Enforcing Structural Regularities in Software using IntensiVE

Johan Brichau [b,1] Andy Kellens [a,2] Sergio Castro [b] Theo D'Hondt [a]

[a] *Vrije Universiteit Brussel, Belgium*

[b] *Université catholique de Louvain, Belgium*

Abstract

The design and implementation of a software system is often governed by many different coding conventions, design patterns, architectural design rules, and other so-called *structural regularities*. To prevent a deterioration of the system's source code, it is important that these regularities are verified and enforced in subsequent evolutions of the system. The Intensional Views Environment (IntensiVE), presented in this article, is a tool suite for documenting such structural regularities in (object-oriented) software systems and verifying their consistency in later versions of those systems.

Key words: software evolution, logic metaprogramming, software documentation

1 Structural Regularities

In the design and implementation of (object-oriented) software systems, the application of various forms of coding conventions, design patterns, design rules, idioms and so on has become a widespread practice. The use of such *structural regularities* explicitly molds the implementation with design and implementation principles that intend to establish improved software qualities such as reusability, extensibility, comprehensibility, and so on. A visitor design pattern, for example, anticipates extensions of the implementation with operations over object trees. Similarly, naming conventions render implementation concepts explicit and improve the understandability of code, especially in collaborative development environments. In

Email addresses: `johan.brichau@uclouvain.be` (Johan Brichau), `andy.kellens@vub.ac.be` (Andy Kellens), `sergio.castro@uclouvain.be` (Sergio Castro), `tjdhondt@vub.ac.be` (Theo D'Hondt).

[1] Funded by a "FIRST" post-doc grant of the Région Wallonne, Belgium

[2] Postdoctoral scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

brief, *structural regularities* is a broad term describing various governing principles of the source code of a system, ranging from low-level, stylistic properties such as naming and coding conventions, over the use of language idioms to more high-level rules describing architectural constraints, design dependencies and the implementation of design patterns.

Structural regularities play an important role in the development process [12]. For example, a developer can communicate certain concepts that are only implicitly available in the implementation to other developers by consistently using intention-revealing names or patterns in the source code to characterize this concept and thus make it explicit. Furthermore, regularities aid in obtaining stylistically more uniform source code, leading to a more comprehensible and maintainable implementation [1]. Next to the aforementioned stylistic reasons for introducing regularities, the correct functioning of the system can depend on whether developers correctly adhere to certain regularities. When regularities expressing architectural or design rules are violated, this can result in erratic and incorrect behavior of a system. For example, when making use of technology such as object-oriented frameworks, when applying design patterns, or when particular platforms such as J2EE are employed, certain regularities are imposed on the source code that – if not correctly adhered to – can result in the introduction of bugs in the source code of a system.

In spite of their intended benefits, the consistent and meticulous application of *structural regularities* in the implementation of a software system is often problematic. The reason for this is that most regularities are not integrally part of the development process and programming languages of current-day implementation practices. Without any means to document and enforce them in the implementation, they can easily be violated in subsequent evolutions of the system. In order to prevent the quality of the source code from deteriorating it is therefore imperative that regularities can be enforced when the system evolves.

This article presents how IntensiVE[3], the Intensional Views Environment [11], is used to document structural regularities in the source code of a system and enforce their consistency during evolution of the system. Key to the tool is that it implements a framework for verifying structural source-code regularities, much in the style of unit testing. Developers can specify the regularities they deem interesting and subsequently invoke their verification. Although IntensiVE is implemented in Smalltalk and it integrates tightly with the VisualWorks development environment, it can equally-well verify regularities in Eclipse Java projects through a loose integration with the Eclipse environment.

[3] `http://www.intensional.be`

2 Intensional Views & Constraints

IntensiVE supports the documentation and verification of structural regularities using the concepts of *intensional views* and *constraints*. An intensional view is a set of source-code entities in the software's implementation that share some structural property. In many cases, this shared structural property (for example a coding convention) also denotes that these entities implement a shared concept. Therefore, typical intensional views are, for example, "all getter methods in the implementation", "all methods that invoke database operations" or "all exception handlers that only perform a logging operation". An important characteristic of intensional views is that these sets are not defined by enumeration but by means of an *intension*. An intension is an executable description that yields, upon evaluation, the set of entities belonging to the view. Although any programming language can be used to define these intensions, the IntensiVE tool is tightly integrated with the logic (meta)programming language SOUL [14] (a derivative of Prolog). Its declarative source-code queries are a powerful means for the definition of intensional views and we will use them throughout this article.

