClause + CCancInterf_RClause
CCancClass_RClause = P(ClassNam)
CCancInterf_RClause = P(InterfNam)

cCanc_removedParts : CCanc_RClause → P(PartNam) :
ce → ce

Definition

A context cancellation reuser clause R ∈ CExt_RClause is *compatible* with a Collaboration contract CC if the following condition hold:

- cCanc_removedParts(R) ⊆ participantNames(CC)

Definition

The *context cancellation* of a well-formed Collaboration Contract CC by a compatible context cancellation reuser clause R ∈ CCanc_RClause is the collaboration contract CCCC, that satisfies all the following properties:

1. class_hier(CCCC)= class_hier(CC), interf_hier(CCCC)= interf_hier(CC), implem_rel(CCCC)= implem_rel(CC)
2. participantNames(CCCC) = participantNames(CC) \ cCanc_removedParts(R)

Context refinement reuser clauses:

The same as for the participant refinement and coarsening, here we don't use the repeating clause, using instead the applicability condition. This is a difference with the original definition of reuse contracts. Our context refinements and coarsenings are always context free, the reuser clauses only specify the links that should be added or removed.

CRef_RClause = AcqRef _RClause + ClassHierRef_RClause + InterfHierRef _RClause + ImplRelRef _RClause
AcqRef _RClause = ClassNam → Acq_Clause
ClassHierRef_RClause = ClassHierarchy
InterfHierRef _RClause = InterfHierarchy
ImplRelRef _RClause = ImplemRelation

acqRef_refClasses : AcqRef_RClause → P(ClassNam) :
ar → Dom(ar)
acqRef_addedAcqClause : AcqRef_RClause → ClassNam → Acq_Clause :
ar → c → ar(c)
acqRef_addedAcqNames : AcqRef_RClause → ClassNam → P(AcqNams) :
ar → c → Dom(acqRef_addedAcqClause(ar, c))
chRef_refSubClasses : ClassHierRef_RClause → P(ClassNam) :
chr → Dom(chr)
ihRef_refSubInts : InterfHierRef_RClause → P(InterfNam) :
ihr → Dom(ihr)
ihRef_superInts : InterfHierRef_RClause → InterfNam → P(InterfNam) :
ihr → i → ihr(i) , if i ∈ Dom(ihr)
{ } , otherwise
irRef_refImplementors : ImplRelRef_RClause → P(ClassNam) :
irr → Dom(irr)
irRef_refSpecifications : ImplRelRef_RClause → ClassNam → P(InterfNam) :
irr → c → irr(c) , if c ∈ Dom(irr)
{ } , otherwise

Definition

An <u>acquaintance refinement</u> reuser clause R ∈ AcqRef_RClause is *compatible* with a Collaboration contract CC if the following two conditions hold:

- acqRef_refClasses(R) ⊆ classNames(cc)
- ∀ c ∈ acqRef_refClasses(R): acqRef_addedAcqNames(R, c) ∩ class_ownAcqNames(CC, c) = ∅

Definition

The *acquaintance refinement* of a well-formed Collaboration Contract CC by a compatible acquaintance refinement reuser clause R ∈ AcqRef_RClause is the collaboration contract CCAR, that satisfies all the following properties:

1. class_hier(CCAR)= class_hier(CC), interf_hier(CCAR)= interf_hier(CC), implem_rel(CCAR)= implem_rel(CC)
2. participantNames(CCAR) = participantNames(CC)
3. ∀ p ∈ participantNames(CCAR) \ acqRef_refClasses(R): participant(CCAR, p) = participant(CC, p)

4. $\forall$ c $\in$ acqRef_refClasses(R):
   - part_sign(CCAR, c) = part_sign(CC, c)
   - class_specInt(CCAR, c) = class_specInt(CC, c)
   - class_acq(CCAR, c) = class_acq(CC, c) $\cup$ acqRef_addedAcqClause(R, c)

