SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2228

A recommendation system for repairing violations detected by static architecture conformance checking

Ricardo Terra[1,2,*,†], Marco Tulio Valente[1], Krzysztof Czarnecki[2] and Roberto S. Bigonha[1]

[1]*Universidade Federal de Minas Gerais, Brazil*
[2]*University of Waterloo, Canada*

SUMMARY

This paper describes a recommendation system that provides refactoring guidelines for maintainers when tackling architectural erosion. The paper formalizes 32 refactoring recommendations to repair violations raised by static architecture conformance checking approaches; it describes a tool—called ArchFix—that triggers the proposed recommendations; and it evaluates the application of this tool in two industrial-strength systems. For the first system—a 21 KLOC open-source strategic management system—our approach has indicated correct refactoring recommendations for 31 out of 41 violations detected as the result of an architecture conformance process. For the second system—a 728 KLOC customer care system used by a major telecommunication company—our approach has triggered correct recommendations for 624 out of 787 violations, as asserted by the system's architect. Moreover, the architects have scored 82% of these recommendations as having *moderate* or *major* complexity. Copyright © 2013 John Wiley & Sons, Ltd.

Received 24 August 2012; Revised 17 August 2013; Accepted 19 August 2013

KEY WORDS: software architecture; refactoring; recommendation system

1. INTRODUCTION

Software architecture erosion is one of the most evident manifestations of software aging [1–3]. The phenomenon designates the progressive gap normally observed between two architectures: the *intended architecture* defined during the architectural design phase and the *concrete architecture* defined by the current implementation of the software system [4–6]. Causes of architectural erosion include deadline pressures, conflicting requirements, miscommunication, developers' unawareness, and the lack of an explicit correspondence between architectural and programming language abstractions. Regardless the causes, when the erosion is neglected over long periods, it can reduce the concrete architecture to a small set of strongly coupled and weakly cohesive components, whose maintenance and evolution become increasingly more difficult and costly [3, 7].

To tackle the erosion process, the first task is to check whether the concrete architecture conforms to the intended one [3, 8–10]. More specifically, the goal of an architecture conformance process is to reveal the implementation decisions that denote architectural violations, that is, the concrete statements, expressions, or declarations in the source code that do not match the constraints imposed by the intended architecture. For this purpose, several architecture conformance techniques have been proposed, including reflexion models [11], intensional views [12], design tests [13], architecture description languages [14], and domain-specific languages [15–17].

After the conformance phase, the next task is to replace the detected violations with implementation decisions consistent with the intended architecture. However, this reengineering effort is usually

*Correspondence to: Ricardo Terra, Universidade Federal de Minas Gerais, Brazil.
[†]E-mail: terra@dcc.ufmg.br

Copyright © 2013 John Wiley & Sons, Ltd.

Figure 1. Proposed architectural repair recommendation approach.

a nontrivial and time-consuming task. For example, Knodel *et al.* described their experience of applying an architecture conformance process to a product line in the domain of portable measurement devices. As a result, they identified almost 5000 architectural divergences in three products of this product line [18]. In previous work [16], we described our own experience in applying conformance techniques to a human resource management system. In this process, we have been able to detect more than 2200 architectural violations. As a last example, Sarkar *et al.* reported their experience in remodularizing a large banking application. Reconstructing the original architecture of this system demanded 2100 person-days just for coding and testing [7].

However, despite of its relevance and in contrast to the variety of techniques available for architecture conformance, the task of fixing architectural violations is usually performed in an ad hoc way. Usually, the only employed tools are the automatic refactorings provided by today's integrated development environments or simple program analysis tools, such as those that extract function call information [7]. In view of such circumstances, a solution based on recommendation system principles may represent a promising approach. Such a solution may provide refactoring guidelines for developers and maintainers when fixing architectural violations. Moreover, by definition, an approach based on recommendations does not have the ambition to provide a fully automatic solution to remove architectural violations, which is certainly a task ahead the state of the art in reengineering tools. In fact, even a bug-free implementation for typical refactorings, that is, refactorings whose scope are limited to a few classes, has been proved to be a complex task [19, 20].

In this paper, we propose an architectural repair recommendation system whose main purpose is to provide refactoring guidelines for developers and maintainers when fixing violations in the module architecture view of object-oriented systems. As illustrated in Figure 1, considering a set of architectural violations raised by a static architecture conformance tool, the proposed recommendation engine—called `ArchFix`—provides refactoring recommendations to guide the process of repairing each detected violation. For example, we may suggest the use of a *Move Method* refactoring to fix a given violation, including a suggestion of a target class.

In a recent short paper, we discussed the preliminary design of our recommendation approach [21]. This paper extends our earlier work with the following contributions: (a) an extensive set of 32 recommendations for fixing architectural violations, including violations due to divergences and absences (in our previous work, we presented a preliminary set of 10 recommendations targeting only divergences); (b) the design and implementation of `ArchFix`, a recommendation system that triggers the proposed architectural refactoring recommendations; and (c) an evaluation of our approach in two real-world systems. For the first system—a 21 KLOC open-source strategic management system—our approach indicated the correct refactoring for 75% of the detected violations. For the second system—a 728 KLOC customer care system used by a major telecommunication company—our approach triggered correct recommendations for 79% of the violations, as endorsed by the architects. Moreover, the architects marked 82% of these recommendations as having *moderate* or *major* complexity.

The remainder of this paper is structured as follows. Section 2 provides a definition for central concepts needed to follow our approach, such as *architectural model*, *architectural violations*, and *refactoring recommendations*. Section 3 presents a formal specification of the proposed architectural repair recommendation system, including the description of a subset of recommendations, underlying algorithms, and similarity functions. Section 4 describes the design and implementation of the `ArchFix` tool. Section 5 presents and discusses results on applying these recommendations in two real-world systems. Finally, Section 6 presents related work and Section 7 concludes the paper.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

2. BASIC CONCEPTS

In this section, we provide definitions of the following fundamental concepts employed in this paper: *recommendation system*, *architectural model*, *architectural violation*, *architecturally defective code*, and *refactoring recommendation*.

Recommendation system

A *recommendation system* is a software system that provides potentially valuable information in a given context [22–24]. In the particular context of software engineering, recommendation systems can recommend, for example, relevant source code fragments to help developers use frameworks and APIs (e.g., Strathcona [25]), software artifacts that should be changed together (e.g., eRose [26]), and replacement methods for adapting code to a new library version (e.g., SemDiff [27]). In this paper, we propose a novel recommendation system, called `ArchFix`, which provides refactoring recommendations to repair architectural violations. The proposed system is defined by the following function:

$$\textit{Arch. Model} \times \textit{Arch. Violation} \times \textit{Arch. Defective Code} \longrightarrow \textit{Refactoring Recommendation}$$

In short, our system receives as input the intended architectural model (*Arch. Model*), the description of the violation (*Arch. Violation*), and the piece of source code that raised the architectural violation (*Arch. Defective Code*). `ArchFix` then returns a *Refactoring Recommendation* that might be useful to repair such violation. In the remainder of this section, we discuss such elements in more detail.

Architectural model

Kruchten defines software architecture using five concurrent *views*, each one addressing a specific set of concerns of interest to different stakeholders [28]. Particularly, our work is centered on the *development view* (a.k.a. *module view*), which describes the software's static organization in its development environment. It concerns low-level design decisions, patterns, and best practices. From this viewpoint, an object-oriented software architecture is defined by a set of modules and their interactions, where we will consider a module as a set of classes [16]. Therefore, we model relations at the level of classes. More specifically, a dependency (`A`, `dep`, `B`) is established whenever a class `A` uses services provided by a target class `B`. We consider that dependencies can be established using the following types of common operations in object-oriented languages (i.e., these types are possible values of the `dep` field): calling methods or attributes (`access`), declaring variables (`declare`), creating objects (`create`), extending classes or implementing interfaces (`derive`), throwing exceptions (`throw`), or using annotations (`useannotation`). For example, the relations (`A`, `create`, `B`) and (`A`, `access`, `B`) indicate that class `A` creates and calls methods of an object of type `B`, respectively.