Consider, for example, the intensional view of all "getter" methods that is defined using the following query:

```
?method isMethodDeclaration,
?method isPublicMethod,
?method reads: ?field named: ?fieldName,
?method methodDeclarationHasName: {get?fieldName}
```

The above expresses *all* conditions that a source-code entity must fulfill to be part of the intensional view. We present queries such that each condition is shown on a separate line. Also note that variables start with *?* and that the syntax of the logic predicates follows Smalltalk's messages syntax. In this simple example, the first condition expresses that an entity belonging to the view (captured by the logic variable *?method*) must be a method declaration (using the logic predicate `isMethodDeclaration`). The following conditions specify that such a method must be public, that it must read a field *?field* named *?fieldName* and that the method's name must start with "get", followed by the field name. The evaluation of this intension yields all methods in the source code of the system that satisfy all these conditions and, consequently, populate the "getter methods" intensional view. In some cases, there exist exceptions to this general rule. For example, it may occur that some non-getter methods' names are prefixed with "get". Therefore, IntensiVE also permits to explicitly include or exclude entities that must or must not be part of the intensional view.

To verify and enforce an actual structural regularity, we impose constraints on the defined intensional views. Depending on the kind of regularity we wish to enforce, two different techniques can be used.

Alternative Views

A first kind of constraint can be expressed using *alternative views*. Such views define an alternative intension for an already existing intensional view. Upon evaluation, this alternative view must contain the exact same set of source-code entities as the original view. A typical alternative view for “getter methods” collects all methods that return an instance variable:

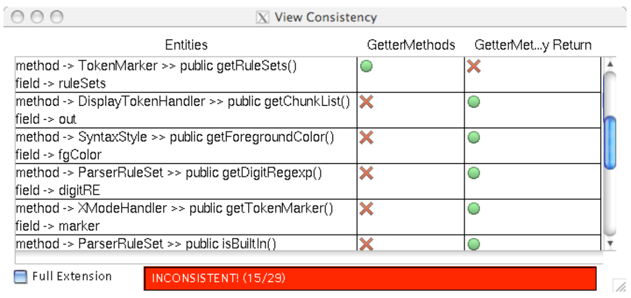
```
1 ?method isMethodDeclaration,
2 ?class definesMethod: ?method,
3 ?class definesVariable: ?field,
4 ?method returns: ?field
```

In this query, lines 2 and 3 express that the method *?method* is defined on a class *?class* that defines a variable *?field*. The last condition specifies that the method must return the instance variable *?field*. Note that this alternative view imposes a rather strict implementation regularity on a “getter method”. In most cases, additional queries are required to cope with possible variations in the implementation, such as array indexing and use of collection classes, indirect field returns, and so on. For brevity, we stick with this strict regularity to explain the concepts of intensional views.

IntensiVE automatically verifies the consistency of an intensional view by checking if the set of entities contained in all alternative views are equal. Any additional or missing entities in alternative views are reported to the developer, which indicates an inconsistency of the source-code with respect to the regularity. Using our example, we can detect methods that return an instance variable but that do not adhere to the naming convention, or vice-versa. To illustrate this, Figure 1 shows how such inconsistent methods are reported by IntensiVE upon verification of this constraint on the source-code of a package of the jEdit open-source project. The inconsistency window displays which (alternative) views each method is missing from, thus reflecting the nature of the inconsistency. The methods named `isBuiltIn` and `getTokenMarker`, for example, are not included in the default view (indicated by the red X in the first column). It means that they do not adhere to the naming convention. However, they do return an instance variable, and therefore were included in the alternative view (shown by the green dot in their last column). Inversely, the method `getRuleSets` follows the naming convention but does not return an instance variable. In the verification that is reported by this screenshot, there are 15 inconsistent methods on a total of 29 “getter methods”, which is an expected high number of inconsistencies: the source-code of jEdit was only used to demonstrate the reporting of inconsistencies and was not implemented with this convention in mind.