Definition

A <u>class inheritance refinement</u> reuser clause R $\in$ ClassHierRef_RClause is *compatible* with a Collaboration contract CC if the following condition hold:

- chRef_ refSubClasses(R) $\cap$ Dom(class_hier(CC)) = $\varnothing$

Definition

The *class inheritance refinement* of a well-formed Collaboration Contract CC by a compatible class inheritance refinement reuser clause R $\in$ ClassHierRef_RClause is the collaboration contract CCCHR, that satisfies all the following properties:

1. participants(CCCHR) = participants(CC), interf_hier(CCCHR)= interf_hier(CC), implem_rel(CCCHR)= implem_rel(CC)
2. class_hier(CCCHR) = class_hier(CC) $\cup$ R

Definition

A interface inheritance refinement reuser clause R $\in$ InterfHierRef_RClause is always compatible with any Collaboration contract CC.

Definition

The *interface inheritance refinement* of a well-formed Collaboration Contract CC by a compatible interface inheritance refinement reuser clause R $\in$ InterfHierRef_RClause is the collaboration contract CCIHR, that satisfies all the following properties:

1. participants(CCIHR) = participants(CC), class_hier(CCIHR)= class_hier(CC), implem_rel(CCIHR)=implem_rel(CC)
2. Dom(interf_hier(CCIHR)) = Dom(interf_hier(CC)) $\cup$ ihRef_refSubInts(R)
3. $\forall$ i $\in$ Dom(interf_hier(CCIHR)): superInts(CCIHR, i) = (superInts(CC, i) $\cup$ ihRef_superInts(R, i))

Definition

A <u>implementation relation refinement</u> reuser clause R $\in$ ImplRelRef_RClause is always *compatible* with any Collaboration contract CC.

Definition

The *implementation relation refinement* of a well-formed Collaboration Contract CC by a compatible implementation relation refinement reuser clause R $\in$ ImplRelRef_RClause is the collaboration contract CCIRR, that satisfies all the following properties:

1. participants(CCIRR) = participants(CC), interf_hier(CCIRR)= interf_hier(CC), class_hier(CCIRR)= class_hier (CC)
2. Dom(implem_rel(CCIRR)) = Dom(implem_rel(CC)) $\cup$ irRef_refImplementors(R, c)

3. $\forall$ c $\in$ Dom(implem_rel(CCIRR)): specifications(CCIRR, c) = (specifications(CC, c) $\cup$ irRef_refSpecifications(R, c))

Context coarsening reuser clauses:

CCoar_RClause = AcqCoar _RClause + ClassHierCoar_RClause +
InterfHierCoar _RClause + ImplRelCoar _RClause
AcqCoar _RClause = ClassNam $\rightarrow$ P(AcqNam)
ClassHierCoar_RClause = P(ClassNam)
InterfHierCoar_RClause = InterfHierarchy
ImplRelCoar_RClause = ImplemRelation

acqCoar_coarClasses : AcqCoar_RClause $\rightarrow$ P(ClassNam) :
ac $\rightarrow$ Dom(ac)
acqCoar_removedAcqNames : AcqCoar_RClause $\rightarrow$ ClassNam $\rightarrow$ P(AcqNams) :
ac $\rightarrow$ c $\rightarrow$ ac(c)
chCoar_coarSubClasses : ClassHierCoar_RClause $\rightarrow$ P(ClassNam) :
chc $\rightarrow$ Dom(chc)
ihCoar_coarSubInts : InterfHierCoar_RClause $\rightarrow$ P(InterfNam) :
ihc $\rightarrow$ Dom(ihc)
ihCoar_superInts : InterfHierCoar_RClause $\rightarrow$ InterfNam $\rightarrow$ P(InterfNam) :
ihc $\rightarrow$ i $\rightarrow$ ihc(i) , if i $\in$ Dom(ihc)
{ } , otherwise
irCoar_coarImplementors : ImplRelCoar_RClause $\rightarrow$ P(ClassNam) :
irc $\rightarrow$ Dom(irc)
irCoar_coarSpecifications : ImplRelCoar_RClause $\rightarrow$ ClassNam $\rightarrow$ P(InterfNam) :
irc $\rightarrow$ c $\rightarrow$ irc(c) , if c $\in$ Dom(irc)
{ } , otherwise

