Usage Contracts: Offering Immediate Feedback on Violations of Structural Source-code Regularities

Angela Lozano[a], Kim Mens[a], Andy Kellens[b]

[a] *Université catholique de Louvain, ICTEAM*
[b] *Vrije Universiteit Brussel, Software Languages Lab*

Abstract

Developers often encode design knowledge through structural regularities such as API usage protocols, coding idioms and naming conventions. As these regularities express how the source code should be structured, they provide vital information for developers using or extending that code. Adherence to such regularities tends to deteriorate over time because they are not documented and checked explicitly. This paper introduces *uContracts*, an internal DSL to codify and verify such regularities as 'usage contracts'. Our DSL aims at covering most common usage regularities, while still providing a means to express less common ones. Common regularities are identified based on regularities supported by existing approaches to detect bugs or suggest missing code fragments, techniques that mine for structural regularities, as well as on the analysis of an open-source project. We validate our DSL by documenting the structural regularities of an industrial case study, and analyse how useful the information provided by checking these regularities is for the developers of that case study.

1. Introduction

Being able to document and preserve architectural integrity and design knowledge of an application is important to increase its longevity [5]. Given that over time the actual implementation structure tends to drift away from initial architecture and design documents, programmers turn to structural

Email addresses: `angela.lozano@uclouvain.be` (Angela Lozano), `kim.mens@uclouvain.be` (Kim Mens), `andy.kellens@vub.ac.be` (Andy Kellens)

source code regularities, such as naming conventions, coding idioms and usage protocols to embed design knowledge in the implementation. Complying with these structural regularities facilitates future changes as they convey coding and design assumptions that need to be respected for the program to remain well designed and correct. In practice however, these regularities often get violated simply because they are not encoded or checked explicitly. Systematic violation of structural regularities can lead to several problems, such as premature ageing of the application or the introduction of defects. Matsumura et al. [30] report on a study in which they found that 32,7% of all bugs in a legacy system were caused by violations of implicit structural regularities.

In this paper we present *uContracts*, a tool for declaring structural source-code regularities (like API usage idioms and coding conventions) in a concise, explicit and verifiable way. The tool is conceived as a domain-specific language (DSL), embedded in the host language and IDE. The proposed DSL allows us to define low-level coding and implementation regularities, in terms of which higher-level regularities related to architecture, design or framework structure can be expressed. We use the generic term *structural source-code regularity* for any of the patterns that can be described in our DSL, since the DSL is not limited to expressing architectural, design or framework patterns, but can also express more low-level coding idioms.

Our *uContracts* DSL was prototyped[1] in the Pharo Smalltalk development environment, because of Smalltalk's facilities for prototyping such tools. Nevertheless, the idea behind the tool remains essentially independent of the language and will therefore be presented as such in the paper.

As illustrated schematically in Figure 1, in the *uContracts* DSL, structural regularities are expressed as *usage contracts* between two parties: a *provider*, i.e. the code entities that will be (re)used, and a *consumer*, i.e. the code entities that will use or extend the (re)used entities. A usage contract defines the expectations of and assumptions made by the reusable entities on the entities that reuse it. Consider for example an application or framework implementing some graphical editors. To maintain a consistent behaviour among the classes implementing these editors, they should respect a variety of implementation guidelines such as: extending the default editor instead of the abstract editor, complying with certain naming conventions, being

[1]`http://www.squeaksource.com/eContracts.html`

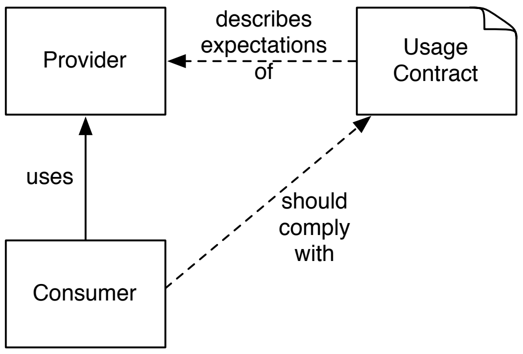


Figure 1: A usage contract, depicted as a contract between a provider and a consumer.

labeled with appropriate annotations, implementing the command pattern (i.e., providing an `isUndoable` method, and an `undo` method when necessary, and following a certain implementation template), etc. It is possible to define a usage contract that describes such regularities and to verify that the classes that implement such editors comply with these regularities.

The idea and terminology of declaring the assumptions that reusing code can make about the code it reuses as a *contract* between two parties, is loosely inspired by our previous work on *reuse contracts* [41]. The underlying approach is different however. The main purpose of the internal DSL presented in the current paper is to offer developers the necessary primitives – similar to what unit testing frameworks do – to express structural code regularities in a straightforward way, while remaining as close as possible to the syntax of the host language. Embracing the Pareto principle (a.k.a. the 80–20 rule), rather than offering a full language that allows us to express any possible regularity, we offer a simple and reduced set of language constructs that is sufficiently powerful to support a majority of frequently occurring structural regularities. Our DSL is complemented with tool support that, after each change to the source code, verifies automatically whether the source code still respects the encoded structural regularities and that provides fine-grained feedback regarding potential violations.

Our motivation for proposing an *internal* DSL to express structural regularities stems from our prior experience with developing and using an *external* DSL for that purpose. More specifically, in our earlier work on SOUL [9, 34] and the IntensiVE tool suite [6, 32, 33] we explored how to express structural regularities in a declarative program query language on top of an object-oriented host language. In spite of the good symbiosis of the declarative language and supporting toolsuite with the underlying language and its IDE, the fact that developers had to learn a new declarative language in which to express their regularities, turned out to be a major inhibitor towards adoption. This prior experience also taught us that, while having a Turing-complete specification language allowed for great flexibility, in practice describing most regularities required only the use of a small subset of the features of our language. We thus decided to develop instead a more lightweight and limited language as an internal DSL, in order for the developer to remain in his comfort zone when encoding these regularities.

The current paper introduces this DSL and presents the following contributions:

- The definition of the *uContracts* DSL for defining usage contracts that capture the structural regularities governing a software system;
- An argumentation that the *uContracts* DSL covers many common regularities, based on a thorough analysis of different kinds of commonly occurring regularities, a literature study and an investigation of the JHotDraw system;
- Prototype tool support in terms of an integration with the Smalltalk language and Pharo IDE;
- A first industrial case study to assess the usefulness of declaring and checking structural regularities by means of usage contracts.

2. Usage Contracts

We now introduce the notion of *usage contracts* in more detail, using an example from the Smalltalk implementation of the FAMIX meta model [10]. FAMIX is the model underlying the Moose reverse engineering tool suite and provides an extensible, object-oriented representation of various programming languages. Developers extending this meta model need to obey a number of structural regularities in order to guarantee proper functioning of the meta model. For example, when creating a subclass of `FAMIXSourcedEntity`, it is imperative that overridden versions of the method `copyFrom:within:` start with a super call.

Figure 2 depicts this regularity schematically as a usage contract. However, this structural regularity is not documented explicitly in the FAMIX

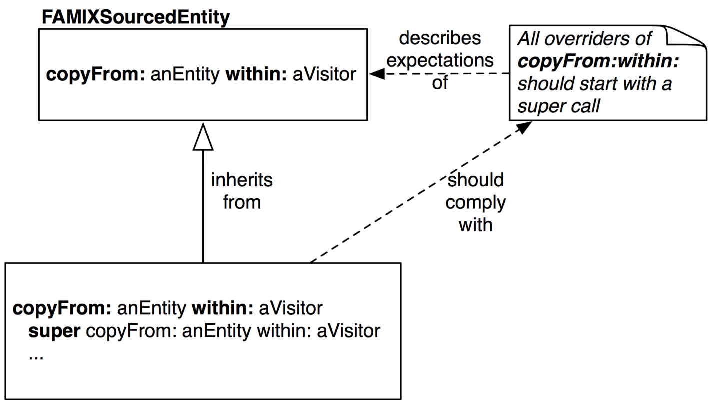

FAMIXSourcedEntity
copyFrom: anEntity within: aVisitor
describes expectations of
All overriders of copyFrom:within: should start with a super call
inherits from
should comply with
copyFrom: anEntity within: aVisitor
super copyFrom: anEntity within: aVisitor
...

Figure 2: Example of a structural regularity in FAMIX, depicted as a usage contract.

framework and is not checked automatically whenever `copyFrom:within:` gets overridden. The absence of such documentation poses a threat to the (re)usability of the framework [31, p.54]. Ideally, to use a framework successfully, the structural assumptions, rules and constraints made by that framework should be conveyed in a concise and explicit way to its users [31, p.54]. Having an accurate and standard way of expressing such usage contracts not only improves their awareness and comprehension but also allows for their automatic validation. But usage contracts are not restricted to application frameworks. In general, any object-oriented application can make certain assumptions that should be obeyed by its subclassers and users.

The above example thus illustrates the need for tools and approaches that:

1. Make the structural [2] contract between source code producers and consumers *explicit*;
2. Provide a means to *concisely and precisely* express the structural regularities defined by such a contract;
3. Provide support for *automatically checking* these regularities while coding;
4. Provide a means to *easily distribute* these contracts together with, or as part of, the original source code.

[2]In fact, a usage contract could be behavioural as well, but in this paper we focus mainly on contracts that describe structural regularities of more syntactic nature.