The *architectural model* considered by `ArchFix` is expressed in terms of architectural constraints, which are formalized as follows:

$$\textit{Module}_1 \quad [\texttt{cannot}|\texttt{must}]\textit{–dep} \quad \textit{Module}_2$$

where *dep* denotes the dependency type, that is, *dep* can be `access`, `declare`, `create`, and so on. As an illustrative example, assume that $\texttt{M}_\texttt{A}$ and $\texttt{M}_\texttt{B}$ are modules, that is, sets of classes. A constraint in the form $\texttt{M}_\texttt{A}$ `cannot-dep` $\texttt{M}_\texttt{B}$ indicates that types from module $\texttt{M}_\texttt{A}$ *cannot* establish a dependency of the `dep` kind with types from module $\texttt{M}_\texttt{B}$, for example, `ViewLayer cannot-access ModelLayer`. On the other hand, a constraint in the form $\texttt{M}_\texttt{A}$ `must-dep` $\texttt{M}_\texttt{B}$ indicates that types from module $\texttt{M}_\texttt{A}$ *must* establish a dependency of the `dep` kind with types from module $\texttt{M}_\texttt{B}$, for example, `DTO must-implement Serializable`.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Architectural violation

Basically, there are two types of violations in the static architecture of software systems: *divergences* (when an existing dependency in the source code violates the architectural model) and *absences* (when the source code does not establish a dependency, that is, prescribed by the architectural model) [8,9,11]. In our approach, divergences occur when architectural constraints of type `cannot` are not respected by the source code. Conversely, absences occur when architectural constraints of type `must` are not respected.

We consider that an *architectural violation* is defined by a tuple [`A`, `viol_type`, `dep`, `B`], where (`A`, `dep`, `B`) is the dependency that caused the violation and `viol_type` indicates whether the violation is due to a divergence or an absence. To increase readability, when expressing a value for `viol_type`, we use the words `cannot` and `must` to denote divergences and absences, respectively. For example, a violation [`ProductView`, `cannot`, `access`, `ProductModel`] denotes a divergence where a class `ProductView` is accessing an object of type `ProductModel`. As another example, a violation [`ProductDTO`, `must`, `extend`, `Serializable`] denotes an absence when the class `ProductDTO` is *not* extending the class `Serializable`.

Formal definition: Assume that $\texttt{M}_\texttt{A}$ and $\texttt{M}_\texttt{B}$ are modules, `A` and `B` are classes, and `dep` denotes a dependency type, that is, `dep` can be `access`, `declare`, `create`, and so on. A violation of a constraint in the form $\texttt{M}_\texttt{A}$ `cannot-dep` $\texttt{M}_\texttt{B}$ happens whenever

$$\exists \texttt{A}\, \exists \texttt{B}\, [\texttt{A} \in \texttt{M}_\texttt{A} \wedge \texttt{B} \in \texttt{M}_\texttt{B} \wedge \texttt{establishes}(\texttt{A}, \texttt{dep}, \texttt{B})]$$

where the predicate `establishes` checks whether there is a dependency of type `dep` from `A` to `B`. Conversely, a violation of a constraint in the form $\texttt{M}_\texttt{A}$ `must-dep` $\texttt{M}_\texttt{B}$ happens whenever

$$\exists \texttt{A}\, \nexists \texttt{B}\, [\texttt{A} \in \texttt{M}_\texttt{A} \wedge \texttt{B} \in \texttt{M}_\texttt{B} \wedge \texttt{establishes}(\texttt{A}, \texttt{dep}, \texttt{B})]$$

Architecturally defective code

An architectural violation, as defined before, statically occurs in a piece of code. In our recommendation system, we consider that the code responsible for a violation (i.e., the *architecturally defective code*) follows one of the code templates defined in Table I. As an example, the template `new B(exp)` covers instantiations of a given class `B`. As another example, consider the architectural constraint `UI cannot—access Bar` and the class `Screen` presented in Code 1.

```
1 public class Screen {
2     private Bar bar;
3     ...
4     public void init() {
5         ...
6         bar.foo(13,'b');
7     }
8 }
```

Code 1: Code template matching example

Assuming that `Screen` is located in module `UI`, a violation [`Screen`, `cannot`, `access`, `Bar`] is raised by the call `bar.foo(13,'b')` located at line 6. This call matches the template `b.f(exp)` with `b` bound to the target object `bar`, `f` bound to the method `foo`, and `exp` bound to the list of expressions `(13,'b')`.

Refactoring recommendation

In `ArchFix`, the recommendations consist in a refactoring activity to repair an architectural violation. In fact, a *refactoring recommendation* is a sequence of refactoring operations, using the functions described in Table II. Although most refactoring functions are familiar, Appendix A

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table I. Code templates.

Template	Interpretation
`class A`	Class implementation
`class A derive B`	Class that extends or implements type `B`
`@B class A`	Class with an annotation of type `B`
`g (p){ S }`	Implementation of a method `g`, with formal parameters `p` and body `S`
`@B g (p){ S }`	Implementation of a method `g`, with annotation `@B`, formal parameters `p`, and body `S`
`g (B b){ S }`	Implementation of a method `g`, with a formal parameter of type `B`, and body `S`
`g (p){ T v = exp_b }`	Implementation of a method `g`, with formal parameters `p` and whose body declares a local variable `v` of type `T` initialized with `exp_b` (which returns an object of type `B`)
`g (p){ return new B(exp) }`	Implementation of a method `g`, with formal parameters `p`, returning an object of type `B` created using parameters `exp`
`g (p) throws B { S }`	Implementation of a method `g`, with formal parameters `p`, and body `S` that can throw an exception of type `B`
`B b; S`	Declaration of a variable `b` of type `B` followed by statements `S`
`B b = exp; S`	Declaration of a variable `b` of type `B`, initialized with `exp` and followed by statements `S`
`catch (B b) { S }`	Implementation of a *catch* clause for exceptions of type `B` and with a body `S`
`b.f(exp)`	Invocation of a method `f` using the target object `b` and actual parameters `exp`
`new B(exp)`	Instantiation of an object of type `B` using parameters `exp`

Table II. Refactoring functions.

Function	Description	Type
`extract(stm)`	Applies an *Extract Method* refactoring [29] with the set of statements `stm`	*lt*
`inline(exp, v, S)`	Inlines `exp` in the uses of variable `v` in the block of code `S` (Appendix A)	*lt*
`move(f, M)`	Moves method `f` to the most suitable class in module `M`, that is, applies a *Move Method* refactoring [29]	*mv*
`move(C, M)`	Moves class `C` to module `M`, that is, applies a *Move Class* refactoring [29]	*mv*
`promote_param(f, v, exp)`	Promotes variable `v` to a formal parameter of method `f`; `exp` is used as the additional a rgument in the calls to `f` (Appendix A)	*lt*
`replace(`S_1`, `S_2`)`	Replaces block of code S_1 with S_2	*lt*
`remove(S)`	Removes block of code `S`, which is equivalent to `replace(S, ffi)`	*lt*
`remove_catch(Ex, S)`	Removes the catch clause for exception `Ex` from the *try-catch* block `S`	*lt*
`unwrap_return(f, T, exp)`	Modifies the return type of method `f` to the type of `exp` and moves the instantiations of the wrapper type `T` to the respective call sites (Appendix A)	*lt*

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

provides a detailed description of the following particular functions: `promote_param`, `inline`, and `unwrap_return`.