Definition

An <u>acquaintance coarsening</u> reuser clause R $\in$ AcqCoar_RClause is *compatible* with a Collaboration contract CC if the following condition holds:

- acqCoar_coarClasses(R) $\subseteq$ classNames(cc)

Definition

The *acquaintance coarsening* of a well-formed Collaboration Contract CC by a compatible acquaintance coarsening reuser clause R $\in$ AcqCoar_RClause is the collaboration contract CCAC, that satisfies all the following properties:

1. class_hier(CCAC)= class_hier(CC), interf_hier(CCAC)= interf_hier(CC), implem_rel(CCAC)= implem_rel(CC)
2. participantNames(CCAC) = participantNames(CC)
3. $\forall$ p $\in$ participantNames(CCAC) \ acqCoar_coarClasses(R): participant(CCAC, p) = participant(CC, p)
4. $\forall$ c $\in$ acqCoar_coarClasses(R):
   - part_sign(CCAC, c) = part_sign(CC, c)
   - class_specInt(CCAC, c) = class_specInt(CC, c)

- class_ownAcqNames(CCAC, c) = class_ownAcqNames(CC, c)\acqCoar_removedAcqNames(R, c)

Definition

A <u>class inheritance coarsening</u> reuser clause R ∈ ClassHierCoar_RClause is always *compatible* with any Collaboration contract CC.

Definition

The *class inheritance coarsening* of a well-formed Collaboration Contract CC by a compatible class inheritance coarsening reuser clause R ∈ ClassHierCoar_RClause is the collaboration contract CCCHC, that satisfies all the following properties:

1. participants(CCCHC) = participants(CC), interf_hier(CCCHC)= interf_hier(CC), implem_rel(CCCHC)=implem_rel(CC)
2. Dom(class_hier(CCCHC)) = Dom(class_hier(CC)) \ R
3. ∀ c ∈ Dom(class_hier(CCCHC)): superclass(CCCHC, c) = superclass(CC, c)

Definition

An <u>interface inheritance coarsening</u> reuser clause R ∈ InterfHierCoar_RClause is *compatible* with a Collaboration contract CC if the following condition hold:

- ihCoar_coarSubInts(R) ⊆ Dom(interf_hier(CC))

Definition

The *interface inheritance coarsening* of a well-formed Collaboration Contract CC by a compatible interface inheritance coarsening reuser clause R ∈ InterfHierCoar_RClause is the collaboration contract CCIHR, that satisfies all the following properties:

1. participants(CCIHC) = participants(CC), class_hier(CCIHC)= class_hier(CC), implem_rel(CCIHC)= implem_rel(CC)
2. ∀ i ∈ Dom(interf_hier(CC))\ihCoar_coarSubInts(R):
3. ∀ i ∈ ihCoar_coarSubInts(R): superInts(CCIHC, i) = superInts(CC, i)
   - (superInts(CC, i) \ ihCoar_superInts(R, i) = ∅) ⇒ (i ∉ Dom(interf_hier(CCIHC)))
   - (superInts(CC, i) \ ihCoar_superInts(R, i) /= ∅) ⇒ (i ∈ Dom(interf_hier(CCIHC)) ∧ (superInts(CCIHC, i) = superInts(CC, i) \ ihCoar_superInts(R, i))

Definition

A <u>implementation relation coarsening</u> reuser clause R ∈ ImplRelCoar_RClause is *compatible* with any Collaboration contract CC if the following condition hold:

- irCoar_coarImplementors(R) ⊆ Dom(implem_rel(CC))

Definition

The *implementation relation coarsening* of a well-formed Collaboration Contract CC by a compatible implementation relation coarsening reuser clause R ∈ ImplRelCoar_RClause is the collaboration contract CCIRR, that satisfies all the following properties:

1. participants(CCIRC) = participants(CC), interf_hier(CCIRC)= interf_hier(CC), class_hier(CCIRC)= class_hier (CC)
2. $\forall$ c $\in$ irCoar_coarImplementors(R): specifications(CCIRC, c) = specifications(CC, c)
3. $\forall$ c $\in$ Dom(implem_rel(CC))\irCoar_coarImplementors(R):
   - (specifications(CC, c) \ irCoar_coarSpecifications(R, c) = $\emptyset$) $\Rightarrow$ (c $\notin$ Dom(implem_rel(CCIRC)))
   - (specifications(CC, c) \ irCoar_coarSpecifications(R, c) /= $\emptyset$) $\Rightarrow$ (c $\in$ Dom(implem_rel(CCIRC)) $\wedge$

     (specifications(CCIRC, c) = specifications(CC, c) \ irCoar_coarSpecifications(R, c)

Appendix B – Table of conflicts

In this appendix we present the table resuming the rules for conflict detection of all the conflicts our model detects, and presented in chapter 6. The table contains as headers the modification operators we define, and in each entry specifies which conflicts can occur when combining the two corresponding modification operator.

As the whole table of conflicts is large, we show it in two halves. To understand how the partial tables compose the complete one, we give first a schema of the complete table.

Conflicts		Parch Ipsum Elementum		Cras ac Sacculus		Parch Ipsum Consilium		Cras ac Consilium		Pari relinquo			Cras ac ullamcorper				Pari conveniebat			Cras ac Convenientia			
		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Parch Ipsum Elementum	[illegible]	[illegible]						[illegible]		[illegible]	[illegible]	[illegible]		[illegible]	[illegible]	[illegible]				[illegible]	[illegible]	[illegible]	
	[illegible]		[illegible]						[illegible]							[illegible]	[illegible]						
Cras ac Sacculus	[illegible]			[illegible]																			
	[illegible]				[illegible]																		
Parch Ipsum Consilium	[illegible]					[illegible]		[illegible]		[illegible]	[illegible]	[illegible]				[illegible]	[illegible]	[illegible]					
	[illegible]						[illegible]		[illegible]	[illegible]	[illegible]	[illegible]											
Cras ac Consilium	[illegible]							[illegible]					[illegible]	[illegible]		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]			
	[illegible]								[illegible]	[illegible]	[illegible]	[illegible]			[illegible]			[illegible]	[illegible]	[illegible]			
Pari relinquo	[illegible]									[illegible]	[illegible]	[illegible]					[illegible]	[illegible]	[illegible]		[illegible]	[illegible]	
	[illegible]										[illegible]	[illegible]					[illegible]	[illegible]	[illegible]				
	[illegible]											[illegible]					[illegible]	[illegible]	[illegible]				
Cras ac ullamcorper	[illegible]												[illegible]										
	[illegible]													[illegible]		[illegible]							
	[illegible]														[illegible]								
	[illegible]															[illegible]							
Pari conveniebat	[illegible]																[illegible]	[illegible]	[illegible]				
	[illegible]																	[illegible]	[illegible]				
	[illegible]																		[illegible]				
Cras ac Convenientia	[illegible]																			[illegible]			
	[illegible]																				[illegible]		
	[illegible]																					[illegible]	
	[illegible]																						[illegible]