Intensional Relations

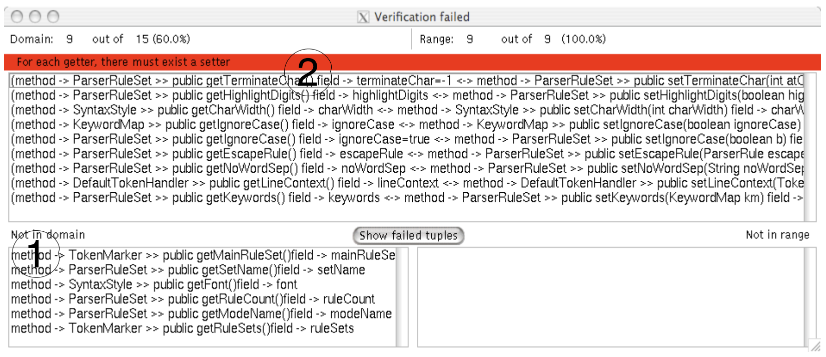
A second kind of constraints are *intensional relations*. Intensional relations impose structural conditions to be verified between the entities of one or two different views. The constraint is implemented as a combination of a source-code query and pre-defined quantifiers. The query expresses the conditions that entities of the dif-



Entities	GetterMethods	GetterMet...y Return
method -> TokenMarker >> public getRuleSets() field -> ruleSets	●	✗
method -> DisplayTokenHandler >> public getChunkList() field -> out	✗	●
method -> SyntaxStyle >> public getForegroundColor() field -> fgColor	✗	●
method -> ParserRuleSet >> public getDigitRegexp() field -> digitRE	✗	●
method -> XModeHandler >> public getTokenMarker() field -> marker	✗	●
method -> ParserRuleSet >> public isBuiltIn()	✗	●

INCONSISTENT! (15/29)

Fig. 1. Inconsistent "getter methods"



```
Domain:  9   out of  15 (60.0%)                Range:  9   out of  9  (100.0%)
For each getter, there must exist a setter
(method -> ParserRuleSet >> public getTerminateChar() field -> terminateChar=-1 <-> method -> ParserRuleSet >> public setTerminateChar(int atC
(method -> ParserRuleSet >> public getHighlightDigits() field -> highlightDigits <-> method -> ParserRuleSet >> public setHighlightDigits(boolean hig
(method -> SyntaxStyle >> public getCharWidth() field -> charWidth <-> method -> SyntaxStyle >> public setCharWidth(int charWidth) field -> charW
(method -> KeywordMap >> public getIgnoreCase() field -> ignoreCase <-> method -> KeywordMap >> public setIgnoreCase(boolean ignoreCase)
(method -> ParserRuleSet >> public getIgnoreCase() field -> ignoreCase=true <-> method -> ParserRuleSet >> public setIgnoreCase(boolean b) fie
(method -> ParserRuleSet >> public getEscapeRule() field -> escapeRule <-> method -> ParserRuleSet >> public setEscapeRule(ParserRule escape
(method -> ParserRuleSet >> public getNoWordSep() field -> noWordSep <-> method -> ParserRuleSet >> public setNoWordSep(String noWordSe
(method -> DefaultTokenHandler >> public getLineContext() field -> lineContext <-> method -> DefaultTokenHandler >> public setLineContext(Toke
(method -> ParserRuleSet >> public getKeywords() field -> keywords <-> method -> ParserRuleSet >> public setKeywords(KeywordMap km) field ->
Not in domain                          Show failed tuples                                   Not in range
method -> TokenMarker >> public getMainRuleSet()field -> mainRuleSe
method -> ParserRuleSet >> public getSetName()field -> setName
method -> SyntaxStyle >> public getFont()field -> font
method -> ParserRuleSet >> public getRuleCount()field -> ruleCount
method -> ParserRuleSet >> public getModeName()field -> modeName
method -> TokenMarker >> public getRuleSets()field -> ruleSets
```

Fig. 2. Intensional relation between "getter" and "setter" methods.

ferent views need to satisfy and the quantifiers express for which elements those conditions must hold (i.e. for all, for exactly one, etc...). For example, consider an intensional view that groups all "setter methods", in a similar way as we defined the "getter methods" view. Using an intensional relation between these views, we can enforce that *for all* classes that define a "setter method" for a particular field, there also *exists* a "getter method" for that same field. The verification of this relation in the jEdit source code is shown in Figure 2. IntensiVE reports the entities of the different views that violate the relation, i.e. when a "getter method" does not have a corresponding "setter method" that writes to the same field. The 9 entities of the source view that violate the relation are shown in pane (1). Pane (2) lists the corresponding entities of both relations that adhere to the relation. Now that we have explored the basic concepts and mechanisms of IntensiVE, we move on to more illustrative uses of the tool.

