Generation and composition of corrective actions to code design problems with Heal

Sergio Castro[1], Andy Kellens[2], Coen De Roover[2], Kim Mens[1]

Abstract

Maintaining consistency between design and implementation is a fundamental issue in software evolution. Although a certain number of existing tools and techniques provide support for correcting inconsistencies when found, they target generally-applicable design rules. Unfortunately, in addition to these common design rules, any software system has an abundance of custom design rules that are very specific to the system at hand. For such custom design rules, verification and correction are often left to the developer or require a considerable investment in the implementation of specific verification and correction metaprograms.

HEAL is a framework that alleviates the problem of diagnosing and correcting violations of custom design rules in code. It supports the implementation and automatic inference of corrective actions (i.e. a program transformation) that may solve a detected design inconsistency. Using an abductive logic reasoner, HEAL infers these corrective actions from the design rules themselves and the code over which these rules are verified.

In a previous work we show how simple solutions are generated from a repository of corrective actions to low level structural problems. In this work, we give one additional step showing that multiple solutions affecting the same source code element can be composed, presenting to the programmer the complete vision of how the source code would look like after executing a set of corrective actions.

Keywords: inconsistency management, abductive reasoning, logic meta programming

1. Introduction

Software developers need mechanisms to ensure that the source code they create and evolve conforms to *design rules*, such as the consistent application of *coding conventions* [2], *idioms* [11], *design patterns* [16] and *design regularities* [25]). The respect of these design rules is important not only for keeping the source code easy to understand and maintain [14], but is also required to preserve the correct behavior of a system (e.g., certain frameworks, such as Hibernate, require the source code to follow specific conventions to function correctly [1]).

Many development tools provide support for documenting and verifying design rules (e.g. IntensiVE [22, 19, 6], NDepend [31], Semmle [30], CheckStyle [9], Lint [18], FindBugs [10], Ptidej [17]). Some of these tools, such as Code Critics [3] and Eclipse's quick fixes [26] suggest automated corrective actions to help a developer resolve detected inconsistencies. None of these tools, however, provide sufficient support for detecting *and* resolving *user-defined* design rules in the implementation of a software system: the implementation of such user-defined design rules almost always requires developers to resort to ad-hoc techniques that involve writing metaprograms that walk an abstract syntax tree and, except for transformation techniques, no support exists for building corrective actions to correct these rules.

To fill this gap, we designed and implemented HEAL, a framework that extends IntensiVE (a tool suite for the documentation and verification of structural design rules) with support for diagnosis and semi-automated correction

Email addresses: Sergio.Castro@uclouvain.be (Sergio Castro), akellens@vub.ac.be (Andy Kellens), cderoove@vub.ac.be (Coen De Roover), Kim.Mens@uclouvain.be (Kim Mens)

[1]Université catholique de Louvain, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium

[2]Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium

of user-defined design-rule inconsistencies. User-defined design rules can be implemented in IntensiVE with the logic meta programming language SOUL [32] that can quantify over a program's source code. Developers can implement design rules by stating logic conditions that source code entities must adhere to. HEAL interprets these logic conditions using abductive logic reasoning [28, 15] to hypothesize about the possible causes for any design rule violation identified by IntensiVE. HEAL returns visual feedback of those causes to the developer and provides specific support for defining appropriate corrective actions making the source code adhere to a design rule. Our technique assumes the existence of a library of basic corrective actions to low-level design-rule conditions. For example, a typical solution to restore the consistency of a method with a wrong statement, could be deleting the faulty statement.

In a previous work we show how simple solutions can be generated from a repository of corrective actions to low level structural problems. We did not support at that time the multiple application of corrective actions over the same source code element, in such a way that the final result is consistent with all the applied corrections. In this paper we show our preliminary work in this direction. We describe in these initial experiments how multiple solutions affecting the same source code element can be composed, presenting to the programmer the complete vision of how a source code element with multiple affectations during the inference process would look like after executing a set of corrective actions.