As can be observed in Table II, the recommendations include two types of refactorings:

- (*lt*) stands for a *local transformation* restricted to the method or class, which is a refactoring more common to handle divergences. For instance, suppose that a given module should not use `List` objects, but can use `Collection`. In this case, function `replace([List],[Collection])` locally replaces a declaration of the unauthorized type `List` with its supertype `Collection`. As another example, we can mention a refactoring that replaces an unauthorized class instantiation with a call to its factory method—`replace([new ProductDAO()],[DAOFactory.getProductDAO()])`.
- (*mv*) stands for a transformation that requires *moving* code, which may be useful to handle both divergences and absences, as will be largely discussed in Section 3.5. As an example, suppose that a class `ProductReport` is implemented in module `View`, whereas the architectural model prescribes that all reports must be implemented in module `Report`. In this case, function `move(ProductReport, Report)` can indicate the correct repair action to handle this particular violation.

3. ARCHITECTURAL REFACTORING RECOMMENDATIONS

This section defines the architectural refactoring recommendations triggered by `ArchFix`. We start by explaining how the recommendations have emerged, and then, we define and illustrate our recommendations.

3.1. Training system

The proposed recommendations emerged after an in-depth investigation of possible fixes for more than 2200 violations we detected in a previously evaluated system called SGP (our training set system is different from the systems used for evaluation in this paper) [16]. The choice of SGP was motivated by the following facts: (i) it is a large web system with around 240 KLOC, (ii) its architecture reflects the one commonly used in Java-based web systems, and (iii) the system was facing a serious architecture erosion process, that is, we detected 2241 architectural violations.

We considered this set of violations as our *training set* to define our recommendations. We analyzed and generalized the solution (refactoring task) employed by the system architect for each violation. More important, we also documented the preconditions that allow each refactoring to be applied. As a result, we reached a catalog of 32 architectural refactoring recommendations. We hypothesized that many of the recommendations would be applicable to other systems. Our evaluation confirms this hypothesis; however, we acknowledge that the catalog is likely incomplete.

3.2. Syntax and auxiliary functions

Assuming the definitions and concepts outlined in Section 2, the architectural refactoring recommendations triggered by our system are defined using the following syntax:

```
[A, viol_type, dep, B]
code_template  ⟹  recommendation,  if preconditions
```

This recommendation syntax is interpreted as follows: whenever the indicated violation [A, `viol_type`, `dep`, B] is found in a piece of code that matches the `code_template` and the `preconditions` hold, the `recommendation` is triggered (i.e., suggested to the user).

A *recommendation* consists of one or more *refactoring* functions, as defined in Table II. Table III lists a set of auxiliary functions used to define the `preconditions`. Some of such functions define a simple source code query language, including functions such as `call_sites(f)`, which

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table III. Auxiliary functions.

Function	Description
`call_sites(f)`	Returns the statements that call function `f`
`can(`T_1`, dep, `T_2`)`	Checks whether a dependency of the `dep` type from type T_1 to type T_2 does not raise any violation, that is, respects the architectural model
`has_catch(Ex, S)`	Checks whether there is a *catch* clause for exception `Ex` in the *try-catch* block `S`
`delegate(f)`	Searches for a delegate method for `f`, that is, a method that just encapsulates a call to `f` (Appendix B)
`equals_sig(`f_1`, `f_2`)`	Checks whether methods f_1 and f_2 have the same signature
`factory(C, exp)`	Searches for a factory method for class `C`, accepting `exp` as input (Appendix B)
`gen_decl(f)`	Declares a variable of type `C`, which defines the type `C`, to be the target of a call to method `f` (Appendix B)
`gen_factory(C, exp)`	Generates a factory for class `C`, accepting `exp` as parameter (Appendix B)
`override(`C_1`, `C_2`)`	Checks whether class C_1 overrides a method defined in superclass C_2
`sub(T)`	Returns the subtypes of type `T`
`suitable_module(E)`	Returns the most suitable module for a source code entity `E` (Section 3.5)
`super(T)`	Returns the supertypes of type `T`
`target(A)`	Returns the target entity of an annotation `A`, which can be a *type* or a *method*
`type(v)`	Returns the type of variable `v`
`typecheck(stm)`	Checks whether code `stm` type checks
`user_code()`	Returns a block of code with only a TODO comment, which the user must fill

returns the call sites of a given method `f`, and `super(T)`, which returns the supertypes of a given type `T`. Other functions are slightly more complex, such as `can(`T_1`, dep, `T_2`)`, which checks whether the establishment of a dependency (T_1, `dep`, T_2) raises an architectural violation, and `typecheck(stm)`, which checks whether a piece of code compiles without type errors. In addition, Appendix B provides a detailed description of the following nontrivial functions: `delegate`, `factory`, `gen_decl`, and `gen_factory`.

3.3. Recommendations

Table IV specifies the recommendations using the aforementioned syntax and functions. The table formalizes recommendations to address both divergences (recs. $D1$ to $D24$) and absences (recs. $A1$ to $A8$). In the specifications, we consider that `M(A)`—or just M_A for the sake of simplicity—is a total function that returns the module of a given class `A`. We also consider that $\overline{M_A}$ is the complement of the module returned by M_A, that is, all classes in the project under analysis except those in M_A. Furthermore, the recommendations are listed according to their priority. For instance, assume a violation [`A, cannot, throw, B`]. In this particular case, a recommendation for removing the `throws` clause (rec. $D15$) has a higher priority than the recommendation to handle the exception internally (rec. $D16$). In this case, the order was defined considering the complexity of the recommended refactorings. More specifically, it is simpler to remove a `throws` clause than to include a `catch` statement and the respective exception handling code.

To provide an overview of our architectural refactoring recommendations, we describe next a small subset of our recommendations:

- $D2$: Replace the unauthorized type `B` with one of its subtypes `B′`. As an example, developers when implementing web-based systems using Google Web Toolkit (GWT) should avoid the use of generic types (e.g., `java.util.Collection`) on GWT

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table IV. Refactoring recommendation.