The approach presented in this paper aims at fulfilling the above requirements by:

1. Explicitly embracing the contract metaphor detailed above. Our approach provides developers the ability to document their software with **contract-like rules**;
2. Declaring these contracts in a simple, **domain-specific language** that is embedded in the host programming language. This language offers a minimal set of primitive language constructs for documenting the structural regularities most commonly occurring;
3. Providing an integration with the surrounding development environment that **checks compliance** of the code to the defined contracts on the fly, i.e. as soon as code gets added or modified;
4. Encoding the contracts using **classes and methods**, such that these contracts can be distributed and versioned together with the source code they govern.

3. Domain analysis

Before presenting our DSL in the next section, in this section we first provide a domain analysis of the kinds of regularities that our DSL should be able to express and verify. We define a 'structural regularity' as a constraint on the *structure* of a source-code entity that dictates how that source-code entity should be reused or extended. As this definition is quite broad, it is impossible to provide an exhaustive list of all possible kinds of structural regularities. However, structural regularities may cover structural characteristics such as:

- Which methods should a class implement? Which methods should be inherited, and which ones overridden in complex inheritance hierarchies?

- Implementation standards such as naming conventions, coding idioms, and patterns.

- Configuration requirements or dependencies such as initialization or method-call orders.

- Dependencies between source code entities. E.g., when overriding `equals`, you should override `hashCode` as well.

To determine which language concepts to include in our DSL, we performed an analysis of the domain of structural regularities, in two steps. First we performed a literature study of approaches that discuss structural regularities or that aim at identifying such regularities. From this literature study, we compiled a list of common types of structural regularities. Secondly, we manually catalogued the structural regularities of an open-source software system, and mapped the identified structural regularities onto the types of regularities found in the first step.

3.1. Types of regularities based on literature study

Structural relation	**References**	**JHotDraw Examples**
class *implements* method	[47]	3
method *creates instance of* class	[47]	3
method *overrides* ancestorMethod	[4]	4,5,6,7,12,15,17,18
method *calls* otherMethod	[47, 29, 27, 26, 28]	1,2,8,9,10,11,13,14,16,21
method *assigns to* field	[26]	
method *returns* statement	[43, 45]	6
method *contains* superSend	[4]	5,19,20
statement1 *comes before*|*after* statement2 *in* method	[46, 47, 3, 27, 7, 43]	8,10,14,16,21
statement *comes first*|*last in* method	[46, 47, 3, 27, 7, 43]	19,20
method *does not have* constraint	[43]	1,2,4,7,11
method *has* constraint1 *and*|*or* constraint2	[43]	8
method *has* constraint1 *if* constraint2	[7]	3,5,8,9,10,13,14,16,21

Table 1: Kinds of structural relations based on a literature study.

We analyzed the literature for typical structural regularities by considering two sources of information: 1) studies that have successfully used structural regularities for fault detection or code completion, and 2) a large body of work regarding techniques that mine for various kinds of structural regularities. Given the vast amount of papers that address either of these two topics, we limited the scope of our literature review to those papers that appeared in the main software engineering conferences with a strong interest in source code engineering (ICSE, MSR, ASE, ICSM, CSMR and WCRE), and to approaches that mine from source code *only*. That is, we *excluded* approaches that use code repository information (cvs, svn, git, etc), bug information, social media information about the code (e-mails, chats, forums, etc), and local history information.

By studying these papers, we distilled a *non-exhaustive* list of different kinds of structural relations, shown in Table 1. The first column presents the canonical form of the identified structural relation; the second column lists

```
/**
 * Deactivates the tool. This method is called whenever the user switches to another tool
 * Use this method to do some clean-up when the tool is switched.
 *  Subclassers should always call super.deactivate.
 * An inactive tool should never be deactivated.
 */
public void deactivate() {
        if (isActive()) {
                if (getActiveView() != null) {
                        getActiveView().setCursor(new AWTCursor(java.awt.Cursor.DEFAULT_CURSOR));
                }
                getEventDispatcher().fireToolDeactivatedEvent();
        }
}
```

Figure 3: Example of a structural regularity in the source code of JHotDraw 5.4b2.

the publications documenting the corresponding structural relation. (The third column will be explained in the next subsection.) Within this table we distinguish three categories of structural regularities:

- Regularities which describe that a source code entity (class or method) should exhibit some *structural* property (implementing or calling a particular method, assigning to some field, containing a super send, . . .);
- Regularities expressing the order of statements in a method (e.g., a method should start by doing a super send, a method should call `prepareUpdate` before calling `update`);
- Logic combinations of regularities and conditional constraints (e.g., *if* some method calls method `update`, *then* it should also call method `prepareUpdate`).

3.2. Analysis of a concrete system: JHotDraw

In order to assess to what extent the catalogue of structural regularities obtained from our literature analysis covers the most common structural regularities, we matched it with JHotDraw (version 5.4.bis1), a well-known open-source GUI framework in Java for drawing structured graphics, which was originally developed as a “design exercise” to illustrate some well-known design patterns. We chose JHotDraw because it is a unique example of a medium-sized application in which important structural rules are documented by means of source-code comments. For example, Figure 3 shows the implementation of a method `deactivate()` implemented by a class `Tool`.

Within the comments of the method we see the regularity (indicated in bold) that subclassers should always perform a super call.

To identify the various structural regularities of JHotDraw, we queried the source code comments for occurrences of the words *should*, *may*, *must*, *can(not)*, *could*, *ought*, *have*, *has*, *need*, and *require*, and then analyzed the context in which these words occurred to distill a (non-exhaustive) list of constraints. During this analysis, comments that were not related to structural regularities were discarded. Eventually, we obtained the list of 22 structural regularities below:

1. Methods inside class `AbstractConnector` must be used internally only.
2. The only caller of method `connectorVisibility` of class `Connector` should be the method `draw`.
3. The classes implementing interface `Tool` must implement method `activate` if method `isUsable` returns true.
4. The classes implementing interface `Figure` should override `basicDisplayBox` but should not override `displayBox`.
5. Subclasses of `AbstractFigure` should perform a super call in methods `deactivate` and `viewSelectionClass`.
6. Subclasses of `DrawApplet`, `DrawApplication` and `VersionRequester` should override method `getRequiredVersions` and return an array of strings.
7. Subclasses of `DrawApplication` should not override `basicDisplayBox` but should override `exit`.
8. The method `init` should be called after creating or loading a `CompositeFigure`, that is, after calling the method `new` or `read`.
9. Calling method `getLocator` requires cloning the instance (calling method `clone`) to avoid that the receiver of `getLocator` can change the internal behavior of a `LocatorHandle`.
10. Calls to the method `addInternalFrameListener` should occur before calling the method `add` when implementing or overriding the method `addToDesktop` in the class `MDIDesktopPane`.
11. The constructor of `StandardLayouter` with parameters should be preferred over the constructor without parameters.
12. Sublasses of `UndoableAdaptor` should override methods `undo` and `redo`.
13. Pop-up menus in subclasses of `CustomSelectionTool` must call `setAttribute`.
14. The status line must be created (i.e. call to `setStatusLine`) before a tool is set (i.e. call to `setTool`).
15. Method `addToDesktop` should be overridden instead of overloaded (seen in class `MDIDesktopPane`).
16. After calling `viewDestroying` on an object you cannot do anything else on that object (seen in class `ViewChangeListener`).
17. When extending the class `DrawApplication`, the methods `createOpenFileChooser` and `createSaveFileChooser` should be overridden.

18. When extending the class `StandardDrawingView`, the method `handleMouseEventException` should be overridden.
19. If method `mouseUp` of class `AbstractTool` is overridden, the last statement should be a super call.
20. If method `mouseDown` of class `AbstractTool` is overridden, the first statement should be a super call.
21. If you call `activate` or `deactivate` from the class `Tool` you should call `isActive` before (seen in class `DrawApplication`).
22. The names of the attributes of class `FigureAttributeConstant` should be used as suffixes of the attributes of class `ContentProducer` starting with the prefix $ENTITY$.

If we map these identified structural regularities onto our different kinds of structural regularities (the third column of Table 1), we see that – with the exception of field assignments – we effectively encountered examples of all the various kinds of structural regularities we identified. Conversely, all structural regularities encountered within JHotDraw are covered by the categories we identified with our literature study, except regularity #22 which expresses a naming convention between the attributes of two different classes.

3.3. Completeness of the DSL

The purpose of our *uContracts* DSL, which will be developed in the next section, is to offer a set of primitive constructs in terms of which developers can describe the structural regularities that govern their code. Just like JUnit [3] or SUnit [4] offer a kind of DSL that allows developers to express, document and execute unit tests, *uContracts* is a DSL that allows developers to express, document and validate the way in which a source code entity is expected to be used. This implies that usage contracts are often specific to a particular application (just like unit tests are), and that defining them thus requires expert developer knowledge about that application. Therefore, validating completeness of the DSL is not straightforward. Although it seems that the types of structural regularities identified in the two previous subsections do cover most common structural constraints, further validation with application experts is still necessary to confirm this. Section 6 will report on a first industrial validation that was already conducted.

[3]`http://www.junit.org`
[4]`http://sunit.sourceforge.net`

4. The *uContracts* DSL

In this section we provide an overview of the *uContracts* domain-specific language for declaring usage contracts. Revisiting the example of the FAMIX structural regularity introduced in Section 2 (Figure 2), that regularity can be expressed as follows in our DSL:

```
copyFromWithinWithCorrectSuperCall
 <selector:#copyFrom:within:>
 contract
   require:
    condition beginsWith:
     (condition doesSuperSend: #copyFrom:within:)
   if: (condition isOverridden)
```

This regularity is named `copyFromWithinWithCorrectSuperCall`, after its purpose. Quite literally, it expresses that we require overridden methods with selector[5] `#copyFrom:within:` to begin with a super call to the same message `copyFrom:within:`.