Our work is structured as follows:

Section 2 discusses the IntensiVE tool suite and its logic meta programming foundations. Some examples show how developers can document and verify system's design rules. Section 3 subsequently demonstrates how associating corrective actions with the hypothesized causes of an inconsistency enables the semi-automated correction of design rule violations. Section 4 illustrates the complete diagnosis and correction process supported by our approach on our examples.We give an overview of our future work in Section 5 and in section 6 we show our conclusions.

2. Documenting and Verifying Design Rules using IntensiVE

IntensiVE [21, 19, 5] is a tool suite for documenting structural design rules and verifying their validity with respect to source code. This tool suite has been applied to document a wide range of such design rules, such as coding conventions, implementation idioms, design pattern instances, architectural dependencies and so on.

At the core of IntensiVE lies the concept of an *intensional view*. Such a view is a set of source code artifacts (such as classes, methods, fields) that are conceptually related (for example, all *visit methods* in the instance of a Visitor design pattern). Key to IntensiVE is that this set of artifacts is not defined by explicitly enumerating all the elements, but rather by means of an intension, i.e. an executable logic query that, upon evaluation, yields the artifacts belonging to the intensional view. To this end, IntensiVE uses the Smalltalk Open Unification Language (SOUL) [32] which is a Prolog-like logic programming language with specialized features for reasoning about the source code of Smalltalk [23, 4], Java [4, 12], C [13] and Cobol [20] programs.

To document structural design rules, relationships are expressed between these intensional views. IntensiVE supports three kinds of relationships, namely *multiple alternative views*, *unary relationships* and *binary relationships*.

As illustrative examples we have chosen the verification of custom design rules that should be respected by an existent web development framework: *Seaside* [27]. Since the design rules in this case study can be expressed with unary relationships, we will focus on this kind of rule.

Unary relationships are declared over a single intensional view and are used to express a number of conditions that need to be respected by the elements belonging to this intensional view. These relationships are of the form: $Qx \in V : conditions(x)$ where Q is a logic quantifier (e.g. $\forall, \exists, \nexists$), V is an intensional view and *conditions* is logic query expressed using SOUL.

In the rest of this section we show how some implementation design rules specific to Seaside can be documented with IntensiVE.

2.1. Example 1. No method should send `renderContentOn:`

In Seaside no method should invoke the message `renderContentOn:`. Instead, for rendering contents the method `renderOn:` should be invoked.

An intensional view named *WAComponents methods* is created to document this design rule. It is defined by means of the following SOUL query:

```
if classInheritsFrom(?class,[WAComponent]),
        methodInClass(?method,?component.class)
```

The actual design rule is documented by imposing a unary relationship over this view. This relationship is defined as follows:

```
∀ ?method ∈ WAComponents methods:
        not(methodSendsSelector(?method,{renderContentOn:}))
```

This unary relationship expresses that for all methods that are part of the *WAComponents methods* intensional view, the method should not send the selector `renderContentOn:`. To this end, the condition of this relationship uses the predicate `methodSendsSelector/2`. This predicate is defined in listing 1.

```
1 methodSendsSelector(?method, ?selector) if
2   statementOfMethod(?statement,?method),
3   ?messagesend isStChildOf: ?statement,
4   ?messagesend equals: RBMessageNode(?,?selector,?)
```

Listing 1: Definition of *methodSendsSelector*

This predicate receives as parameters a method and a selector. It verifies if the selector is used in the parse tree of the method.

2.2. *Example 2. Certain methods should contain a super call*

In Seaside, certain methods called *initialize* in classes inheriting from `WAComponent` should contain a `super` call as the first statement of their implementation.

An intensional view named *WAComponents initializers* is created to document this design rule. It is defined by means of the following SOUL query:

```
if classInheritsFrom(?class,[WAComponent]),
        methodWithNameInClass(?method,initialize,?class)
```

This query will gather all `initialize` methods in classes inheriting from `WAComponent`.