3 Enforcing the Abstract Factory Design Pattern

The Abstract Factory design pattern insulates the creation of objects (a.k.a. product objects) from their usage. Its implementation consists of an abstract class that defines an interface of abstract "product-creation" methods, and several concrete subclasses (a.k.a. concrete factories) that implement these methods. Instead of creating product objects directly, clients create objects by invoking the product-creation methods. The use of this pattern makes it possible to interchange the kind of products that are created by switching between different concrete factories. In addition, the pattern groups the creation of compatible products into separate concrete facto-

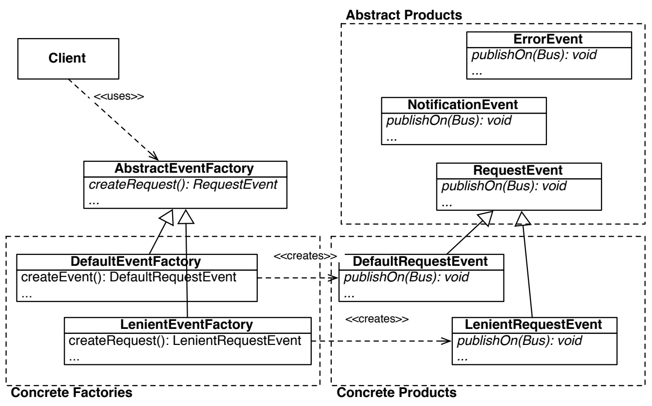


Fig. 3. Abstract Factory Design Pattern

ries. Figure 3 illustrates the structure of the design pattern, applied to the creation of "events".

Regularities

In this design pattern, a number of structural regularities need to be respected:

(1) The most important regularity is that objects created by the factory should not be created outside of the factory. If they are created outside of the factory, the main reason for using the pattern is lost. It would mean that incompatible objects can exist at execution time, probably resulting in faulty behavior.
(2) The factory needs to define product-creation methods for each kind of product. This regularity is partially enforced through the definition of abstract methods in the factory superclass, which require an implementation in each of the concrete factories. However, it is not checked that the abstract superclass defines a creation method for each product and that it effectively creates a new instance of such a product. In our example, we need to enforce that a product-creation method exists for each class of events in the library and that such a method effectively creates a new instance of such an event class.
(3) Each concrete factory must create compatible product objects. Product objects must be characterized as compatible by the developers by, for example, organizing them into separate hierarchies.
(4) Developers often use the naming convention that each concrete factory's name is also postfixed with "Factory".

Although each of the regularities will most probably be adhered to when the factory is first implemented, subsequent evolutions of the software implementation may easily break these regularities, especially in collaborative development environments. Most importantly, developers can violate the abstract factory through direct instantiation of objects that should be created through the factory. Additions of new kinds of events must also trigger the addition of an object-creation method to the factory. These regularities are enforced using the following intensional views and relations.

Intensional Views

Three intensional views form the heart of the IntensiVE documentation for this design pattern:

(1) The "Concrete Factories" view gathers all concrete factories implemented in the system. The query that defines this view collects all subclasses of the abstract factory superclass `AbstractEventFactory`. Regularities (1) and (4) are enforced by defining the following alternative views, which are supposed to contain the same elements. The first alternative gathers all classes in the system whose methods contain instance creation statements for classes identified as products. Such product classes are identified using the intensional view below. A second alternative view contains all classes whose name terminates with `Factory`.

(2) The "Products" intensional view groups all classes that are created by concrete factories. It gathers all classes for which an instance creation statement is found inside the implementation of the concrete subclasses of the `AbstractEventFactory` class. An alternative view is defined by collecting all concrete classes that are subtypes of the "Abstract Products" types, which are defined next. Verifying consistency of this view ensures that all possible products have a corresponding 'product creation' method in some concrete factory, which is part of regularity (2).