[A, cannot, declare, B]		
B b; S	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ super(B) ∧ typecheck([B′ b ; S]) ∧ B′ ∉ M_B	$D1$
B b; S	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ sub(B) ∧ typecheck([B′ b ; S]) ∧ B′ ∉ M_B	$D2$
B b = exp; S	$\Longrightarrow$ propagate([exp], b, [S]), if can(A, access, B)	$D3$
g (B b) { S }	$\Longrightarrow$ remove([B b]), if typecheck([g(){ S }])	$D4$
catch (B b) { S }	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ super(B) ∧ typecheck([catch(B′ b){ S }]) ∧ B′ ∉ M_B	$D5$
[A, cannot, access, B]		
b.f	$\Longrightarrow$ replace([b.f], [D; c.g]), if g = delegate(f) ∧ {D,c} = gen_decl(g) ∧ type(c) ∉ M_B	$D6$
b.f	$\Longrightarrow$ replace([b.f], [D; c.g]), if equals_sig(f, g) ∧ {D,c} = gen_decl(g) ∧ type(c) ∉ M_B	$D7$
b.f	$\Longrightarrow$ g = extract([b.f]), move(g, M), if M = suitable_module(g) ∧ can(A, access, M)	$D8$
b.f	$\Longrightarrow$ remove([b.f]), if $\overline{M_A} = \emptyset$	$D9$
g (p){ T v = exp_b }	$\Longrightarrow$ promote_param(g, v, [exp_b]), if ∀C ∈ call_sites(g), can(C, access, B)	$D10$
[A, cannot, create, B]		
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [FB.getB(exp)]), if FB = factory(B, [exp]) ∧ can(A, access, FB)	$D11$
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [null]), if $\overline{M_A} = \emptyset$	$D12$
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [FB.getB(exp)]), if FB = gen_factory(B, [exp]) ∧ can(A, access, FB)	$D13$
g (p){return new B(exp)}	$\Longrightarrow$ unwrap_return(g, B, [exp]), if ∀C ∈ call_sites(g), can(C, create, B)	$D14$
[A, cannot, throw, B]		
g (p) throws B { S }	$\Longrightarrow$ remove([throws B]), if typecheck([g (p) { S }])	$D15$
g (p) throws B { S }	$\Longrightarrow$ remove([throws B]), replace([S], [try {S} catch(B b) {S′}]), if can(A, declare, B) ∧ S′ = user_code()	$D16$
g (p) throws B { S }	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ super(B) ∧ B′ ∉ M_B	$D17$
g (p) throws B { S }	$\Longrightarrow$ move(g, M), if M = suitable_module(g) ∧ M ≠ M_A	$D18$
[A, cannot, derive, B]		
class A derive B	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ super(B) ∧ typecheck([A derive B′]) ∧ ¬override(B, B′) ∧ B′ ∉ M_B	$D19$
class A derive B	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ can(A, derive, B)	$D20$
[A, cannot, useannotation, B]		
@B class A	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ M ≠ M_A	$D21$
@B class A	$\Longrightarrow$ remove([@B]) if M_A = suitable_module(A)	$D22$
@B g (p){ S }	$\Longrightarrow$ move(g, M), if M = suitable_module(g) ∧ M ≠ M_A	$D23$
@B g (p){ S }	$\Longrightarrow$ remove([@B]) if M_A ≠ suitable_module(A)	$D24$
[A, must, throw, B]		
g (p){ S }	$\Longrightarrow$ replace([g (p){ S }], [g (p) throws B { S }]), remove_catch(B, S) if has_catch(B, S)	$A1$
g (p){ S }	$\Longrightarrow$ move(g, M) if M = suitable_module(g) ∧ M ≠ M_A	$A2$
[A, must, derive, B]		
class A	$\Longrightarrow$ replace([A], [A derive B]), if M_A = suitable_module(A) ∧ typecheck([class A derive B])	$A3$
class A	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ M ≠ M_A	$A4$
[A, must, useannotation, B]		
class A	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ M ≠ M_A ∧ target(B) = type	$A5$
class A	$\Longrightarrow$ replace([class A], [@B class A]), if M_A = suitable_module(A) ∧ target(B) = type	$A6$
g (p){ S }	$\Longrightarrow$ move(g, M), if M = suitable_module(A) ∧ M ≠ M_A ∧ target(B) = method	$A7$
g (p){ S }	$\Longrightarrow$ replace([g(p)], [@B g(p)]), if M_A = suitable_module(g) ∧ target(B) = method	$A8$

interfaces to reduce the size of the generated JavaScript code. Therefore, whenever possible, they should rely on more specialized types (e.g., `java.util.ArrayList` instead of `java.util.Collection`).

- $D9$: Remove a call to a given method `f` when no class in the system can access the class where `f` is implemented. It is particularly useful when developers access methods whose usage is restricted. For instance, developers tend to establish dependencies with the Java API `System` class (e.g., by calling `System.out.println`) as a practice of rudimentary debugging. Nevertheless, these calls must be removed, especially in web-based systems.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

- *D*11: Replace a `new` operator with a call to the `get` method of a Factory `FB`. It addresses the situation where developers—due to unawareness or forgetfulness—create directly objects of classes that have a well-defined factory.
- *D*16: Remove the `throws` clause and insert a `try−catch` block around the body of the method to handle a given exception internally. In this particular case, the developers must provide the code that handles the exception, as required by the auxiliary function `user_code`.
- *D*20: Move a class `A` to a most suitable module `M`. It is particularly useful when developers mistakenly implement a class in the wrong module, for example, a `ProductReport` class in the `View` layer.
- *A*3: Make class `A` extend or implement `B`. It addresses the situation where developers have failed to derive from the base types of the module. For instance, an `Entity` class must implement `Serializable` to provide persistence. However, assume that a given entity class `Product` does not implement `Serializable`. Because `Entity` classes rely extensively on the same types, the `suitable_module` function will likely infer that `Product` is indeed in its correct module and therefore, must implement `Serializable`.

Section 5 provides concrete examples on the use of the proposed recommendations in two real systems. Furthermore, a detailed description of all architectural refactoring recommendations is available in a companion website.[‡]

3.4. Algorithm

In order to provide an operational description for our approach, Algorithm 1 presents the algorithm followed by `ArchFix` to trigger architectural refactoring recommendations. First, function `getRecommendations()` returns a list with the possible recommendations for a particular violation (line 1). For instance, assume a violation [`A`, `must`, `useannotation`, `B`], the function returns the priority-sorted list of recommendations [`A5`, `A6`, `A7`, `A8`]. Next, we iterate over this list (lines 2–6). Whenever the code responsible for the violation matches the code template of a possible recommendation and the preconditions hold (line 3), the recommendation is returned (line 4). When none of the possible recommendations follows our conditions, we do not return any recommendation (line 7).

Algorithm 1 Recommendation Algorithm

Input: Architectural model ($arch$), violation ($viol$), and defective code ($code$)
Output: Refactoring recommendation

```
1: potentialRecs ← getRecommendations(viol)                    ▷ Sorted by priority
2: for each rec in potentialRecs do
3:     if (code matches rec.code_template and ⟨code, viol, arch⟩ satisfies rec.preconditions) then
4:         return rec.recommendation
5:     end if
6: end for
7: return ϕ                                                    ▷ No recommendation
```

Note that when multiple recommendations match a given violation, the algorithm returns only the one with the highest priority.

3.5. Module suitability

In Table IV, 14 out of 32 recommendations (e.g., *D*18, *D*20, *D*21, *A*4, etc.) include a suggestion to move methods or classes to more *suitable* modules, as computed by the function `suitable_module`. In essence, this function considers the structural similarity among source

[‡]http://www.dcc.ufmg.br/~terra/spe2013

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

code entities to make a recommendation. In order to measure this similarity, we use the Jaccard similarity coefficient [30], which is a statistical measure for the similarity between two sets. To calculate this coefficient, we assume that a given module is represented by the set of dependencies it establishes with the other modules. This assumption is based on the fact that our recommendations are proposed to remove divergences or absences in the structural dependencies established between the modules of object-oriented software architectures.

Based on such assumptions, the Jaccard similarity between modules M_1 and M_2 is defined by the following:

$$sim\,(M_1, M_2, dep) = \frac{|Deps(M_1, dep) \cap Deps(M_2, dep)|}{|Deps(M_1, dep) \cup Deps(M_2, dep)|}$$

where dep denotes the dependency type (e.g., `access`, `create`, etc.) that motivated the similarity calculation. In addition, $Deps(M, dep)$ is a set of pairs (dep, T), denoting the existence of a dependency of type dep between a class C_i and a given type T, where $C_i \in M$. When $dep = *$, we do not consider a particular dependency type when calculating $Deps$, that is, all dependencies of any type are included in the resulting set.

Suppose that a violation is detected in module M_1 involving a dependency of type dep. Suppose also that the recommendation for this violation requires moving the class responsible for the violation to another module M_2. In this case, M_2 is defined as the module m that provides the following maximal value:

$$\max \left\{ \max_{\forall m} sim(M_1, m, dep),\ \max_{\forall m} sim(M_1, m, *) \right\}$$

In other words, the most suitable module is the module with the highest Jaccard coefficient—as returned by the sim function—considering two possible sets of dependencies: (1) only dependencies of type dep and (2) all dependencies, independently of their type (i.e., $dep = *$). This second alternative is particularly important when making recommendations for absences, because by definition, the module M_1 in this case misses a dependency of type dep, that is, $Deps(M_1, dep) = \emptyset$.