The design rule is documented by imposing a unary relationship over the intensional view *WAComponents initializers*. This relationship is defined as:

```
∀ ?method ∈ WAComponents initializers:
        methodBeginsWithSuperCall(?method)
```

This unary relationship expresses that for all methods that are part of the *WAComponents initializers* intensional view, the first statement of the method should be a `super` call. To this end, the condition of this relationship uses the predicate `methodBeginsWithSuperCall/1`. This predicate is defined in listing 2.

```
1 methodBeginsWithSuperCall(?method) if
2   methodWithName(?method,?name),
3   argumentsOfMethod(?args,?method),
4   methodWithFirstStatement(?method,
5     RBMessageNode(RBVariableNode(super),?name,?args))
```

Listing 2: Definition of *methodBeginsWithSuperCall*

This predicate receives as parameter the method that should begin with a `super` call. After requesting the name and arguments of this method (lines 2 and 3), It verifies if the first statement of the method is a message sent (`RBMessageNode`, line 5) to the pseudo-variable `super` having the same name and parameters than the receiving method.

The definition of the predicate `methodWithFirstStatement/2` is shown in listing 3.

```
1 methodWithFirstStatement(?method,?statement) if
2   statementsOfMethod(<?statement|?otherStatements>,?method)
```

Listing 3: Definition of *methodWithFirstStatement*

3. Diagnosing and Correcting Violations of Design Rules

In the previous section, we discussed how design rules can be specified and verified in IntensiVE by means of relations between intensionally-defined (i.e. by means of a logic query) sets of source code elements. This section describes how abductive logic reasoning allows to diagnose violations of those design rules and how corrective actions (i.e. source code transformations) enable the semi-automatic correction of such inconsistencies.

3.1. Diagnosing Violations with Abductive Inference

Reiter [29] defines diagnosis as the search of the components of a system that —under the assumption that these components are faulty— will explain inconsistencies between a description of this system and certain observations. Following Reiter's definition, we assume that violations of design rules are caused by faults in the system's components. At this time, we do not consider the possibility that the actual documentation of these design rules is incorrect.

The IntensiVE tool suite discussed in Section 2 documents design rules as relations between sets of source code elements. The logic meta programming foundations of this suite (i.e. it relies on logic queries that quantify over the program's code to define those sets and relations) implies that both the program to verify and its consistency rules, can be regarded as a logic theory. In logic, the process of hypothesizing causes for *abnormal observations* is referred to as abductive reasoning [28]. Given that we can consider abnormal observations as inconsistencies, HEAL makes use of a logic abductive reasoner for finding causes to design problems in code. Such abductive reasoner, in addition to generate answers to a query using *SLDNF* resolution [15], will attempt to find hypothetic updates to the logic theory, that will change the result of a query. Intuitively, abductive inference is typically accomplished with the analysis of a logic proof tree (i.e., a tree representation for the logic resolution process of a query). Examining this tree, it is possible to suggest the right updates to the theory that would change the result of a query according to a particular intention.

For a detailed explanation of abductive inference, and how it is implemented in HEAL we refer to our previous work [8].

3.1.1. Declaring Reification Predicates as Abducibles

IntensiVE reports design rule violations as a set of source code elements for which the relations do not hold. To diagnose a violation, it therefore suffices to re-evaluate the logic queries that define the violated relations —this time using abductive logic resolution with the queries' free variables bound to the source code elements that correspond to the violation.

SOUL offers several predicate libraries for reasoning about a system's source code. Each library provides reification predicates that reify the basic relations between the nodes in an abstract syntax tree representation of the program (e.g., `methodWithNameInClass/3`). In addition, each library provides higher-level predicates that quantify over relations between AST nodes that are not explicit in the AST representation. Examples include `classCallsClass/2` and `accessorMethodInClassForInstanceVariable/3`. As the higher-level predicates are implemented in terms of the reification predicates, the abductive logic reasoner only considers the latter as abducible predicates —each explanation should be minimal. Implementation-wise, predicates are declared as abducible using rules of the following form:

```
abducible(?abductiveIntention(?abduciblePredicate)) if
        ?groundingConditions
```

where *?abductiveIntention* is a constant `in` or `out` indicating whether *?abduciblePredicate* has to be added to or removed from the logic theory respectively. Although *?abduciblePredicate* can contain variables, explanations comprise ground atoms only. Logic variables within *?abduciblePredicate* are therefore either bound in the query evaluated by the abductive logic interpreter (i.e. the one corresponding to a design rule violation) or bound via the *?groundingConditions* conditions. This supports reasoning with incomplete information.