(3) The "Abstract Products" view groups the types that represent each kind of product to be created by a factory. These types are the return types of the abstract 'product-creation' methods of the abstract factory superclass. An alternative view that needs to ensure regularity (2) is often application-specific. In the particular case upon which we based this example, a set of Java interfaces was defined for each abstract product, following a naming convention.

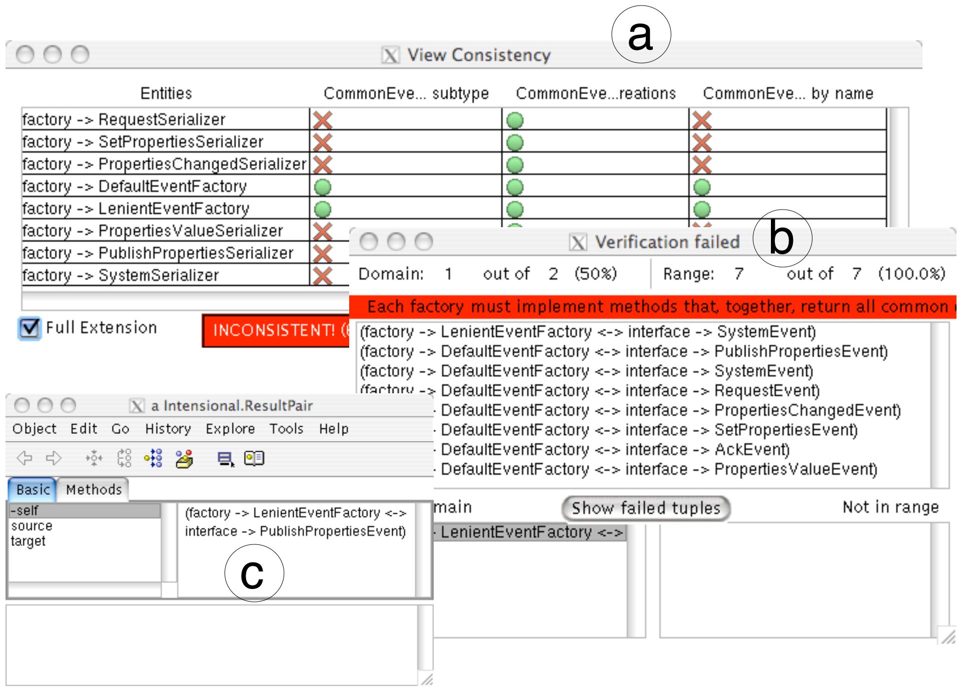
Intensional Relations

In addition to the intensional views and their alternatives, we define the following intensional relations:

(1) For each factory in the "Concrete Factories" view, all created products must be a subtype of exactly one abstract product in the "Abstract Products" view. The query that implements this relation searches for all classes created in a factory and verifies that they are a subtype of the same type (an abstract product). When this relation is enforced, it ensures regularity (3).

(2) Inversely, for each abstract product in the "Abstract Products" view, each concrete factory of the "Concrete Factories" view must implement a method that effectively creates an instance of a subtype of the abstract product. This relation completes the verification of regularity (2) and will report all kinds of products that are not created by a concrete factory.

Discovering Violations



View Consistency

Entities	CommonEve... subtype	CommonEve...reations	CommonEve... by name
factory -> RequestSerializer	✗	●	✗
factory -> SetPropertiesSerializer	✗	●	✗
factory -> PropertiesChangedSerializer	✗	●	✗
factory -> DefaultEventFactory	●	●	●
factory -> LenientEventFactory	●	●	●
factory -> PropertiesValueSerializer	✗	●	✗
factory -> PublishPropertiesSerializer	✗		
factory -> SystemSerializer	✗		

[✓] Full Extension INCONSISTENT! (

Verification failed

Domain: 1 out of 2 (50%) Range: 7 out of 7 (100.0%)