4. THE `ARCHFIX` TOOL

We developed a prototype tool called `ArchFix` that implements our approach. In its current implementation, `ArchFix` relies on violations raised by the Dependency Constraint Language (DCL) [16, 31], which is a domain-specific language for defining structural constraints between modules in Java. Compared with other approaches, DCL relies on a simple syntax and supports *architecture compliance by construction*, in order to proactively prevent architecture decay [32]. For example, to capture divergences, DCL supports constraints expressing that dependencies *only can*, *can only*, or *cannot* be established by specified modules. In addition, to capture absences, it is possible to specify constraints to check that particular dependencies *must* be present in the source code.

More specifically, we implemented `ArchFix` as an extension of `DCLcheck`, an Eclipse-based conformance tool that checks architectural constraints defined in DCL. As illustrated in Figure 2, `ArchFix` exploits some preexisting data structures available in `DCLcheck`, such as the graph of existing dependencies, the defined architectural constraints, and the detected violations. Moreover, `ArchFix` also reuses some auxiliary functions from `DCLcheck`, for example, a function that checks whether a type can establish a particular dependency with another type (function `can`, Table III).

`ArchFix` follows an architecture with three main modules:

- *Recommendation engine*: Based on the specification shown in Table IV, this module is responsible for suggesting the appropriate refactoring recommendation for a particular violation (when applicable). In fact, this module implements the Algorithm 1 presented in

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

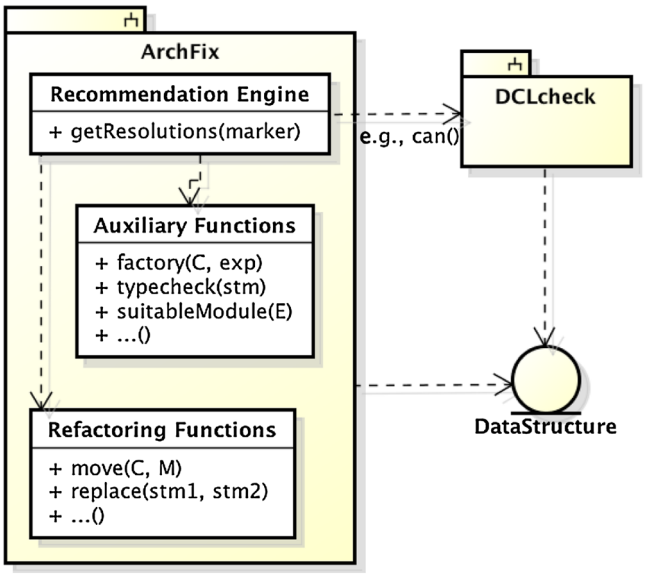

Figure 2. **`ArchFix`** architecture.

Section 3.4. In our current implementation, `ArchFix` was designed as a marker resolution because `DCLcheck` marks architectural violations in the source code. In other words, `ArchFix` is invoked whenever the developers request *Quick Fixes* for *problem markers*[§] that represent architectural violations.

As illustrated in Figure 2, the module `Recommendation Engine` inspects the recommendations using services from modules `Auxiliary` and `Refactoring Functions`. More specifically, the Recommendation Engine exports the public method `getResolutions` that receives a problem marker as input — which contains information on the architectural violation (e.g., violated constraint and the architecturally defective code)—and then returns a list of potential refactoring recommendations.

- *Auxiliary functions*: This module implements the auxiliary functions listed in Table III.
- *Refactoring functions*: This module is responsible for applying the refactorings listed in Table II. In the current stage of its implementation, `ArchFix` is mostly a recommendation engine, that is, the tool suggests refactorings to repair architectural violations. Nevertheless, `ArchFix` already supports some simple refactorings, such as replacing a type and removing annotations (which are only executed when the user accepts a given recommendation).

To illustrate `ArchFix`'s interface, assume an architectural constraint of the form `ControllerLayer cannot−declare HibernateDAO`.

Assume also a class `ProductController` that declares a variable of a type `ProductHibernateDAO`, that is, a class that introduces a violation in the form [`ProductController`, `cannot`, `declare`, `HibernateDAO`]. When the maintainer requests a recommendation to fix such violation, `ArchFix` indicates the most appropriate refactoring (Figure 3). The provided recommendation suggests replacing the declaration of the unauthorized type `ProductHibernateDAO` with its interface `IProductDAO` (which corresponds to recommendation $D1$ in Table IV). This recommendation is particularly useful to handle violations due to references to a concrete implementation of a service, instead to its general interface.

[§]A *problem marker* (an object of type `org.eclipse.core.resources.problemmarker`) represents an error or warning listed in the *Problems* view of the Eclipse integrated development environment.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

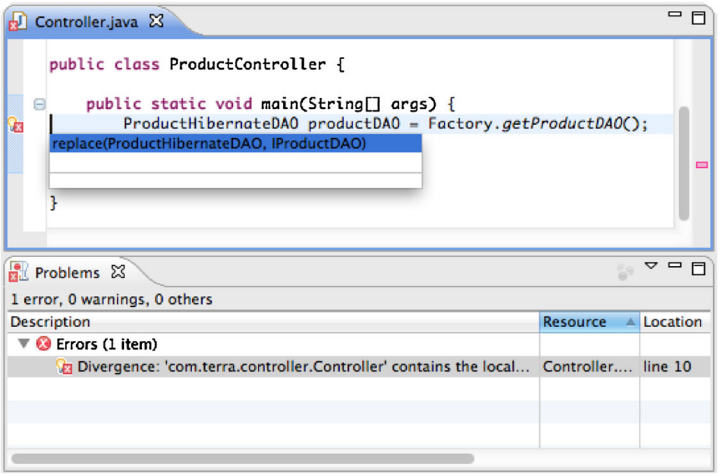

Figure 3. `ArchFix` interface.

5. EVALUATION

5.1. *Research questions*

We designed a study to address the following research questions:

- *RQ #1* – For what portion of architectural violations detected in a real-world system can the proposed approach provide refactoring recommendations?
- *RQ #2* – Does the proposed approach provide *correct* recommendations for repairing architectural violations?
- *RQ #3* – How *complex* is to discover a *correct* refactoring without the support of our recommendation system?
- *RQ #4* – How complex is to reject an *incorrect* refactoring provided by our recommendation system?

The strategies we follow to assert *correctness* (*RQ #2*) and to measure *complexity* (*RQ #3 and RQ #4*) are presented in Section 5.3 (respectively in Sections 5.3.2 and 5.3.3).

5.2. *Target systems*

Our evaluation relies on two Java-based systems. The first one is a 21 KLOC open-source strategic management system, called Geplanes.[¶] The system handles strategic management activities, including management plans, goals, performance indicators, actions, and so on. The second one is a large and complex customer care platform of a major Brazilian telecommunication company. Due to a nondisclosure agreement, we will omit the company's name in this paper and will refer to this second system just as BrTCom. The system has 728 KLOC, and it handles a full range of customer related services, including account activation, claims and inquiries, offers, and so on. Table V shows information on the size of both systems.

5.3. *Methodology*

To provide answers to our research questions, we performed the following major steps:

[¶]http://www.softwarepublico.gov.br

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table V. Target systems.

	Geplanes	BrTCom
LOC	21799	728814
Subsystems	1	146
Packages	25	2289
Classes	278	4724
Interfaces	1	1893
External libraries	47	58

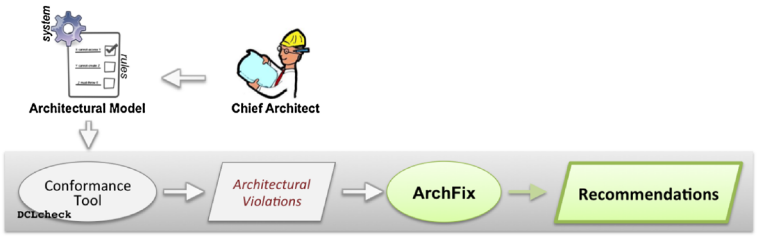

Figure 4. Methodology followed in the evaluation.