3.1.2. Declaring Corrective Actions for Abducibles

In our library, corrective actions are declared as rules of the following form:

```
correct(?abducedHypothesis, ?action) if ?generatorConditions
```

Here, *?abducedHypothesis* is a hypothesis from an abduced explanation (i.e. a ground atom corresponding to an abducible predicate) and *?action* a Smalltalk expression performing the corrective action upon the user's request. An abduced explanation consists of abducible predicates that will explain an observation once they are added to or removed from the logic theory.

In our problem setting, those abducibles are ground reification predicates that reify the relations between the nodes in an abstract syntax tree representation of the program. An explanation that requires a predicate to be added to or retracted from the logic theory implies changes to the program's AST. Rather than changing the reification of the program and subsequently constructing the modified program from the updated reification, we directly change the program's AST itself. This is possible because SOUL's symbiosis with Smalltalk (i.e. logic variables can be bound to Smalltalk objects and Smalltalk expressions can be used within logic queries) enables reifying an AST node as the AST node itself (i.e. a Smalltalk object rather than a compound term). Hence, most corrective actions are performed by sending messages to AST nodes. The logic conditions *?generatorConditions* support the corrective action.

In the rest of this section we give examples about how we can associate abducibles with corrective actions. These examples will be revisited in the next section. As a first example, the corrective action associated with the retraction of a predicate `statementOfMethod/2` from a theory is shown in listing 4.

```
1 correct(out(statementOfMethod(+?statement,+?method)),
2   [[Soul.MLI forSmalltalk updateMethod: ?method withCode: ?newMethodCode]]) if
3     methodSourceCode(?method,?previousMethodCode),
4     generate(methodWithoutStatement(?method,?statement),?newMethodCode)
```

Listing 4: Corrective action for *statementOfMethod/2* (deleting the statement).

The `out` functor in the declaration of the corrective action implies that this action is intended to retract the ground predicate `statementOfMethod(?statement, ?method)` from our logic theory. The action will suggest the elimination of the conflictive statement from the method.

Note that multiple corrective actions can be associated with an abduced hypothesis. For example, although in the previous case deleting a conflicting statement could be the right action to accomplish in many situations, in other occasions we would like to replace such statement by another one (like in the *Seaside* case). A corrective action producing such effect is shown in listing 5.

```
1 correct(out(statementOfMethod(RBMessageNode(?,{renderContentOn:},?),+?method)),
2   [[Soul.MLI forSmalltalk updateMethod: ?method withCode: ?newMethodCode]]) if
3     methodSourceCode(?method,?previousMethodCode),
4     statementsOfMethod(?statements,?method),
5     replaceOccuranceWith(?statements,?newStatements,
6       RBMessageNode(?rec,{renderContentOn:},?args),
7       RBMessageNode(?rec,{renderOn:},?args)),
8     methodSignature(?method,?signature),
9     methodSourceCode(?signature,?newStatements,?newMethodCode)
```

Listing 5: Corrective action for *statementOfMethod/2* (replacing the statement).

As a last example, in listing 6 we define a corrective action associated with the addition of the abduced hypothesis `statementsOfMethod/2` to a theory.

```
1 correct(in(statementsOfMethod(+?modelStatements,+?method)),
2   [[Soul.MLI forSmalltalk updateMethod: ?method withCode: ?newMethodCode]]) if
3     methodSourceCode(?method,?previousMethodCode),
4     methodSignature(?method,?signature),
5     statementsOfMethod(?oldStatements,?method),
6     adapt(?oldStatements,?modelStatements,?newStatements),
7     methodSourceCode(?signature,?newStatements,?newMethodCode)
```

Listing 6: Corrective action for *statementsOfMethod/2*.