Each factory must implement methods that, together, return all common

```
(factory -> LenientEventFactory <-> interface -> SystemEvent)
(factory -> DefaultEventFactory <-> interface -> PublishPropertiesEvent)
(factory -> DefaultEventFactory <-> interface -> SystemEvent)
(factory -> DefaultEventFactory <-> interface -> RequestEvent)
DefaultEventFactory <-> interface -> PropertiesChangedEvent)
DefaultEventFactory <-> interface -> SetPropertiesEvent)
DefaultEventFactory <-> interface -> AckEvent)
DefaultEventFactory <-> interface -> PropertiesValueEvent)
```

main Show failed tuples Not in range

LenientEventFactory <->

a Intensional.ResultPair

Object Edit Go History Explore Tools Help

Basic Methods

-self
source
target

(factory -> LenientEventFactory <-> interface -> PublishPropertiesEvent)

Fig. 4. Verification of the Factory Design Pattern.

When verifying the aforementioned views and relations, IntensiVE reports all source-code entities that violate the imposed constraints as shown in Figure 4. Screen (a) presents the verification of the "Concrete Factories" view with respect to its alternatives. In this case, we can see that the `DefaultEventFactory` and `LenientEventFactory` classes are the only ones that fully adhere to the regularities of an abstract factory class. All other classes directly create instances of products and do not follow the naming scheme, thereby violating regularities (1) and (4). In screen (b), the verification of the intensional relation (2) shows that there is one factory (`LenientEventFactory`) that does not create a product (`PublishPropertiesEvent`). Screen (c) is an inspector opened on the violating entity, which displays the source-code entities involved in the violation.

4 Additional Applications

Bad Smell and Bug Detection

Many bad smells in code or code that can potentially lead to a runtime bug are much alike structural regularities except that they are undesirable properties of the code. IntensiVE can be used equally well to detect entities that expose such undesired structural properties. For example, the following query detects unused "getter methods" (a particular "dead code" bad smell):

```
?method isGetterMethod,
not(?somemethod calls: ?method),
```

The first condition in this query collects all entities that are part of the "getter methods" intensional view. The second condition filters only those methods for which there does not exist a caller method (*?somemethod*). The intensional view associated to this query can be inspected by the developers to investigate the necessity of fixing the bad smell or not.

Potential runtime bugs are often also detectable based on (undesired) structural properties of the source-code. For example, subtle errors can occur in Java when a constructor calls a non-final method of its class. In particular, an error occurs when the called method is overridden in a subclass and it references fields. These fields have not yet been initialized by the constructor of the subclass and thus contain the default initialization values, which is often an unexpected result. Although this bug is not very common, knowing that it exists in the code can save valuable time. Therefore, we define an intensional view using the following query that gathers the classes *?class*, their constructors *?constructor*, the called method(s) *?aMethod* and instance fields *?var* involved in the potential bug pattern. In summary, the query will find constructors that (transitively) invoke a method that is defined in the same class or any of its subclasses and which reads but does not write to a field defined on the same class.

```
?class isClassDeclaration,
    ?class definesConstructor: ?constructor,
    ?constructor callsTransitiveOnSelf: ?aMethod,
    ?subclass definesMethod: ?aMethod,
    ?subclass isSubClassOf: ?class,
    ?subclass definesVariable: ?var
    ?aMethod reads: ?var,
    not(?aMethod writesTo: ?var)
```

Visualized Regularities

Above, we have shown how IntensiVE directly supports reasoning over the implementation by using queries over source-code entities. In essence, the results of the source-code queries are also the results displayed in any of the consistency-checking tools of IntensiVE. Developers thus verify regularities in terms of the source-code entities that implement them. Although this works well for many regularities, often more appropriate design documentation that describes regularities is used in a software project. For example, the use of the State Design Pattern to implement state machines, is often documented using state diagrams. These diagrams describe the states and possible transitions for the state machine implementation. As a consequence, the regularities that must be enforced in the implementation are documented using state diagrams.

We have extended IntensiVE with a tool that visualizes the state diagrams exactly as they are implemented in the source code. In particular, the possible state transitions in the source code were gathered into an intensional view that lists pairs of “state” classes. Each pair thereby represents a possible state transition from the one state to the other state. The query that extracts this view reasons over the methods implemented on each state class (the source state) and extracts the creation of other state classes in the call-flow of these methods as possible destination states. Instead of portraying these intensional views as a collection of source-code entities, we passed on these entities to a visualisation script that draws their corresponding state diagrams. Figure 5 presents such a state diagram as it is shown in IntensiVE.