5.3.1. Triggering recommendations. As illustrated in Figure 4, the chief architect of each system first defined the architectural constraints based on an existing high-level model of their systems. Because the constraints were provided in natural language, we translated the informal definitions to DCL and validated them with the architects in a 30-min individual meeting.

Next, we executed the `DCLcheck` tool to detect violations in both systems. A second meeting was scheduled with the architects to validate the architectural violations raised by `DCLcheck`. Although Geplanes' meeting lasted 25 min, BrTCom's meeting lasted almost 1 h due to the considerable number of violations unrecognized by the architect. This additional time was necessary to refine the initial definitions of modules and to disregard violations that in fact represent exceptions to general rules or that are not relevant (e.g., violations detected in test classes). Finally, we executed `ArchFix` to provide refactoring recommendations for the true violations asserted by the architects.

5.3.2. Correctness evaluation. We showed the recommendations to the chief architect of each system who classified them as *correct*, *partially correct*, or *incorrect*. We instructed the architects to classify a recommendation as correct when it is the appropriate solution to the detected violation (e.g., a violation whose fixing involves replacing an instantiation with the respective factory method, and our approach has precisely suggested this refactoring), as *incorrect* when it is definitively not part of the architectural fix (e.g., a violation whose fixing involves making the class implement a particular interface, but our approach has suggested moving the class to another module), and as *partially correct* when the recommendation is only part of the required refactoring (e.g., a violation whose fixing involves replacing an annotation with a new one, but our approach has only suggested inserting the new annotation, without a suggestion to remove the existing one).

A third meeting was scheduled with the architects to evaluate the correctness of the triggered recommendations. Because Geplanes triggered few violations (only 41 in total), this classification was possible during a 30-min meeting. For each recommendation, we reminded the architect of the constraint, presented the code responsible by the violation, and the recommendation triggered by our approach. For BrTCom, due to the large number of detected violations in this system (787 in total), we grouped similar recommendations before the meeting and asked the architect to classify only a subset of the recommendations in a given group. For instance, a single constraint (named `TC11`) raised 270 violations due to forbidden declarations. Because the violations were very similar, we randomly selected six to be scored by the architect. In this way, in two 30-min meetings with the architect, it was possible to classify 89 recommendations that represent the whole set of recommendations triggered for the BrTCom system.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

5.3.3. Complexity evaluation. We conducted a 30-min meeting with the Geplanes' architect to assess the complexity of all triggered recommendations. In the case of BrTCom, similarly to the correctness evaluation, we asked the architect to assess the complexity of the subset of 89 recommendations in a single 50-min meeting.

During these meetings, for each correct recommendation, we asked the architects to assess how complex would be for a typical developer to discover the suggested refactorings without the support of our recommendation system. More specifically, we instructed the architects to consider the scope of the classes that might be inspected by the developers as the most relevant property to assess this complexity. We defined three levels of complexity: *minor* (when discovering the correct refactorings does not require inspecting other classes), *moderate* (when discovering the correct refactorings might require inspecting classes located in well-known modules), or *major* (when discovering the correct refactorings might require inspecting classes whose location is not known a priori).

As an example of a recommendation with *minor* complexity, assume a violation in which a `View` class misses a required annotation, and our approach correctly suggests adding the annotation. In this case, a typical developer can conclude after a local inspection that the class is indeed a `View` class and requires the annotation. On the other hand, as an example of a recommendation with *moderate* complexity, assume a violation in which an object of a `Product` class is created in a module, that is, not allowed to perform this operation, and our approach correctly suggests replacing the direct instantiation with the respective factory method. In this case, a typical developer needs to inspect other classes in the current module or even in the `Factory` module to find the appropriate factory method. Finally, as an example of a recommendation with major complexity, assume a violation in which a class is located in the wrong module, and our approach correctly suggests moving the class to another module. In this case, a typical developer may need to perform a system-wide inspection to determine the most suitable module.

Finally, we also asked the architects to assess how complex would be for a typical developer to discover that a given recommendation is incorrect. In this case, we also relied on the scope of the classes that might be inspected by the developers to make the decision.

5.4. Geplanes results

The results achieved by applying our methodology to Geplanes are discussed next.

Recommendations (RQ #1). Table VI lists the architectural constraints prescribed by the Geplanes' chief architect. Figure 5 illustrates such constraints and the detected architectural violations in the form of a reflexion model [11].[‖] They are mainly related to the model-view-controller-based framework used by Geplanes' current implementation. In general, constraints `GP1`—`GP7` require that classes from some modules receive particular annotations, constraints `GP8`—`GP9` restrict the modules that are allowed to receive particular annotations, and constraints `GP10`—`GP11` forbid some modules to create classes of specific modules. As reported in Table VI, we have found 41 architectural violations, and `ArchFix` was able to provide recommendations for all of them.

Correctness (RQ #2). Table VI also presents the results of the correctness classification, according to Geplanes' architect. As reported, 31 recommendations were classified as *correct* (75%), eight recommendations were classified as *partially correct* (20%), and two recommendations were classified as *incorrect* (5%).

As reported in Table VI, most violations are related to constraints `GP1`—`GP7`. In such cases, the usual recommendation was adding the required annotation to the class or method where the violation was detected (recs. *A*6 and *A*8). For example, as can be observed in Code 2, class `AnomaliaCrud` is missing annotation `@Controller` required by the underlying model-view-controller-based framework. Therefore, this absence represents a violation of constraint `GP6`, as illustrated by an

[‖] We are using this reflexion model just for illustrative purposes. In fact, as mentioned in Section 5.3.1, we relied on DCL to detect architectural violations in our target systems.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table VI. Recommendations and correctness evaluation (Geplanes).

Violations		Number	Recommendations (correct – pr. cor. – incor.)	Total
GP1	[Entities, must, useannotation, javax.persistence.Entity]	3	A5(1–0–1); A6(1–0–0)	2–0–1
GP2	[Entities, must, useannotation, javax.persistence.Id]	1	A8(1–0–0)	1–0–0
GP3	[Entities, must, useannotation, javax.persistence.GeneratedValue]	1	A8(1–0–0)	1–0–0
GP4	[Entities, must, useannotation, linkcom.neo.bean.DescriptionProperty]	18	A6(18–0–0)	18–0–0
GP5	[Controllers, must, useannotation, linkcom.neo.controller.DefaultAction]	1	A6(1–0–0)	1–0–0
GP6	[Controllers, must, useannotation, linkcom.neo.controller.Controller]	2	A6(1–0–1)	1–0–1
GP7	[Services, must, useannotation, linkcom.neo.bean.ServiceBean]	1	A6(1–0–0)	1–0–0
GP8	[$\overline{\text{Entities}}$, cannot, useannotation, javax.persistence.Entity]	1	D21(1–0–0)	1–0–0
GP9	[$\overline{\text{Controllers}}$, cannot, useannotation, linkcom.neo.controller.Input]	1	D24(1–0–0)	1–0–0
GP10	[System, cannot, create, [Services, DAOs, Controllers]]	5	D12(2–3–0)	2–3–0
GP11	[DAOs, cannot, create, linkcom.neo.persistence.QueryBuilder]	7	D11(2–0–0); D13(0–5–0)	2–5–0
		41		31–8–2

Pr. cor.: partially correct; incor.: incorrect.