The condition in the body of the rule (line 7) binds *?newMethodCode* to a string corresponding to the source code of a Smalltalk method which follows the pattern given by *?modelStatements*. The actual corrective action (line 2) is a

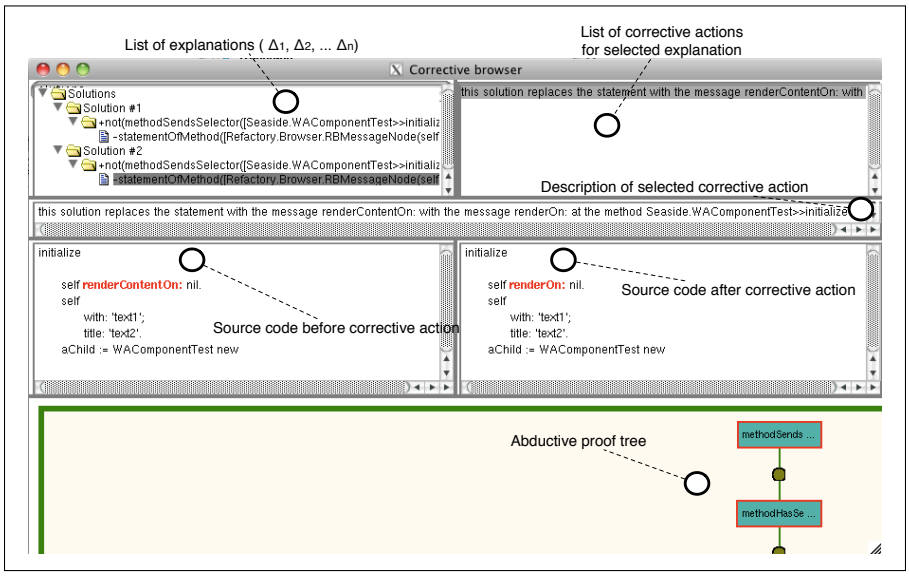

Figure 1: Corrective actions for a method invoking *#renderContentOn:*.

Smalltalk block that, upon request by the user, will update the source code of the method *?method* with the string *?newMethodCode*—thus correcting the hypothesized cause of an inconsistency.

In the next section we will make use of these associations of abducible predicates with corrective actions, in order to diagnose and correct inconsistent source code.

4. Illustrative Examples

In this section we illustrate how the consistency checks presented in section 2 can be diagnosed and corrected in our approach. At the end of this section, we will also show how different corrections can be composed with our tool.

4.1. Correcting methods sending `renderContentOn:`

By means of a unary relationship imposed on top of the *WAComponents methods* intensional view, we constrained that the methods belonging to this view should not invoke the message `#renderContentOn:`. We verify that this property holds, executing this query for every method in the view:

```
not(methodSendsSelector(?method,{renderContentOn:}))
```

In case of failure (i.e., when a tested method does send the selector `#renderContenOn:`), HEAL will attempt to find an update to our logic theory that would change the result of this query, making it true. Using logic abductive reasoning, we find that in case of retracting all the facts from the theory with this form:

```
statementOfMethod(wrongCallStatement, aFaultyMethod)
```

the consistency will be restored. This solution is found since `statementOfMethod/2` was declared as abducible (see listings 4 and 5).

Figure 1 depicts the HEAL browser (produced with the browser scripting language *Glamour* [7]) with the actual feedback for the methods violating the design rule that the message `#renderContentOn:` should not be used.

The bottom pane contains the top of the abductive proof tree providing a graphic representation for the abductive inference process. It is drawn using the *Mondrian* [24] visualization engine.