Whereas this part of the contract expresses what regularity the overridden methods should respect, another part of the contract specifies to what classes the contract is applicable. In this particular example, the usage contract should be applicable to all subclasses of `FAMIXSourcedEntity`:

```
classesInFAMIXSourcedEntityHierarchy
   <liableHierarchy: #FAMIXSourcedEntity>
```

Note that both parts of the description of this usage contract are actually syntactically valid Smalltalk code, making use of the keyword-style message syntax of this language and the notion of pragmas (a kind of annotations). Within the Smalltalk method describing the regularity, `contract` and `condition` are pseudo-variables that are part of the implementation of our DSL. Furthermore, since usage contracts and their structural regularities are defined directly by using classes and methods in the underlying Smalltalk language, we can readily use features of that language such as inheritance, to extend and reuse existing usage contracts and structural regularities.

In general, defining a usage contract requires the following steps and parts:

[5] *selector* is Smalltalk terminology for a method name

4.1. Defining a contract

Within our DSL, every usage contract is represented by a class that inherits from the class `ECContract`. Such a contract groups a set of related structural regularities that are applicable to the same context. Before declaring the structural regularity exemplified above, we should thus first create a new subclass named[6] `FAMIXSourcedEntityContract` of the class `EContract`.

4.2. Specifying the liable classes

The next step in declaring a usage contract lies in specifying the liable entities of the contract: the classes to which the contract is applicable. Within our DSL, we declare these liable classes by using a class method (static method) on the contract class. We already illustrated how to declare a usage contract that is applicable to all subclasses of `FAMIXSourcedEntity`, by defining a class method `classesInFAMIXSourcedEntityHierarchy` on the contract class `FAMIXSourcedEntityContract`.

The actual conditions that declare the liable classes are defined using pragmas. In the example above, the `liableHierarchy:` pragma with as argument `#FAMIXSourcedEntity` indicated that our usage contract was restricted to all classes in the hierarchy of the `FAMIXSourcedEntity` class. Our DSL currently supports the following pragmas for defining the liable classes:

`liableClass:aRegExp` indicates that the contract is applicable to all classes with name matching some regular expression `aRegExp`;

`liableHierarchy:aClassName` indicates that the contract applies to all classes in the class hierarchy of the class named `aClassName`;

`liablePackage:aRegExp` indicates that the contract applies to all classes, in all packages with name matching some regular expression `aRegExp`.

When the class method that declares the liable classes contains multiple pragmas, these pragmas then represent multiple conditions that *all* need to be satisfied. In other words, the liable classes of the contract are those classes that satisfy the *conjunction* of all conditions specified by the different pragmas in the class method (e.g., all the classes in a certain class hierarchy that belong to a particular package).

[6]It is good policy to name the contract class after the class or hierarchy over which it defines a usage contract.

In case a single usage contract requires the *disjunction* of conditions for specifying the liable classes of the contract, this can be done by specifying multiple class methods[7]. In other words, we consider a class to be liable if it matches all conditions of at least one of the class methods that specifies the liable classes. This feature allows us, for example, to specify that a usage contract is applicable to all classes of a package `A` *or* of a package `B`.

Furthermore, our DSL also provides pragmas that can be used within the class method that declares liable classes, to exclude exceptional cases:

`exceptClass:aRegExp` excludes all classes of which the name match some regular expression `aRegExp`;

`exceptHierarchy:aClass` excludes all classes that belong to the class hierarchy of `aClass`;

`exceptPackage:aPackage` excludes all classes that belong to `aPackage`.

4.3. Determining liable methods

A single usage contract can group multiple structural regularities defined over different sets of methods belonging to the liable classes of the contract. Each of these regularities is declared by means of a separate instance method on the contract class. Inside each such method, the liable methods (i.e., the methods which that structural regularity pertains to) are defined using pragmas.

In our FAMIX example, we declared a regularity over all methods named `copyFrom:within:`, by defining an instance method `copyFromWithinWithCorrectSuperCall` with pragma `<selector:#copyFrom:within:>`. In combination with the liable classes declared for this example, the structural regularity defined in the method `copyFromWithinCorrectSuperCall` is thus applicable to all methods named `#copyFrom:within:` implemented by a class in the hierarchy of `FAMIXSourcedEntity`.

Our DSL currently offers the following pragmas for declaring liable methods:

`selector:aRegExp` takes all methods whose name matches the regular expression `aRegExp`;

[7]The names of those class methods are not important as long as they define pragmas recognized by the DSL. Unrecognized pragmas are ignored.

`protocol:aRegExp` takes all methods that are classified in a protocol[8] matching the regular expression `aRegExp`.

As was the case for the pragmas defining the liable classes, multiple pragmas defining the liable methods can be combined within a single method to indicate a conjunction of conditions (e.g., to select all methods with a particular name in a certain protocol). Furthermore, exceptions to be excluded can be declared using the following pragmas:

`exceptSelector:aRegExp` excludes all methods of which the name matches the regular expression;

`exceptProtocol:aRegExp` excludes all methods in a protocol matching the regular expression;

`exceptClass:aClass selector:aSelector` excludes the method with name `aSelector` in class `aClass`.

Our DSL does not yet support declaring liable methods as a disjunction of multiple pragmas, since each method corresponds to an individual regularity. In order for the same conditions to be applicable to multiple, disjunctive sets of methods, the same contract method needs to be duplicated, with different pragmas indicating the liable methods.

4.4. Defining structural regularities

Having defined in the contract class, per structural regularity, an instance method defining the liable methods to which that regularity applies, we still need to complete the body of that method with the actual definition of the regularity to be checked. At the start of this section, we already illustrated how to define the FAMIX regularity that each method overriding the method `copyFrom:within:` should start with a super call to the same message.

4.4.1. Contract terms

In general, each method representing a structural regularity consists of one or more contract terms. Each contract term represents a single requirement that must be fulfilled by each liable method for the contract to hold.

[8]In Smalltalk, every method is annotated with a protocol. These protocols serve to group sets of related methods, for example all accessor methods.

Our DSL allows making a distinction between contract terms expressing a hard *requirement* on the source code structure, in order for the program to behave correctly, and mere *suggestions* for improvement, that are more stylistic in nature and could for example increase program comprehensibility.

In our DSL, contract terms are created by sending keyword messages to the pseudo-variable `contract`. In our example regularity above, we specified a hard requirement as `contract require: ... if: ...` Our *uContracts* DSL currently supports the following contract terms:

`require:aCondition` states that the condition `aCondition` *must* be satisfied by each liable method;

`suggest:aCondition` only *suggests* that condition `aCondition` be satisfied by each liable method;

`require:aCondition if:aCondition2` *requires* `aCondition` to be satisfied by each liable method for which `aCondition2` holds as well;

`suggest:aCondition if:aCondition2` *suggests* that `aCondition` be satisfied by the liable methods, but only if `aCondition2` holds.

4.4.2. Contract conditions

Within each contract term (regardless of whether it be required or suggested), the conditions of the contract need to be declared. Table 2 provides an overview of the different kinds of conditions supported by our *uContracts* DSL. It distinguishes three kinds of conditions: structural conditions, logic conditions and locality conditions.

The first group of conditions allows us to specify structural constraints on liable methods. For example, these conditions can be used to declare whether a method's source code should contain a particular message send, assign to a particular field, whether the method is overridden, and so on.

Of special interest is the `custom:` condition, which takes as argument a visitor that visits the AST of the liable method, and allows developers to express custom-made conditions. As mentioned earlier, the goal of our DSL is to support a large amount of regularities with as simple a language as possible. A limitation of this approach is therefore that not all regularities can be expressed. To alleviate this limitation, the `custom:` condition was added to our language to allow developers to express in a relativily straightforward manner many regularities that are not supported currently by our DSL. In Subsection 6.3.4 we will revisit the `custom:` condition and illustrate its use.

Structural conditions	
`assigns:aRegExp`	Does the method assign to a field whose name matches the reg. exp.?
`calls:aRegExp`	Does the method send a message matching the reg. exp.?
`references:aRegExp`	Does the method refer to a class matching the reg. exp.?
`returns:anExpression`	Does the method contain a return expression `anExpression`?
`doesSuperSend:aRegExp`	Does the method contain a `super` send of a message matching the reg. exp.?
`doesSelfSend:aRegExp`	Does the method contain a `self` send of a message matching the reg. exp.?
`inProtocol:aRegExp`	Does the method belong to a protocol matching the reg. exp.?
`isOverridden:aSelector`	Is the method named `aSelector` overridden?
`isOverridden`	Is the liable method overridden?
`isImplemented:aSel`	Is the method named `aSel` implemented by the class of the liable method?
`custom:aVisitor`	Does the AST of the liable method match the conditions expressed by `aVisitor`?
Logic conditions	
`and:aCond1 with:aCond2`	Do `aCond1` *and* `aCond2` hold for the method?
`or:aCond1 with:aCond2`	Does `aCond1` *or* `aCond2` hold for the method?
`not:aCond`	Does the condition `aCond` *not* hold for the method?
Locality conditions	
`beginsWith:aCond`	Does the first statement of the method satisfy the condition `aCond`?
`endsWith:aCond`	Does the last statement of the method satisfy the condition `aCond`?
`does:aCond after:aCond2`	Is `aCond` satisfied after a statement that satisfies `aCond2`?
`does:aCond before:aCond2`	Is `aCond` satisfied before a statement that satisfies `aCond2`?