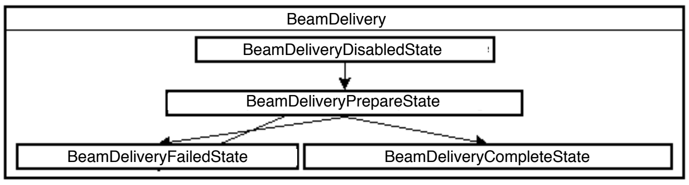


Fig. 5. The State diagram of a sample State Pattern implementation.

These diagrams reflect the actual state machine behavior as it is implemented in the source code using the State design pattern. The names of the states in the figure are the names of the classes that implement each state.

5 Related Work

Code checkers: Lint [10], P^3 [3], CheckStyle [2], FindBugs [9] and many others provide developers a means to verify a wide range of generally applicable regularities such as common mistakes, bad smells, bad programming style, violations of platform-specific constraints and so on. These tools provide a *dedicated* and often highly *optimized* means to identify locations in the source code that infringe on such regularities and can provide additional support, such as (semi-)automated correction of the detected infringements. While IntensiVE does not provide the same kind of dedicated support as code checkers, our tool suite is sufficiently versatile to express the same kinds of regularities as those verified by code checkers, as exemplified in Section 4. In addition, IntensiVE is not limited to verifying the regularities supported by code checkers, but also is able to document and verify a broad scope of e.g. non-stylistic and domain-specific regularities.

Architectural and design conformance checkers: are dedicated tools that aim at verifying a high-level description of a software system (e.g. design patterns, architectural descriptions, dependencies between components, . . .) with respect to the actual implementation of that system. Examples of these tools are Reflexion Models [13], Ptidej [7] and RevJava [6]. As illustrated by the Factory design pattern documentation in Section 3, IntensiVE can also be used to document regularities at the architectural and design level. Similar to the comparison to code checkers, IntensiVE is not specifically dedicated nor limited to these kinds of regularities but provides a general framework for documenting and verifying regularities.

Meta-programming systems: CCEL [4], Law-governed systems [12], IRC [5] and SCL [8] offer developers languages for writing meta-programs that reason about programs. One application domain of these meta-program systems is the implementation of meta-programs that verify source-code regularities or that allow for imposing constraints on the source code of a system. IntensiVE is related to this group of tools in that the intension is specified by means of a meta-program, expressed using the meta-language SOUL. The concepts of intensional views and constraints provide developers with a conceptual framework and tool support for expressing and verifying structural regularities, using these meta-programming systems.

References

[1] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[2] Checkstyle, December 2006. http://checkstyle.sourceforge.net.

[3] C. Depradine and P. Chaudhuri. P^3: a code and design conventions preprocessor for java. *Software - Practice and Experience*, 33(1):61–76, 2003.

[4] C. Duby, S. Meyers, and S. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Technical Conference Proceedings*, pages 99–115. USENIX Assoc., 10-13 1992.

[5] M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. In *Working Conference on Reverse Engineering (WCRE)*, pages 182–191. IEEE Computer Society Press, 2004.

[6] G. Florijn. Revjava - design critiques and architectural conformance checking for java software. Technical report, Software Engineering Research Centre (SERC), 2002.

[7] Y. Guéhéneuc. Three musketeers to the rescue – meta-modeling, logic programming, and explanation-based constraint programming for pattern description and detection. In *Workshop on Declarative Meta-Programming at ASE 2002*, 2002.

[8] D. Hou and J. Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[9] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[10] S. Johnson. Lint, a C program checker. In M. McIlroy and B. Kemighan, editors, *Unix Programmer's Manual*, volume 2A. AT&T Bell Laboratories, seventh edition, 1979.

[11] K. Mens and A. Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *Conference on Software Maintenance and Reengineering (CSMR)*, pages 239–248, 2006.

[12] N. Minsky. Law-governed systems. *Software Engineering Journal*, 6(5):285–302, 1991.

[13] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT)*, pages 18–28. ACM Press, 1995.

[14] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, January 2001.