The pane in the top-left corner lists the abduced explanations for the violation. Once users have selected a possible explanation from this list, they are offered the corresponding list of corrective actions in the next pane. In the second row, a pane shows a textual description for the selected corrective action. In the two panes present in the third row, we

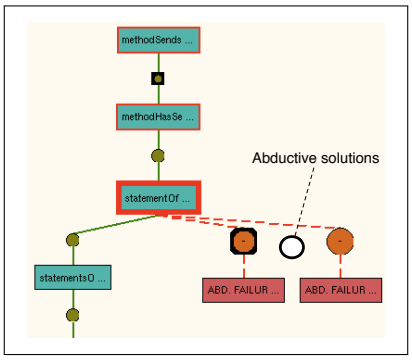

Figure 2: Proof tree for the *renderContenOn* design rule.

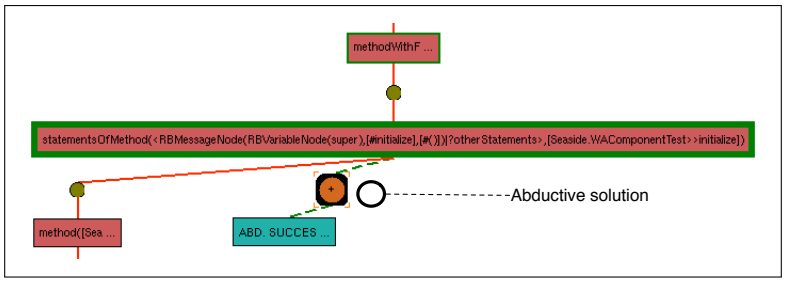

Figure 3: Proof tree for the *super* design rule.

can see a comparison between how the code of a violating method looks before applying certain corrective action (left pane) and what the method would look like if the corrective action is applied (right pane) making the method comply with the design rule.

Note that there are two corrective actions available: the first just eliminating the faulty statements, the second replacing these statements with correct values.The browser depicts the two solutions for every detected inconsistency. The proof tree fragment shown in figure 2 illustrates how the solutions were found using abductive reasoning: complete lines denote logic SLDNF resolution and broken lines denote abductive resolution.

4.2. Correcting methods missing a super call

The design rule stating that methods overriding `initialize` should contain a `super` call as their first statement was documented in Section 2. IntensiVE identified a method with this name in class `IntensionalViewLayoutManager` as one of the violations of this design rule.

As we described before, such verification is accomplished testing that for all the methods that should follow this design rule, the following query succeeds:

```
methodBeginsWithSuperCall(?method)
```

The abductive logic interpreter detected that although this query was failing for the method: `IntensionalViewLayoutManager>>initialize`, our logic theory can be updated in such a way that this query would succeed.

Examining the definition of `methodBeginsWithSuperCall/2` (listing 2) and `methodWithFirstStatement/2` (listing 3), and inspecting the proof tree of the failing query (figure 3), we can observe that the failure of the predicate `statementsOfMethod/2`, in charge of verifying that the first statement of a method is a `super` call, is the cause of the failure of the query. Since the predicate `statementsOfMethod/2` is declared as an abducible and associated with a corrective action (listing 6), the interpreter was therefore able to abduce an explanation for this violation. This explanation has the form:

```
statementsOfMethod(<superCallStatement | otherStatements>,
        aFaultyMethod)
```