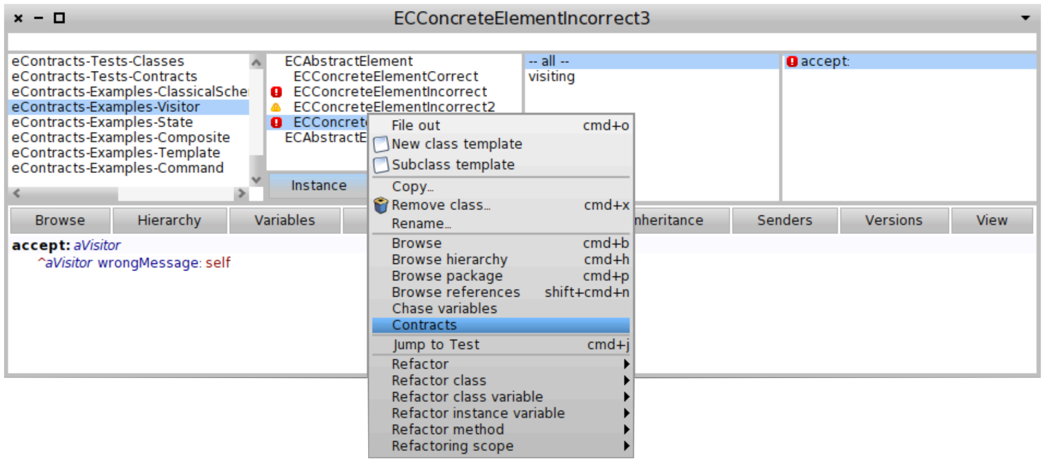
Table 2: Possible contract conditions provided by the *uContracts* DSL

The second and third group contain a number of higher-order conditions. These higher-order conditions allow for the logic combination of other conditions (and, or, not), and provide a means to constrain where a particular condition should be satisfied in the liable method's body. For example, the locality conditions allow us to express that a particular condition should be satisfied by the first or last statement of the method, or before/after a statement matching another condition.

5. Tool support

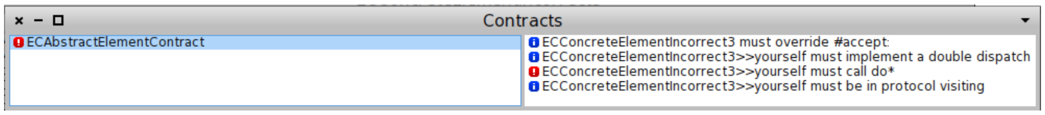
In order to get the cooperation needed for the industrial case study we conducted (see Section 6), and in the hope of reaching wider adoption of our tool, we implemented our *uContracts* DSL as an extension to the Pharo 1.4 Smalltalk development environment. In addition to implementing this domain-specific language within the Smalltalk language, we tightly integrated the DSL with the development tools of the surrounding Pharo environment and added some additional tools as well [9]. Figure 4 shows an example of this integration.

[9] *uContracts* can be downloaded at `http://www.squeaksource.com/eContracts.html`



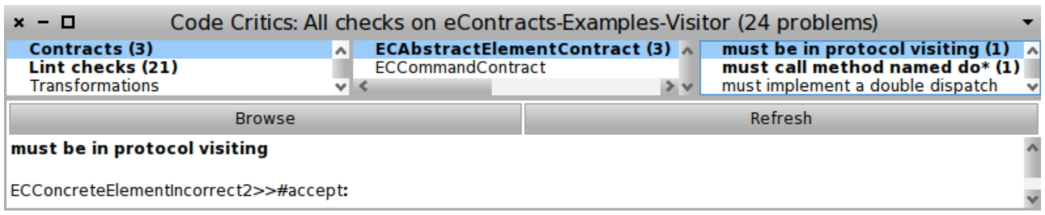
```
accept: aVisitor
    ^aVisitor wrongMessage: self
```

Figure 4: A screenshot of Pharo's class browser. Failure of only suggested terms is indicated in yellow; failure of at least one required term in red.



ECConcreteElementIncorrect3 must override #accept:
ECConcreteElementIncorrect3>>yourself must implement a double dispatch
ECConcreteElementIncorrect3>>yourself must call do*
ECConcreteElementIncorrect3>>yourself must be in protocol visiting

Figure 5: A screenshot of our contract browser in Pharo, which shows all terms of the chosen contract. A blue icon indicates a contract term (i.e. regularity) that is respected, while red or yellow icons indicate violated contract terms.


Code Critics: All checks on eContracts-Examples-Visitor (24 problems)
Contracts (3)
Lint checks (21)
Transformations
ECAbstractElementContract (3)
ECCommandContract
must be in protocol visiting (1)
must call method named do* (1)
must implement a double dispatch
Browse
Refresh

must be in protocol visiting

ECConcreteElementIncorrect2>>#accept:

Figure 6: A screenshot of the integration with Pharo's *Code Critics* tool. This Lint-like tool shows all contracts in the system. For each contract, it lists the conditions of that contract along with the violations of the conditions.

As soon as a method is compiled, all usage contracts applicable to this method are checked immediately by our *uContracts* tool. If the method violates any of the contracts in which it is involved, this will be indicated by either a yellow (signifying only violated suggestions) or red (signifying violated requirements) exclamation mark to the left of the method name (as is the case with the `accept:` method in Figure 4) in the class browser.

Classes that contain methods with contract breaches are highlighted in a similar way (also see Figure 4). Finally, via a context-sensitive menu, it is possible from within the class browser to find all contracts to which a given class or method is liable.

By checking structural regularities *immediately* during development, developers are warned of contract breaches early on. Our approach can thus be considered as a kind of live documentation: each time either a contract or a liable entity changes, the contract is rechecked and developers are informed immediately of contract breaches. These checks do not pose a significant overhead, as our language currently offers only structural conditions and all conditions are always local to a certain method, and can thus be verified by simply traversing the method's parse tree.

Via a context-sensitive menu in the class browser, a developer can get access to more detailed information regarding a breached contract. An example of such detailed information is presented in Figure 5. For each of the contracts that are applicable to the selected class, the tool indicates whether the contract is violated or not. By clicking on a contract, all violated regularities in that contract are shown. We use the blue colour to indicate that there is no violation, yellow for suggestions, and red to indicate required regularities that were not respected. The tools described above allow developers to assess which contracts are violated for a particular source-code entity, and why (i.e. which of the regularities of the contract were breached).

Finally, by means of an integration with Pharo's *Code Critics* tool (see Figure 6), we also provide support for browsing all violations of a particular contract, or of a group of contracts.

6. Checking Structural Regularities in an Industrial Application

To validate our approach, we applied it to an industrial case study.[10] The case study was a medium-sized (see Table 3), interactive web application that supports event planning and resource management and was implemented using Pharo Smalltalk and the Seaside web development framework. The application had been under continuous development for 4 years by 4 experienced Smalltalk developers. It was designed as a component framework (reusable

[10]For reasons of confidentiality, we cannot disclose the name of the case study nor of the company involved.

editors, reusable views, reusable resources, etc.) so that it could be extended in a controlled manner as the requirements of the clients evolved.

Number of packages:	45
Number of classes:	827
Number of methods:	11777
Number of lines of code:	94151

Table 3: Some size metrics about the industrial case study.

6.1. Case study setup

The main goal of the case study was to qualitatively assess the expressive power of our DSL. We wanted to investigate to what extent it was able to support structural regularities appearing in a realistic application.

For conducting the case study, we sat together with one of the main developers of the web application, to explain the concept of usage contracts for expressing and checking structural regularities. A discussion followed resulting in the compilation of a list of potential structural regularities that were deemed of interest by the developer. The developer picked these structural regularities as they corresponded to frequently occurring errors, framework constraints that could be violated easily and that could result in subtle bugs, or violations of conventions that would have a detrimental effect on the consistency or comprehensibility of the system.

The actual definition of these contracts was done by the authors of the paper and required less than 2 working days, but we did not assess how difficult it would be for a non-expert of the system. It would have been ideal if we would have been allowed to integrate the *uContracts* tool in the daily development environment and process of the developers, to be used by them. But given that it was a new tool of which we could not present any prior experience reports yet on the return on investment of using the tool, it was hard to persuade the developers to use the tool right away. Therefore, we set up an alternative experiment, where the authors of this paper expressed the list of 13 structural regularities (identified by a developer) as usage contracts in the *uContracts* DSL. Compliance of the original implementation with these usage contracts was then checked at two different points in time. In December, all identified contract breaches were reported back to the original developer of the system. Three months later, in March, the contracts were reverified to provide an assessment of the amount of new violations introduced between December and March. In the end, the results of this initial experiment turned out to be sufficiently convincing to actually persuade the developers to start using the tool themselves.