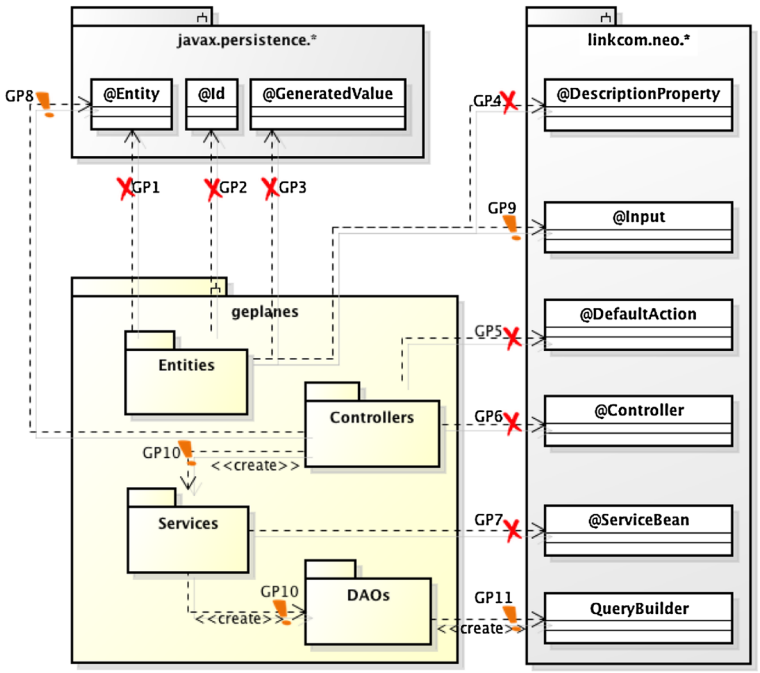

Figure 5. Geplanes' reflexion model (an '*x*' denotes absences and a '!' denotes divergences).

arrow with an '*x*' mark from `Controllers` to `@Controller` in Figure 5. In this particular case, our approach correctly suggested adding annotation `@Controller` (rec. *A6*) to the `AnomaliaCrud` class.

```
1  package br.com.linkcom.sgm.controller.crud;
2
3  public class AnomaliaCrud extends SGMCrudController<AnomaliaFiltro, Anomalia, Anomalia> {
4      private FerramentaAnomaliaService fas = new FerramentaAnomaliaService();
5      ...
6  }
```

Code 2: Controller class `AnomaliaCrud`

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

```
1  package br.com.linkcom.sgm.controller;
2
3  @Entity
4  @SequenceGenerator(name = "sq_verbo", sequenceName = "sq_verbo")
5  public class Verbo {
6      private Integer id;
7      private String nome;
8      ...
9      @Id
10     @GeneratedValue(strategy=GenerationType.AUTO, generator="sq_verbo")
11     public Integer getId() {
12         return id;
13     }
14 }
```

Code 3: *"Controller"* class `Verbo`

`ArchFix` also suggested removing instantiations of objects that are provided by dependency injection techniques (rec. $D12$). For example, the `AnomaliaCrud` class (Code 2) creates an object of `FerramentaAnomaliaService` (line 4). This instantiation represents a divergence regarding constraint `GP10`, as illustrated by an arrow with an '!' mark from `Controllers` to `Services` in Figure 5.

This particular divergence was regarded as *partially correct* because, besides removing the instantiation, the architect also indicated the need to create a local setter method for the respective attribute `fas`.

As a final example, class `Verbo`—located in module `Controllers`—has an annotation `@Entity`, as can be observed at line 3 in Code 3. The use of this annotation represents a divergence regarding constraint `GP8`, as illustrated by an arrow with an '!' mark from `Controllers` to `@Entity` in Figure 5. In this case, our approach correctly suggested moving `Verbo` to module `Entities` (rec. $D21$). More specifically, the `suitable_module` function returned `Entities` as the module with the highest similarity —$sim(\texttt{Verbo}, \texttt{Entities}, *) = 0.4216$—, whereas the current module of the class (`Controllers`) has a significant lower similarity—$sim(\texttt{Verbo}, \texttt{Controllers}, *) = 0.0763$.

Complexity (RQ #3, RQ #4). In the case of correct recommendations, two recommendations have been scored as having a *minor* complexity ($A5$ and $A6$), four recommendations ($D11$, $D21$, $D24$, and $A8$) as having a *moderate* complexity, and another recommendation ($D12$) as having a major complexity. For example, rec. $A5$ for violations `GP1` was classified as having a *minor* complexity, because it just requires adding an annotation to classes that represent database entities (which can be inferred by the presence of other database annotations). On the other hand, rec. $D11$ for violations `GP11` was classified as having a *moderate* complexity, because it requires replacing the direct instantiation of a query builder object with its factory method. In this case, the developer should search for the factory inspecting classes located in the `DAO` module (i.e., a previously known module). Finally, rec. $D12$ for violations `GP10` has been scored as *major* complexity. Although it solely suggests to remove the instantiation of `DataSource` and `Controller` objects, the developer must be aware that such objects are provided by a dependency injection framework.

In the case of incorrect recommendations, rec. $A5$ has been considered as having *moderate* complexity and rec. $A6$ as presenting a *minor* complexity. Rec. $A5$ suggests moving a class without a particular annotation to other module. In this case, the developer must inspect the classes of the target module to realize that the moving is incorrect. On the other side, rec. $A6$ suggests adding a specific annotation and the developer can infer that this annotation is incorrect by considering only the class itself.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table VII. Recommendations and correctness evaluation (BrTCom).

Violations		Number	Recommendations (correct – pr. cor. – incor.)	Total
TC1	[DTOs, must, implement, java.io.Serializable]	63	A3(50–0–0); A4(0–0–13)	50–0–13
TC2	[SAOs, must, extend, brtcom.server.sao.AbstractSAO]	1	A4(0–1–0)	0–1–0
TC3	[Controllers, must, useannotation, brtcom.client.controller.Controller]	1	A6(1–0–0)	1–0–0
TC4	[DataSources, must, useannotation, brtcom.client.datasource.DataSource]	1	A6(1–0–0)	1–0–0
TC5	[$\overline{\text{brtcom.server.dao.BaseJPADAO}}$, cannot, create, DAOs]	13	D11(13–0–0)	13–0–0
TC6	[$\overline{\text{DAOs}}$, cannot, throw, brtcom.server.dao.DAOException]	15	D15(11–0–0); D16(2–0–0)	13–0–0
TC7	[[$\overline{\text{CtrlLayer, DSLayer}}$], cannot, useannotation, CtrlDSAnnotations]	20	D21(2–0–0); D22(18–0–0)	20–0–0
TC8	[[$\overline{\text{Services, SAOLayer}}$], cannot, depend, SAOs]	5	D20(1–0–0)	1–0–0
TC9	[System, cannot, create, [Controllers, DataSources]]	3	D12(3–0–0)	3–0–0
TC10	[CtrlLayer, cannot, create, java.util.Date]	84	D13(0–84–0)	0–84–0
TC11	[ScreenWrappers, cannot, useannotation, JavaLangAnnotations]	18	D21(0–0–5); D22(13–0–0)	13–0–5
TC12	[System, cannot, depend, java.lang.System]	14	D9(14–0–0)	14–0–0
TC13	[ServicesAsync, cannot, declare, UnallowedAbstractTypes]	270	D2(270–0–0)	270–0–0
TC14	[ServerLayer, cannot, depend, ClientUtil]	279	D7(225–0–0)	225–0–0
		787		624–85–18

Pr. cor.: partially correct; incor.: incorrect.

5.5. *BrTCom results*

The results for BrTCom are discussed next.

Recommendations (RQ #1). Table VII lists the 14 architectural constraints prescribed by the BrTCom's chief architect. These constraints are mainly used to enforce several architectural rules, such as decomposition in layers (e.g., TC8), factories (e.g., TC5), interfaces (e.g., TC13), persistence patterns (e.g., TC1), and so on. ArchFix raised 727 recommendations for the 787 violations we have detected using the constraints in Table VII (92%). Regarding the 60 violations without recommendation, 54 violations are associated to TC14. Particularly in such violations, ServerLayer classes were calling methods that should not be implemented in ClientUtil classes but rather in a layer common both to server and client modules.