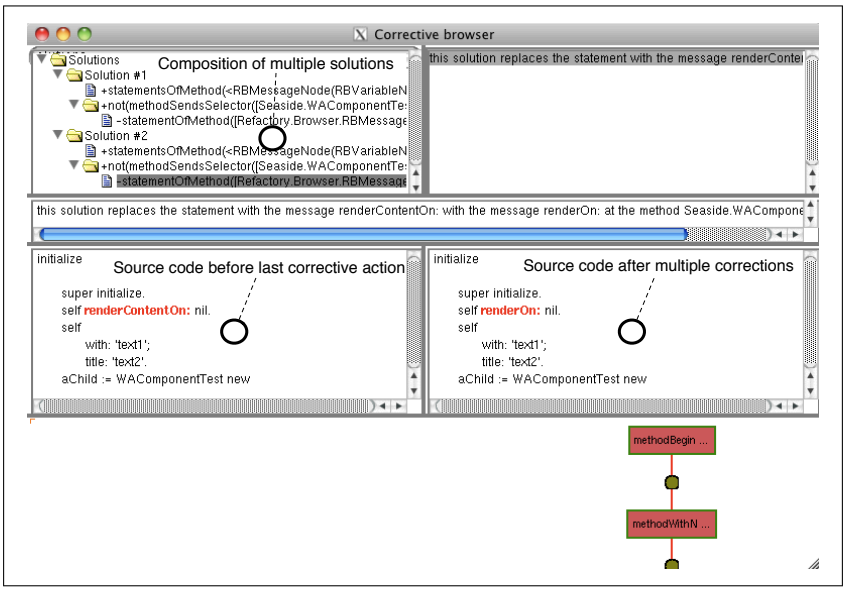

Figure 4: Corrective actions for a composite solution.

4.3. Composing solutions

Finally we will consider how the previous solutions can be composed by HEAL.

The main idea behind composition of solutions is that HEAL, during the abductive inference process, is able to remember previous assumptions over the structure of source code elements. In this way, new corrective actions can take always into consideration the last state of modified source code elements.

For example, if we combine our two previous verifications in one single design rule, we would express it with the following query:

```
not(methodSendsSelector(?method,{renderContentOn:}))
methodBeginsWithSuperCall(?method)
```

In case of a method with multiple faults (i.e., sending `#renderContentOn:` and not begining with a `super` call), the abductive reasoning process will:

- Detect that the first predicate is failing. Then it will attempt to generate an hypothesis in which the possibly correct method code does not send `#renderContentOn:` anymore.
- Detect that the new code of the method is still breaking a design rule (it does not include a `super` call). Then generate a new hypothesis starting from the last version of the method, in which the new problem is corrected.

Note that the result of method transformations are not applied to the real code during the inference process. However, the abductive reasoner can know how the last version of the method looks like in any state of such inference process. Once the inference process is finished, it is the responsibility of the programmer to select and apply the right combination of corrective actions to a faulty source code element.

Figure 4 shows the output of HEAL for our composition example. There we can see how multiple composed corrective actions are presented to the programmer. At the left bottom of the HEAL browser, we can see how the code would look like just before applying the last corrective action. Note that in this version of the method one corrective action has already been applied, since the method already include a `super` call. However, the method is still making a call to `#renderContentOn:`. The final state of the method, after applying all the corrections, is shown at the panel of the right. No errors are left.

5. Future work

Our current ongoing work focusses on researching *strategies* for generating more refined corrective actions for inconsistencies. Particularly, we are interested in finding heuristics for filtering the number of results generated by our approach, i.e., guiding the abductive inference process.

In our problem domain, one promising direction seems to develop heuristic strategies that analyze not only the structure of the source code artifacts that violate a design rule, but also of the artifacts that satisfy this rule. By comparing these two code fragments, information might be obtained concerning the exact statements in the source code that should be modified in order to correct the violation.

Finally, we are also interested in the generalization of our technique for the diagnosis and correction of inconsistencies between different kinds of software models. Concretely, we are interested to know if HEAL can be used for diagnosing inconsistencies between design and UML models.

6. Conclusion

In this paper we have presented Heal, a framework based on abductive logic reasoning for diagnosing and correcting violations of design rules in program code. Heal has been implemented as an extension to the IntensiVE tool suite, that allows for documenting design rules and verifying these design rules with respect to the source code. Underlying this IntensiVE tool suite lies the use of the logic program query language SOUL to quantify over a system's source code and define sets source code artifacts referred as Intensional Views. Design rules are expressed with queries denoting relationships that should hold between Intensional Views. By means of logic abduction over these logic queries, our approach is able to provide a developer feedback concerning possible hypotheses that might cause violations of design rules. In order to correct the hypothesized causes of inconsistencies, our approach provides a library of associated corrective actions for basic structural problems in code. HEAL composes these primitive solutions in order to generate a set of corrective actions that will bring design and code back to a consistent state.