6.2. Overview of documented regularities

Name of regularity	**Description**	**Liable methods**	**Exceptions**	**Errors December**	**Errors March**
Usage contract for the model entities (214 liable classes)					
Object equality idiom	Equality is implemented using a double dispatch protocol	19	0	0	0
Marking dirty objects (§6.3.2)	State changes must mark model objects as dirty	333	5	7	2
Grouping initialization methods	Protocol naming convention	110	0	53	13
Initialization methods idiom	Requires super call	110	0	0	0
Grouping internal methods	Internal methods are put in a special protocol	193	65	11	2
Usage contract for the persistent entities (75 liable classes)					
Initialization via database (§6.3.3)	Requires super call	44	0	1	0
Custom object identity	No overrides of hash or equals	74	1	1	0
Usage contract for the interface code (598 liable classes)					
Rendering methods restriction	Use of internal methods restricted to a certain scope	7410	0	3	0
Certain methods should not be called directly (§6.3.1)	Certain methods should not be called directly from within interface code	7410	0	3	2
Call ordering within cascade (§6.3.4)	Certain messages need to be sent at the end of a method	531	0	0	0
Preferred framework construct	Prefer particular constructs of the framework over alternatives	801	0	2	0
Incorrect use of transaction mechanism	Code in transaction blocks cannot perform non-local returns	595	0	0	0
Immutable while rendering	Rendering methods should not assign to instance variables	801	0	0	1

Table 4: Overview of the documented structural regularities in the industrial case study.

Table 4 gives an overview of the 13 structural regularities that were documented. Most of them were either related to the particular framework

(Seaside) that was used to implement the web application, or were a consequence of particular architectural choices in the design of the application. For example, since the application was intended to be easily extended, part of the application was conceived as a component framework with reusable editors, reusable views, etc. The use of these components is regulated by several rules specific to the particular application, hence the need for a customizable tool/language such as *uContracts*, in which these rules can be codified and verified.

We divided the codified regularities in three categories, each of which was documented as a separate usage contract consisting of several structural regularities: those related to the *model* of the web application, those related to how *persistence* is handled in the application, and those related to how the *interface* is constructed via the Seaside web development framework. For each of these usage contracts, we show the number of liable classes; for each of the structural regularities we show the number of liable methods and exceptions.

6.3. Examples of documented regularities

We now take a closer look at four of these documented regularities, before analyzing and discussing the results of the case study in more detail.

6.3.1. Certain methods should not be called directly

Certain operations on particular business objects in the case study, that require some additional handling, should never be called directly but always be performed via a separate layer of manager objects. Given that the visibility of methods in Smalltalk is always public and cannot be modified, the developers of the case study introduced a naming convention (prefixing these methods with ‘private’) to indicate which methods should not be called directly via the interface, but should pass via a manager object. As calling these methods directly can result in that the object is put in an inconsistent state, it is imperative that no such calls occur.

We document this regularity by adding a contract method to the class implementing the usage contract for the interface code. As liable classes, this usage contract selects the classes belonging to all packages of the system, except those packages containing business object and database code. On the class implementing the contract, this is reflected by a class method named `interfaceCode` with as implementation:

```
interfaceCode
  <package:'App-*'>
  <exceptPackage:'App-Model*'>
  <exceptPackage:'App-Database*'>
```

The actual regularity is implemented by the method `noCallsToPrivate`:

```
noCallsToPrivate
  <selector:'*'>
  contract require:
    (condition not: (condition calls:'private*'))
```

This regularity expresses that, for *all* methods in the interface code (selectors matching the wildcard symbol `*`), it is required that the method does *not* contain any call to a method with name matching the pattern `private*`.

6.3.2. Marking dirty objects

Within the model of the web application, a model-view-controller mechanism is employed to identify which model entities have changed and should therefore be re-rendered in the interface. In order to ensure that the interface properly reflects the current state of the model, it is imperative that all locations in the model that actually change the state (i.e., that assign to a field), also mark the changed model object as dirty, using the method `markAsChanged:`.

To document this regularity we created a usage contract that is applicable to all classes representing domain objects. These liable classes of the contract are defined in terms of the hierarchy of domain objects in the system:

```
domainClasses
  <hierarchy:#AppDomainObject>
```

Inside this usage contract we document the structural regularity as follows:

```
dirtyFlag
  <selector:'*'>
  contract
    require: (condition calls: #markAsChanged:)
    if: (condition assigns: '*')
```

This regularity applies to all methods on model classes and states that, when the method performs an assignment to any field, it is required to call `markAsChanged:` as well.

6.3.3. Initialization via database

Objects in the model of the web application are initialized by means of a method `initializeWithDatabase:`. This method is responsible for registering the object with the database session and keeping track of internal bookkeeping. As all other initialization logic requires that the object is registered in the session *first*, it is important that all methods that override `initializeWithDatabase:` start with a super call; if not, the rest of the initialization of the object is potentially compromised. It is also suggested that this method be classified in the protocol `initialize-release`.

This structural regularity was readily documented by adding a method to the usage contract applicable to all persistent domain classes.

```
persistentDomainClasses
  <hierarchy:#AppPersistentDomainObject>
```

And the actual regularity itself was expressed as follows:

```
initializationOfDatabase
  <selector:#initializeWithDatabase:>
  contract
    require:
      (condition beginsWith:(condition doesSuperSend))
    if: (condition isOverridden).
  contract suggest:
     (condition methodInProtocol:'initialize-release')
```

This regularity applies to all implementors of `initializeWithDatabase:`. It actually consists of two separate contract terms: one that requires the super send constraint, and another one suggesting the protocol naming convention to be used.

6.3.4. Call ordering within cascade

The fourth regularity we discuss is related to the use of the Seaside web development framework. Within Seaside, web pages are rendered programmatically: the framework offers a canvas on which the application draws the interface. This is done by creating particular nodes (e.g., paragraphs, ordered lists, list items) that are each represented by a single object. The parameters of these nodes are set by sending messages to these objects (mostly using a message cascade); the contents of the node are set by sending the message `with:` to the node. The Seaside framework dictates that these calls to `with:`

within a cascade should always be the last call since all other messages sent to the node after this call are ignored by Seaside. This was a frequent source of bugs during development of the web application, which is why this rule was important to be documented and checked with our tool. The bugs are not hard to discover through testing, neither are they to correct, but they are annoying and it would be preferable if they could be avoided altogether with an automatic check as soon as a method gets compiled.

We documented this regularity as part of the usage contract that is applicable to all packages containing the user interface code (the same contract as in Section 6.3.1). Within this contract, we declared the regularity below:

```
withShouldBeTheLastMessageInACascade
  <selector:'render*'>
  contract
    require:
      (condition not:(
         condition
           custom: WithInCascadeVisitor
           description:'With: should be last'))
```

As liable methods, we select all methods within the interface code that are prefixed by 'render'. As our DSL has no native primitives yet to express constraints on the order of messages in a cascade, we implement a custom condition using a visitor called `WithInCascadeVisitor`. This visitor is responsible for matching the AST of the liable methods for any cascade nodes in which a `with:` is sent in any but the last position. In order for the contract to succeed, the visitor should *not* find any such match within the AST of the liable method.

The visitor class `WithInCascadeVisitor` subclasses from the class `CustomConditionVisitor` that is provided by our *uContracts* tool. To express the regularity above, we overrided a single method on `WithInCascadeVisitor` that visits all cascade nodes within the method:

```
acceptCascadeNode: aNode
  super acceptCascadeNode: aNode.
  (aNode messages allButLast
    anySatisfy: [:msg | msg selector = #with:])
     ifTrue: [self match: aNode]
```

The first line of this method performs a super call in order to ensure that the rest of the AST is traversed. The rest of the method checks if any of the messages in the cascade, except for the last one, send the message `with:`. If this is the case, the node is added to the matches of the visitor.

We can observe that, thanks to the use of the `CustomConditionVisitor`, this solution is only slightly more complex than if we would have used a native condition of the DSL.

6.4. Case study discussion

After having declared all 13 regularities in a way similar to the 4 regularities exemplified above, we checked compliance of the original source code with those regularities at two different occasions (December and March) and reported the results back to the original developer, who processed all violations marked by the tool together with us. We classified each reported violation either as an exception (i.e., an accepted deviation to the regularity, which we afterwards documented explicitly as an exception to the regularity, by using the pragmas provided by our DSL to exclude exceptional cases), or as actual errors (which the developer used as input to actually correct the source code).

Table 4 shows, for each regularity, the number of identified errors at the two different points in time. In December, 7 locations in the code were identified where the dirty flag was incorrectly omitted and which were bugs in the system. Similarly, the tool identified 3 methods that incorrectly called, from within the interface, methods tagged as 'private' and thus circumvented the object manager infrastructure. All other errors indicated either a violation of a naming convention, or of the intended design of the system. When verifying the (evolved) system again in March, we found that, with three months in between, although many of the previously detected errors had either been corrected or flagged as exceptional cases, a number of new violations to the documented regularities had been introduced in the system. Despite the fact that the developers were made aware in December that certain important regularities in their system do get violated, this did not prevent them from accidentally introducing new violations of these regularities, thus serving as an illustration of the need for tools like *uContracts*. In fact, the results of this initial industrial case study eventually persuaded the developers to actually integrate *uContracts* in their daily development environment, so that they could get early feedback on detected violations as soon as they appear.

The description of 11 of the 13 regularities which we documented consisted of only one or two simple contract terms. In spite of the limited scope of this case study, it supports our intuition that a very simple language suffices to document already a large number of useful structural regularities. Only in the case of the 'Call ordering within cascade' regularity discussed above, and of the 'Incorrect use of transaction mechanism' regularity which checks that code in transaction blocks cannot perform non-local returns (not shown above), was the use of the `custom:` condition required.

7. Discussion

Having explained our approach and supporting tool, and their validation on an industrial case study in the previous sections, in this section we now discuss their main strengths and shortcomings.