Appendix C – HotDraw Model-View-Controller collaboration contract

In this appendix we present the complete collaboration contract for the HotDrawMVC. Parts of it were introduced in graphical notation in chapter 3. The invocations corresponding to the main interactions in the Model-View-Controller pattern are highlighted, indicating the method invocation to which they correspond in interaction schema of the pattern, and the order of occurrence. The schema of the pattern is as presented in [BMRSS97] and depicted in Figure 4.11.

```
HotDrawMVC {
    Superclass = { (CanvasTool, SimpleEventHandler),
                   (ConstructionTool, CanvasTool),
                   (ShapeTool, ConstructionTool),
                   (ConnectingLineTool, ShapeTool),
                   (BasicFigure, BasicObservable),
                   (Shape, BasicFigure),
                   (LinearShape, Shape),
                   (Line, LinearShape),
                   (ConnectingLine, Line)}

    SuperInterface = { (DrawingCanvas, SequenceOfFigures),
                       (Drawing, SequenceOfFigures),
                       (Figure, Observable),
                       (LineFigure, Figure),
                       (FigureObserver, Observer)}

    Implements =  {  (SimpleEventHandler, EventHandler )
                     (SimpleDrawingCanvas, DrawingCanvas),
                     (SimpleDrawingCanvas, FigureObserver),
                     (SimpleDrawing, Drawing),
                     (BasicObservable, Observable),
                     (LinearShape, LineFigure) }

    Interface EventHandler{
        Bool handleEvent;
    }

    Class SimpleEventHandler{
        Bool handleEvent { this.mouseDown }; 1 - handleEvent
        Bool mouseDown {}
```

```
}

Class CanvasTool{
      DrawingCanvas canvas;
}

Class ConstructionTool {
      Figure figure;

      Bool mouseDown {  this.newFigure
                        canvas.addFigure };
      Figure newFigure { this.basicNewFigure };
      Figure basicNewFigure {}
}

Class ShapeTool {
      Int anchorX;
      Int anchorY;

      Bool mouseDown { super.mouseDown }
}

Class ConnectingLineTool {
      Figure basicNewFigure { };
      Bool mouseDown {  super.mouseDown,
                        ((LineFigure)figure).getNumberOfPoints,
                        canvas.otherFigureAt.requestConnection,
                        ((LineFigure)figure).setLocator,      2 - service
                        canvas.moveFigureBehind,
                        canvas.repaint  }
}

Interface SequenceOfFigures{
      Void addFigure;
      Void moveFigureBehind;
      Figure otherFigureAt
}

Interface DrawingCanvas{
      Void repaint;
}

Interface Drawing{
}

Interface Observer {
      Void update
}

Interface FigureObserver {
}

Class SimpleDrawingCanvas {
      Drawing drawing;
```

```
        Void addFigure { drawing.addFigure,
                         this.repaint };
        Figure otherFigureAt { drawing.otherFigureAt };
        Void repaint {};
        Void moveFigureBehind { drawing.moveFigureBehind,
                                this.repaint};
        Void update { this.repaint }        5 - display
}

Class SimpleDrawing {
        Vector figures;

        Void addFigure { figures.addElement };
        Figure otherFigureAt {};
        Void moveFigureBehind { figures.indexOf,
                                figures.removeElement
                                figures.insertElementAt }
}

Interface Observable {
        Void addObserver;
        Void deleteObserver
}

Interface Figure {
        Locator requestConnection
}

Interface LineFigure {
        Int getNumberOfPoints;
        Void setLocator
}

Class BasicObservable {
        Object observerList;

        Void addObserver {};
        Void deleteObserver {};
        Void notifyObservers { this.notifyObserver };                3.3 - notify
        Void notifyObserver { ((Observer)observerList).update }      4 - update
}

Class BasicFigure {
        Void notifyObserver {}
        Locator requestConnection {}
}

Class Shape {
        Void changedShape { this.notifyObservers };     3.2 - notify
        }

Class LinearShape {
        Int getNumberOfPoints {};
```

```
        Void setLocator { this.basicSetLocator,
                          this.changedShape};          3.1 - notify
        Void basicSetLocator {}
    }

    Class Line {
        Int getNumberOfPoints {}
    }

    Class ConnectingLine {
        Void basicSetLocator { this.figureFromLocator.deleteObserver
                               this.figureFromLocator.addObserver };
        Figure figureFromLocator {}
    }

    Class Locator {
    }

    Class Vector {
        Void addElement {};
        Int indexOf {};
        Void removeElement {};
        Void insertElementAt {}
    }

    Class Bool {
    }

    Class Int {
    }

    Class Void {
    }
}
```