Acknowledgements

We would like to acknowledge Johan Brichau for his invaluable comments and suggestions. This work was partially funded by the Interuniversity Attraction Poles program of the Belgian Science Policy Office.

[1] C. Bauer and G. King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

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

[3] A. P. Black, S. Ducasse, O. Nierstrasz, and D. Pollet. *Pharo by example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[4] J. Brichau, C. De Roover, and K. Mens. Open unification for program query languages. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, 2007.

[5] J. Brichau, A. Kellens, S. Castro, and T. D'Hondt. Enforcing structural regularities in software using intensive. In *International Workshop on Advanced Software Development Tools and Techniques (WASDeTT). Co-located with the European Conference in Object-Oriented Programming (ECOOP)*, 2008.

[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] P. Bunge. Scripting browsers with glamour. Master's thesis, Bern University, April 2009.

[8] S. Castro, C. D. Roover, A. Kellens, A. Lozano, K. Mens, and T. D'Hondt. Diagnosing and correcting design inconsistencies in source code with logical abduction. *Science of Computer Programming*, In Press, Corrected Proof:–, 2010.

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

[10] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving your software using static analysis to find bugs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 673–674, New York, NY, USA, 2006. ACM.

[11] J. Coplien. *Advance C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[12] C. De Roover. *A Logic Meta Programming Foundation for Example-Driven Pattern Detection in Object-Oriented Programs*. PhD thesis, Vrije Universiteit Brussel, August 2009.

[13] C. De Roover, I. Michiels, K. Gybels, K. Gybels, and T. D'Hondt. An approach to high-level behavioral program documentation allowing lightweight verification. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC06)*, pages 202–211. IEEE Computer Society, 2006.

[14] T. D'Hondt, K. D. Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In M. Aksit, editor, *Software Architectures and Component Technology*, pages 207–224. Kluwer Academic Publisher, January 2001. Proceedings of SACT 2000.

[15] P. Flach. *Simply logical: intelligent reasoning by example*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
[17] Y.-G. Guéhéneuc. Ptidej: A flexible reverse engineering tool suite. In *ICSM*, pages 529–530. IEEE, 2007.
[18] S. Johnson. Lint. http://www.jutils.com/, 2007.
[19] A. Kellens. *Maintaining causality between design regularities and source code*. PhD thesis, Vrije Universiteit Brussel, 2007.
[20] A. Kellens, K. D. Schutter, T. D'Hondt, L. Jorissen, and B. V. Passel. Cognac: A framework for documenting and verifying the design of cobol systems. In *Proceedsings of the 13th European Conference on Software Maintenance and Reengineering (CSMR09)*, pages 199–208, 2009.
[21] K. Mens and A. Kellens. IntensiVE, a toolsuite for documenting and testing structural source-code regularities. *10th Conference on Software Maintenance and Re-engineering (CSMR)*, pages 239–248, 2006.
[22] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. The intensional view environment. *International Conference on Software Maintenance (ICSM) Industrial and Tool Volume*, pages 81–84, 2005.
[23] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2001.
[24] M. Meyer, T. Girba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization*, pages 135–144, 2006.
[25] N. H. Minsky. Law-governed regularities in object systems: part i: an abstract model. *Theor. Pract. Object Syst.*, 2(4):283–301, 1996.
[26] Object Technology International, Inc. *Eclipse platform. Technical overview. White Paper*, July 2001.
[27] M. Perscheid, D. Tibbe, M. Beck, S. Berger, P. Osburg, J. Eastman, M. Haupt, and R. Hirschfeld. *An Introduction to Seaside*. Software Architecture Group (Hasso-Plattner-Institut), 2008.
[28] C. S. Pierce. *The Collected Papers of Charles Sanders Peirce*. Harvard University Press, 1935.
[29] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, April 1987.
[30] Semmle Ltd. SemmleCode. http://semmle.com/, 2010.
[31] SMACCHIA.COM S.A.R.L. NDepend. http://www.ndepend.com/, 2010.
[32] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.