Usefulness. Our industrial case study illustrated the relevance and usefulness of a tool like *uContracts* when building a reusable web application in Smalltalk. It could be argued that the use of a language with a stronger or static type system would reduce the need for such a tool. Most of the regularities which we expressed in the Smalltalk case study, however, were not related to types, nor caused by the lack of a static type system. Also, our analysis of undocumented structural regularities in JHotDraw gives evidence that similar regularities do in fact exist in programs written in statically-typed languages like Java.

Practical expressivity. *uContracts* offers developers a simple and declarative language to document structural regularities concisely and readily. The constructs provided by our DSL were determined by studying literature related to checking structural regularities, and were cross-referenced with regularities discovered in JHotDraw. It is therefore not surprising that, when applied to the industrial case, our approach was able to express almost trivially 11 of the 13 structural regularities that were deemed of interest by the original developers. Note that it was not our goal to have a 'complete' language that supports *all* possible types of regularities, but rather to offer an easy-to-use language that can be adopted easily by developers and that readily supports frequently occurring regularities.

The main limitation of our DSL is that it is not very flexible for expressing regularities that are not directly supported by the primitive constructs of our DSL. For example, the properties that can currently be checked with *uContracts* are mainly structural: they describe structural properties of the source code entities of a system or higher-level architectural or design patterns that can be expressed in terms of such lower-level structural source code regularities. More fine-grained information like control-flow, call graphs, or relations between source code entities are not yet supported, thus limiting the expressive power of the language. Consider for example structural regularity #16 of JHotDraw, which expresses that after invoking a particular method on an object, the state of that object can no longer be changed. As we only reason over the structure of individual methods, we can only provide a very naive approximation of this regularity at best. Likewise, regularity #22, which states a naming convention between two groups of classes, cannot currently be expressed by our approach.

However, our language does provide a `custom:` condition to offer full access to the parse tree of a method. Given that this condition is expressed by means of a *visitor*, the resulting regularities are still declarative and just slightly more complex than regularities that are directly supported by our DSL, as was illustrated by the fourth regularity of the industrial case study. Hence, while our DSL is sufficiently expressive to cover many regularities as simply as possible, the `custom:` condition complements our other language constructs such that a large variety of realistic structural regularities can be expressed.

Finally, we realize that our DSL definition is not definitive and will most likely evolve over time to include new kinds of primitive conditions (e.g., for reasoning at a sub-statement level).

Adoption of the DSL. While we opted for implementing *uContracts* as an internal DSL, it is not a prerequisite for such a language to be useful. We do however feel that, by bringing the DSL as close as possible to the host language, it eases understanding of the declared regularities and encourages developer adoption. Internal DSLs tend to eliminate the symbolic barrier of the language while keeping the power and tooling of the base language [15].

In previous work, we explored tools such as IntensiVE [32, 33, 6] that provided more generic support for expressing source code regularities in terms of the external DSL SOUL [9]. We are thus well-placed to compare both kinds of approaches. We do not claim the *uContracts* approach to be "superior", since there are clearly pros and cons for each approach. However, frequent objections to conduct industrial trials on our previous tools were the language barrier, a lack of integration with the IDE, and a lack of support for immediate feedback. In fact, similar problems (lack of tooling, interference with the work-flow, inter-operation issues with the base language, and integration with other DSLs) have been identified by others as common obstacles for the adoption of external DSLs [16].

uContracts, being a DSL embedded in a *dynamic language*, avoids several of these issues by blending in perfectly with the development process and environment: with a shared syntax and IDE, 'on-the-fly' validation, and seamlessly integrated with the source code entities of the base language. In spite of that, as most other internal DSLs, *uContracts* is less expressive than its external counterparts [15] such as Semmle [44] and SOUL [9].

In general, *uContracts* was implemented as an internal DSL for two main reasons. Simplicity: the main goal of our DSL was to have a language that is simple to use by an "average" developer without having to learn a completely new language, yet powerful enough to support expressing a wide variety of regularities. Productivity: by keeping its syntax close to the Smalltalk base language, and by drastically limiting the number of available constructs, we provide a light-weight approach to check simple but frequently occurring regularities. In this sense, our approach is complementary to approaches such as SOUL and Semmle. To some extent we trade-off expressiveness for ease of use and ease of adoption.

Nevertheless, our tool still remains a proof of concept. Even though it was stable enough to use on an industrial case, it would require additional effort and validation to turn it into a tool that could be adopted on a wider industrial scale. In that sense it is, at this stage of the research, too early to provide statistically significant evidence backed up by user studies.

Integration with language and development environment. One of the main advantages of our tool is that it does not hinder developers by disrupting their mental and development process. The feedback provided by the tool is directly integrated with the development environment and does not require developers to switch between multiple tools. Furthermore, it does not enforce the documented regularities; instead, source-code entities that breach a contract are indicated as a warning. Although developers are made aware of violations at development time, they are not forced to immediately act upon these warnings, thereby disturbing their work flow.

In terms of computational overhead, the simplicity of our language results in that even the most complex of our contracts can be checked at development time in less than half a second and that therefore the developer is not slowed down by the tool. Especially since our regularities are local to individual methods (for this reason, we do not check relations *between* source-code entities), they only need to be re-checked upon addition of a new method, or when an existing method gets changed. Consequently, this allows our approach to be used in an incremental fashion, thereby minimizing the computational overhead of verifying all applicable contracts.

Explicit documentation. An important strength of our approach is that it bridges the gap between structural regularities and source code, by making regularities that were previously documented only implicitly (or not at all), documented explicitly and verifiably, as part of the source code. As we provide a tight integration with the surrounding development environment, developers become aware of violations of such regularities immediately during development and can take action accordingly. Therefore, our approach provides a lightweight means – akin to unit testing – to detect violations of structural regularities early on.

Support for other languages. As the *uContracts* tool is implemented as an embedded DSL for Smalltalk, it is inherently tightly coupled with this host language. This does not imply, however, that the ideas presented in this paper are restricted to Smalltalk only.

First, our approach uses the reflective capabilities of Smalltalk as a means to retrieve and validate the documented contracts. While this use of reflection eases the implementation of our tool, it is not a requirement for implementing a tool like ours. Although the validation of the contracts requires access to a detailed representation of the source code of the analysed system, such presentations are commonly available for a wide variety of languages (for example, the Eclipse DOM for Java).

Second, our approach is implemented as an internal DSL embedded within Smalltalk. Smalltalk is well-suited to host such embedded DSLs due to the simplicity and structure of its syntax. Similar features can be found in languages such as Ruby, that have also been used successfully as host language for internal DSLs [17]. While implementing an internal DSL for Java may not be as straightforward and elegant as for Smalltalk or Ruby, it should be possible to implement a similar language on top of Java as well.

It is difficult to speculate what *uContracts* would look like for another programming language, without having explicitly tried to actually implement it in that language. Any implementation of uContracts within, e.g., Java or C# would look significantly different (both from the point of view of the user as of the implementation) as one of its goals is to remain as close as possible to the base language. Implementation challenges include getting access to a navigable representation of programs in that language, and embedding the syntax within that language. These challenges have already been discussed in literature [15, 16].

Several prototypes of *uContracts* for Ruby have been developed (independently) by some of our students. Some students simply ported our Smalltalk solution, which resulted in DSLs less expressive or with a more convoluted design than alternative implementations. The alternative implementations were rather different from our Smalltalk solution, in spite of the fact that Ruby is also a dynamically typed language with powerful reflective capabilities akin to Smalltalk. In general, the solutions of students who already had a deep knowledge of Ruby, and whose prototypes kept a syntax close to the base language, were judged as the better prototypes. This experience resulted in two observations. First, to implement something like *uContracts* for another language, having a deep and intimate knowledge of that language and its reflective facilities is an advantage (which may make it less trivial to quickly implement a prototype for other languages, especially for statically typed languages which often have a more restricted or more intricate reflective API). Second, much care should be taken to devise a syntax that is as close as possible to the syntax and spirit of the host language.

Correctness of uContracts. An incorrectly defined contract might accidentally detect entities that do not actually breach regularities (false positives), might fail to detect cases that require enforcement (false negatives), or might miss needed structural constraints (incomplete specification). As soon as a contract is defined, the developer should check which entities are covered by the contract, and fix them if needed. Although incorrect contracts may become obvious by reporting too many contract breaches or breaches that shouldn't occur, there is no guarantee that contracts are defined correctly. It remains the responsibility of the developer who defined a contract to make sure that its definition is as correct and complete as possible, or to fix the contract when such situations are discovered. Given that the contract language is relatively easy to use and remains close to the developer's base language, checking and correcting a contract should not be too cumbersome and, hopefully, should not happen too often. In any case, as with testing frameworks, reported contract breaches should always be interpreted with care.

Suitability of uContracts to detect bugs. Since the focus of our approach is not on functionality but rather on style and structural issues, detected contract breaches do not necessarily correspond to bugs, which does not imply that such contract breaches are less important. In some cases, however, the contracts may point at potential bugs which can be more difficult to detect with traditional testing, because the cause of the bug is not in the algorithm or in the data but in the way source code entities are used.

The *uContracts* reported on in this paper were obtained by asking the developers which were the main sources of errors due to inappropriate reuse. Our industrial validation experiment verified these rules with respect to the existing code base. Detected contract breaches indeed revealed a number of bugs in the code. When the tool was used later during actual development, many of the bugs that were detected by our tool where avoided (solved immediately upon detection) and therefore never even made it to the repository, making it hard to provide quantitative assessments in hindsight on the number and kind of bugs detected and solved during actual development.

Impact of source code evolution on the contracts. As for unit tests, any modification or restructuring of the source code (even behaviour-preserving refactorings) may affect some of the structural regularities. Contracts affected by such code evolution need to be updated. The tool provides no automated support for that, nor do we intend to add such support at this stage. Nevertheless, restructurings can and should be seen as interesting opportunities to add new contracts that encode the new structure. Whenever the code structure is being improved or updated, it is a good idea to encode the new desired structure as a set of contracts to be respected, thus adding guarantees that the new structure is and remains respected by programmers.

8. Threats to validity

In this paper we introduced *uContracts*, an internal DSL tightly integrated with the Smalltalk development environment, serving as a lightweight specification language for documenting and verifying structural source-code regularities. We conducted an initial validation by applying the DSL and its corresponding tool to an industrial case study. The fact that most of the contracts specified for the industrial case required less than 10 lines of DSL code, that it took us less than two working days in total to define them all, and that most of the regularities found in typical object-oriented applications such as JHotdraw too can be expressed straightforwardly using our DSL, indicates that the primitives it provides are sufficient to express most common structural constraints. Besides, the fact that the tool is still in use at the company to verify the already encoded regularities of their product, indicates that implementing it as an internal DSL was an appropriate choice.

However, it should be kept in mind that our current DSL is still a proof of concept. Given that is was evaluated on a single industrial case only, and that the developers exposed to it were expert Smalltalk programmers, accompanied by the authors of this paper, we cannot claim yet that our findings on the effort required to use the DSL are generalizable. The nature of the application analyzed may introduce some bias too, because it was designed as a framework that is easy to extend in a controlled manner (i.e., with clear reusable components that were designed to be extended). Furthermore, given that the company only has a single product, it was hard to assess the impact of the tool on the developers' productivity, because we had no no baseline against which to compare the potential benefical impact of using *uContracts*.

Whereas our validation does seem to indicate a potential relevance for such a tool, we are currently exploring if the idea is worth evolving into an industrial-strength prototype which would allow us to conduct full user studies and provide us more conclusive evidence regarding the usability, expressivity and relevance of the approach.

9. Related work

We were certainly not the first to propose an approach for documenting and checking (structural or other) regularities in source code.

Contract-like approaches. Within the software engineering community, the contract analogy has been widely used, the most well-known example probably being **Eiffel**'s **design-by-contract** [35]. Contracts in Eiffel allow to describe obligations for the client (consumer) method to respect (precondition), so that the supplier (producer) does not need to verify its validity, as well as to describe obligations of the supplier (postcondition), so that the client can be oblivious to the internal details of the method. Eiffel's contracts have a different focus than *uContracts*, however. While Eiffel's contracts focus on verifying behavioral assumptions, *uContracts* attempt to codify and validate *structural* assumptions.

Reuse contracts [41] were initially introduced as a solution to the *fragile base class problem*, i.e. the problem of evolving an existing class (producer) which other classes (consumers) already reuse through inheritance. Such an evolution can cause subtle behavioral conflicts when the change invalidates certain assumptions the subclasses made about their base class. Reuse contracts make such assumptions explicit by specifying how the subclass reuses its base class in terms of a set of primitive reuse modifiers such as 'extension', 'concretization' and 'refinement'. Later variants of reuse contracts extrapolated the idea to incorporate collaborations between different classes as well.

Component contracts [11] offer a framework to design object-oriented components and discover composition errors. Using Prolog, component contracts describe and verify structural and behavioral constraints related to consistency, integrity, and evolution of components. This approach differs from most other approaches discussed here as it is conceived for a model-driven engineering process, so the contracts are supposed to be used for instantiation and integration of components.

Verifying structural regularities. Various tools exist that aim at validating the structural regularities governing a particular domain or framework. Examples of such tools are **Lint** [24] (for C); **PREfast** [39] (for C/C++); **JLint** [25], **ESC/Java** [14], **CheckStyle** [1], **FindBugs** [22], or **PMD** [2] (for Java); and Microsoft's **FxCop** (for C#). While such tools identify a wide variety of commonly-occurring bugs, they tend to be limited to a predefined set of rules and typically provide only limited support for validating high-level *application-specific* regularities. Only **FindBugs** [22] and **CheckStyle** [1] provide support for application-specific regularities by means of a visitor, which is similar to our `custom:` condition. Nevertheless, these tools aim at detecting inconsistent, unusual or deviant code which does not necessarily reflect causes of semantic bugs caused by incorrect usage of classes or methods, which is the purpose of *uContracts*.

Within literature we can also identify several approaches that enable developers to document and validate the structural regularities governing their systems. As early as 1996, Minsky [36, 37] proposed his formalism of **law-governed regularities** to allow the explicit and formal declaration of certain regularities in object-oriented software systems. These declarations form the architectural constraints or 'laws' that the system should obey and could be enforced by the development environment. Hakala et al. [19] developed FRED, a tool to generate a task list to instantiate a Java framework (**FRamework EDitor**) based on the concept of **specialization patterns**. Specialization patterns are defined in terms of roles (played by source code entities) and contracts (commitments among the roles), each role participating in a pre-established number of contracts. Roles can have structural properties, called constraints. Specialization patterns aim at describing how to initialize the hotspots of a framework. In this sense contracts are production rules for a role, that can be optional, mandatory, done or undone. Specialization patterns are defined in a graph to specify roles, contracts, and multiplicity which can be matched to pieces of code to evaluate the completion of the pattern. But FRED does not check structural constraints at the level of method bodies.

Oliveira et al. [38] present a framework to instantiate design models by declaring constraints. They proposed the **Reuse Description Language** (RDL) to describe framework extension points and the way in which they should be extended. In this language, they define the invariants that should be respected when instantiating a framework. The invariants express the contextual conditions in which certain structural properties should be true (much like describing a feature diagram in a structured way). Apart from basic logic operators (implies, and, or), the language also defines constructs for assignment, mutual exclusion, and parallel composition. An important disadvantage of RDL, however, is the poor support for the analysis which is done using XML/XSLT.

The **Dependency Constraint Language** DCL [42] was developed as a declarative, statically checked language to describe structural constraints between modules. Although DCL supports the analysis of a diverse set of dependencies (access, declare, handle, create, extend, implement, derive throw and use annotation), the definition of modules (sets of classes identified by pattern matching on their name) limits its expressiveness.

Filho et al. [40] propose using aspects to document and validate software regularities in an industrial MDE application. They provide a DSL developed in AspectJ to enforce those constraints that occurred frequently in their case study. The main limitation of their approach is the lack of expressiveness offered by the aspect-oriented programming language used (insufficient support for logic operators and advanced regular expressions).

Logic-based approaches. There exist a significant number of approaches that rely on logic programming as a means to encode structural regularities. Examples of such approaches are **SOUL** [9], **ASTLOG** [8], **JQuery** [23], **CodeQuest** [18], and **Semmle** [44]. These approaches offer developers a broad range of constructs to query their source code. While their use is not limited to verifying structural regularities only, they can be used to this end, by mere evaluation of the logic program that describes a regularity. Below we discuss a number of more dedicated approaches that make use of such logic query languages to provide support for structural regularities.

Hou and Hoover [20] proposed a **framework constraint and specification language** (**FCL**) for defining framework-specific typing rules. Although FCL is based on first-order predicate and set theory, it aims to be a language as close as possible to a programming language. The idea behind FCL is to include the constraints as part of the framework, and to validate them using static analysis. The language includes a wide variety of constructs ranging from control flow to composition of constraints. However, their approach was limited to framework instantiation, disregarding usage patterns and design knowledge. Also, FCL originally did not have an automatic checker of constraints, even though a later extension of their tool (**SCL** – Structural Constraint Language) was able to check encoded constraints [21]. SCL is however incapable of pointing out the reason for a rule to fail.

Mens et al. [32, 33, 6] developed the model and tool-suite of **intensional views** (IntensiVE) to help developers with documenting structural source-code regularities, verifying them and offering fine-grained feedback when and where the code does not satisfy those regularities. The IntensiVE tool-suite, which builds upon the underlying Prolog-like logic metaprogramming language SOUL [9], is tightly integrated with the Smalltalk host language, and was purposefully designed as a non-coercive set of tools that could be used to document and check structural regularities in source code. As explained in the introduction, the main difference with IntensiVE is that the approach described in the current paper does *not* rely on an underlying logic programming language. Instead, it offers an *internal* DSL which is much closer to the base programming language, thus avoiding the adoption barrier from which the IntensiVE approach seemed to suffer. Also, instead of offering the full expressiveness of a full logic programming language (which sometimes gave rise to heavy computations due to search space explosion), the current DSL, though more limited in expressiveness, was carefully crafted to be able to provide an immediate response to developers when used in a unit testing like way.

Eichberg et al. [12] proposed a DSL embedded in Datalog to validate dependency constraints across sets of source code entities. These sets of entities are declared as comments with a special prefix. Again, as for IntensiVE, while the use of a full-fledged program query language results in that this approach is more expressive than *uContracts*, the strength of our current approach lies in the fact that we can efficiently support a large amount of different kinds of regularities with only a very basic set of language constructs that remain close to the programming language in which they are embedded.

Several of the approaches discussed above have the disadvantage that they document the structural constraints separately from the code [6, 11, 13, 38]. Even if this documentation is explicit and verifiable, it still requires the developer to explicitly document the constraints and regularities in a separate model outside the code and to explicitly check conformance of the model with the source code. Consequently, programmers have to leave the comfort zone of their programming language, thereby disrupting their development process and lowering adoption of such tools.

As opposed to our current approach, for efficiency reasons many of the approaches above do not provide feedback to the software developer *immediately* upon a change to the source code. IntensiVE [32, 6], for example, typically checks the structural regularities just before committing a new version of the source code, so that problems can be fixed before the commit. Eichberg et al. [12] integrated their logic queries to the IDE by adding the validation to the build process of Eclipse. Terra and Valente [42] also validate their dependencies constraints with nightly builds. This validation requires several consecutive processes (each one dependent on the previous one) which include extraction of dependencies, parsing of constraints, and validation of constraints.

10. Conclusion and future work

In this paper we addressed the problem of documenting and validating structural regularities that describe how particular source-code entities should be reused or extended. Based on an analysis of the domain of structural regularities, we proposed *uContracts*, a simple domain-specific language integrated with the Smalltalk programming language and Pharo development environment. *uContracts* supports the declaration of so-called 'usage contracts' that allow developers to express structural regularities to be respected by the source code.

Although our DSL offers a limited vocabulary, this vocabulary is sufficiently expressive to describe a wide range of such regularities. Furthermore, by means of the `custom:` construct, regularities that cannot be expressed directly using our DSL can still be documented by means of a parse tree traversal as exemplified in our case study.

Due to the tight integration with the development environment, these structural regularities are checked whenever a developer changes a source-code entity. As a result, developers are informed *immediately* of potential breaches of the declared usage contracts. To validate our approach, we applied it to a medium-sized industrial application for which a set of regularities was identified by one of its developers. This validation illustrated that, given the fact that we offer a restricted language, we were able to express easily relevant regularities and detect interesting violations in the source code.

This paper also presented a first validation of our approach and language, focussing on whether we were able to express a variety of common regularities in an industrial case study. For future work, we plan to conduct more empirical studies in which we investigate the ease of use of the approach in real-life development settings.

The language constructs selected for this paper were based on a literature study and on an analysis of regularities in the JHotDraw system. However, we also intend to further extend our language and the tool suite that accompanies it to support other types of regularities. These additional regularities will be derived from a larger scale study of existing software systems to identify the most common kinds of regularities.

Finally, as future work we will also build a similar DSL and set of supporting tools for other programming languages than Smalltalk, such as Java and Ruby. In fact, a prototype for the Ruby language is currently being implemented.

11. Acknowledgements

The authors would like to thank Tom Van Cutsem for his comments on an earlier draft of this paper. During the writing of this article Angela Lozano was funded as a post-doc researcher on a Belgian FNRS-FRFC project. Andy Kellens was funded by the *Stadium* SBO project sponsored by the “Flemish agency for Innovation by Science and Technology” (IWT Vlaanderen). Kim Mens is professor in computer science at the Louvain School of Engineering of the Université catholique de Louvain (UCL) in Belgium.

[1] CheckStyle, August 2014. `http://checkstyle.sourceforge.net`.

[2] PMD, August 2014. `http://pmd.sourceforge.net`.

[3] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the European Software Engineering Conference/International Symposium on Foundations of Software Engineering (ESEC/FSE 2007)*, pages 25–34. ACM, 2007.

[4] G. Arévalo, S. Ducasse, S. Gordillo, and O. Nierstrasz. Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. *Information and Software Technology*, 52:1167–1187, November 2010.

[5] K. Bennett and V. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE 2000, pages 73–87, New York, NY, USA, 2000. ACM.

[6] J. Brichau, A. Kellens, S. Castro, and T. D'Hondt. Enforcing structural regularities in software using IntensiVE. *Science of Computer Programming: Experimental Software and Toolkits (EST 3)*, 75(4):232–246, April 2010.

[7] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 163–173, New York, NY, USA, 2007. ACM.

[8] R. F. Crew. ASTLOG: a language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages (DSL 1997)*, pages 18–30, Berkeley, CA, USA, 1997. USENIX Association.

[9] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *Proceedings of the International Conference on Principles and Practices of Programming in Java (PPPJ 2011)*, pages 71–80, New York, NY, USA, 2011. ACM.

[10] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[11] J. Dong, P. S. C. Alencar, and D. D. Cowan. Automating the analysis of design component contracts: Research articles. *Software Practice and Experience*, 36:27–71, January 2006.

[12] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the International Conference on Software Engineering (ICSE 2008)*, pages 391–400, New York, NY, USA, 2008. ACM.

[13] M. Feilkas, D. Ratiu, and E. Jürgens. The loss of architectural knowledge during system evolution: An industrial case study. In *Proceedings of the International Conference on Program Comprehension (ICPC 2009)*, pages 188–197. IEEE, 2009.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002)*, pages 234–245, New York, NY, USA, 2002. ACM.

[15] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. `http://martinfowler.com/articles/languageWorkbench.html#ExternalDsl`.

[16] J. Gray, K. Fisher, C. Consel, G. Karsai, M. Mernik, and J.-P. Tolvanen. DSLs: the good, the bad, and the ugly. In *Companion to the Conference on Object-oriented programming systems languages and applications (OOPSLA 2008)*, pages 791–794, New York, NY, USA, 2008. ACM.

[17] S. Günther and T. Cleenewerck. Design principles for internal domain-specific languages: A pattern catalog illustrated by Ruby. In *Proceedings on the Conference on Pattern Languages of Programs (PLoP 2010)*, pages 3:1–3:35, New York, NY, USA, 2010. ACM.

[18] E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest: Scalable source code queries with Datalog. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2006)*, pages 2–27, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] M. Hakala, J. Hautamaki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Annotating reusable software architectures with specialization patterns. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, pages 171–180, Washington, DC, USA, August 2001. IEEE.

[20] D. Hou and H. J. Hoover. Towards specifying constraints for object-oriented frameworks. *Information Systems Frontiers*, 4:393–407, December 2002.

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

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

[23] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 178–187, New York, NY, USA, 2003. ACM.

[24] 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.

[25] K. Knizhnik and C. Artho. Jlint: Find bugs in java programs. `http://jlint.sourceforge.net/`.

[26] Z. Li and Y. Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the European Software Engineering Conference/International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, pages 306–315. ACM, 2005.

[27] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *Proceedings of the European Software Engineering Conference/International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, pages 296–305. ACM, 2005.

[28] A. Lozano, A. Kellens, K. Mens, and G. Arevalo. Mining source code for structural regularities. In *Proceedings of the Working Conference on Reverse Engineering (WCRE 2010)*, pages 22–31. IEEE, 2010.

[29] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 48–61. ACM, 2005.

[30] T. Matsumura, A. Monden, and K.-i. Matsumoto. The detection of faulty code violating implicit coding rules. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2002)*, pages 173–182, Washington, DC, USA, 2002. IEEE.

[31] M. Mattsson. *Evolution and Composition of Object-Oriented Frameworks*. PhD thesis, University of Karlskrona/Ronneby, 2000.

[32] K. Mens and A. Kellens. Towards a framework for testing structural source-code regularities. In *Proceedings of the International Conference on Software Maintenance (ICSM 2005)*, pages 679–682. IEEE Computer Society, 2005.

[33] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views: A case study. *Elsevier Journal on Computer Languages, Systems Structures*, 32(2–3):140–156, July-October 2006. Special Issue: Smalltalk.

[34] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Elsevier Journal on Expert Systems with Applications*, 23(4):405–431, 2002.

[35] B. Meyer. Lessons from the design of the Eiffel libraries. *Communications of the ACM*, 33:68–88, September 1990.

[36] N. H. Minsky. Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of Object Systems (TAPOS)*, 2(1), 1996.

[37] N. H. Minsky. Law-governed regularities in object systems; part 2: A concrete implementation. *Theory and Practice of Object Systems (TAPOS)*, 3(2), 1997.

[38] T. C. Oliveira, P. S. C. Alencar, I. M. Filho, C. J. P. de Lucena, and D. D. Cowan. Software process representation and analysis for framework instantiation. *IEEE Transactions on Software Engineering*, 30:145–159, 2004.

[39] S. Podobry. Using PREfast for static code analysis, March 2011. `http://www.codeproject.com/Articles/167588/Using-PREfast-for-Static-Code-Analysis`.

[40] R. S. Silva Filho, F. Bronsard, and W. M. Hasling. Experiences documenting and preserving software constraints using aspects. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD 2011)*, pages 7–18. ACM, 2011.

[41] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: managing the evolution of reusable assets. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA 1996)*, pages 268–285. ACM, 1996.

[42] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Software Practice and Experience*, 39(12):1073–1094, Aug. 2009.

[43] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the International Conference on Automated Software Engineering (ASE 2009)*, pages 283–294. IEEE Computer Society, 2009.

[44] M. Verbaere, E. Hajiyev, and O. de Moor. Improve software quality with SemmleCode: an Eclipse plugin for semantic code search. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 880–881. ACM, 2007.

[45] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the European Software Engineering Conference/International Symposium on Foundations of Software Engineering (ESEC/FSE 2007)*, pages 35–44. ACM, 2007.

[46] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the*

International Conference on Software Engineering (ICSE 2006), pages 282–291. ACM, 2006.

[47] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2009)*, pages 318–343. Springer-Verlag, 2009.