Correctness (RQ #2). As can be observed in Table VII, BrTCom's architect has scored 624 recommendations as *correct* (79%), 85 recommendations as *partially correct* (11%), and 18 recommendations as *incorrect* (2%).

For example, constraint TC1, which specifies that DTO classes must implement Serializable, raised violations in 63 classes. For 50 classes, ArchFix suggested the correct refactoring, that is, making the class implement Serializable (rec. $A3$). Nevertheless, Data Transfer Objects (DTOs) by their nature rely extensively on types from the Java API. For this reason, ArchFix—based on the most suitable module calculated by the sim function described in Section 3.5—has improperly recommended moving the other 13 classes to the Constants module (rec. $A4$), whose classes are also heavily based on Java's built-in types.

The highest number of correct recommendations for a single constraint has been raised for the 270 violations associated to constraint TC13, which forbids ServicesAsync classes to declare abstract types due to a pattern recommended by the GWT framework. For each of such violations, ArchFix suggested a more specialized type (rec. $D2$). For instance, most of the violations were due to references to List and Map in GWT interfaces. In such cases, ArchFix has properly suggested replacing these abstract types with the concrete types ArrayList and HashMap, respectively.

Complexity (RQ #3, RQ #4). BrTCom's architect has assessed the complexity of correct recommendations in the following way: four recommendations as having a *minor* complexity ($D9$, $D22$, $A3$,

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table VIII. Classification of the detected violations (focusing on correctness).

Architectural defect type	Constraints	# viols.	# recs.	# correct recs.	Precision
Misusage of persistence patterns	GP1−3; TC1	68	68	54	0.79
Misusage of domain-specific patterns	GP4−7; TC2−4	25	25	23	0.92
Bypassing layers	GP8−9; TC6−8; TC14	321	261	261	1.00
Unintended dependencies	GP10; TC9; TC11−12	40	40	32	0.80
Bypassing creational patterns	GP11; TC5	20	20	15	0.75
Context exploration	TC10	84	84	0	0.00
Misusage of interfaces	TC13	270	270	270	1.00
		828	768	655	

and $A6$), two recommendations ($D2$ and $D11$) as having a *moderate* complexity, and six recommendations ($D7$, $D12$, $D15$, $D16$, $D20$, and $D21$) as presenting a *major* complexity. For example, rec. $D2$ for violations `TC13` has been considered as having a *moderate* complexity, because the concrete implementations for an abstract type usually have a well-known location (e.g., the package `java.util`). On the other hand, rec. $D7$ for `Server` classes making an unauthorized use of `Client` services associated to constraint `TC14` has been scored as having a *major* complexity, because it is not trivial to delimit the scope of the task of searching for a class that provides equivalent services to the ones provided by another class.

Regarding the incorrect recommendations, one recommendation ($D21$) has been considered as having a *major* complexity and another recommendation ($A4$) as presenting a *moderate* complexity. Rec. $D21$ suggests moving an incorrectly annotated class to another module (where the annotation does not represent a violation). The task of discarding it has been considered as having a *major* complexity, because a global understanding of the system is required to decide whether the indicated module is correct or not.[**]

5.6. Qualitative discussion

Based on the experience gained with the Geplanes and BrTCom case studies, we conducted a deeper analysis on the major themes of the detected violations and the root causes of them. Table VIII classifies the detected violations according to a set of architectural defect types initially proposed by Knodel and Popescu [8] and later extended by us [16]. As a general fact, the architects ascribed most of the detected violations to the lack of awareness about the architectural model [33] and *copy-and-paste* procedures [34].

Next, we rely mainly on this classification to answer our research questions.

RQ #1 – For what portion of architectural violations detected in a real-world system can the proposed approach provide refactoring recommendations?

As reported in Table VIII, our approach was able to trigger recommendations for 768 out of 828 detected violations (93%).

Particularly, we were able to provide recommendations for all architectural defect types, with the exception of 60 constraints whose violations were classified in the *bypassing layers* category. However, we argue that our approach can be improved to handle some of the violations in this category. As an example, we noticed that an additional recommendation suggesting the replacement of an exception with another one could correctly repair the two violations without recommendations in the case of constraint `TC6`. As another example, BrTCom's architect mentioned that an additional

[**] A similar recommendation ($A5$) was triggered in Geplanes' case study, and it was considered as a having *moderate* complexity. Basically, the recommendations involves moving a Data Transfer Object, a well-known design pattern. On the other hand, rec. $D21$ triggered in BrTCom involves moving a domain-specific class, which explains its higher complexity.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

recommendation suggesting to move the accessed class or method might address the 54 violations without recommendations in the case of constraint `TC14`. Currently, `ArchFix` only suggests moving the class or method where the violations have been detected. For example, in Code 1, we can suggest to move the method `init` to another class, but currently we do not suggest moving the method `foo`. In short, our proposed approach handles the vast majority of the detected violations and further significant improvement are possible and some of them seem to be general to object-oriented systems.

RQ #2 – Does the proposed approach provide correct recommendations for repairing architectural violations?

As can be observed in Table VIII, our approach triggered correct recommendations for 655 out of 828 detected violations (79%). More specifically, we achieved a precision greater than 75% for all architectural defect types, with the exception of the constraints whose violation were classified in the *context exploration* category. We define *precision* as the number of correct recommendations by the total number of recs. triggered by `ArchFix`.

According to the architects' feedback, `ArchFix` was very precise handling the architectural defect types that do not involve complex refactorings. As evidence, BrTCom's architect highlighted the 270 violations on constraint `TC13` (*misusage of interfaces*) fixed by recommendation *D*2— which suggests replacing the declaration of an unauthorized abstract type (mostly, `List`) with one of its concrete subtypes (mostly, `ArrayList`)— as one of the 'most important recommendations'. According to the architect, by not following this recommendation (associated to the correct use of the GWT framework), the current implementation experiences an overhead in the size of the generated JavaScript code, with important negative consequences both in terms of CPU performance and network bandwidth consumption. On the other hand, `ArchFix` was unable to achieve good precision on violations related to *context exploration*. For example, constraint `TC10` defines that `Client` classes cannot create `Date` objects (to avoid time synchronization bugs). As a result of the conformance process, we found 84 violations of this constraint in BrTCom classes. However, the recommendations suggested by `ArchFix` just include the removal of the instantiations. This refactoring was classified by BrTCom's architect as partially correct. In fact, he indicated that the correct repair action in this case would require a refactoring in the `Server` interfaces to return `Date` instances in particular cases. Therefore, instead of creating the `Date` on the client process, the client code should invoke the refactored interfaces.

As another relevant finding, we noticed that some incorrect recommendations were triggered because the `suitable_module` function did not indicate the current (and also correct) module as the most suitable one. For instance, in the case of the five violations in constraint `TC11` due to *unintended dependencies*, our approach failed to indicate the most suitable module. However, the module calculated with the second best similarity was in fact the correct one. For this reason, we are considering a revision in our `suitable_module` implementation to indicate more than one module whenever the similarity value is very close to the highest calculated one, or even boost the priority of the current module.

Last but not least, as a practical contribution of our evaluation, the architects of both systems opened a maintenance request in the issue management platform of their systems requesting a correction for the detected violations and suggesting the use of the recommendations provided by `ArchFix`. Particularly, Geplanes' maintainers have already repaired all detected violations in the system's main development trunk.

RQ #3 – How complex is to discover a correct refactoring without the support of our recommendation system?

As reported in Table IX, most correct recommendations were scored as having *moderate* or *major* complexity. Particularly, 241 violations related to *bypassing layers* were the ones classified predominantly as having a *major* complexity. After asking the architects for clarification, we realized that repairing these violations requires a global understanding of the system, including knowledge on

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table IX. Classification of the detected violations (focusing on complexity).

	Complexity					
	Correct recs.			Incorrect recs.		
Architectural defect type	Minor	Moderate	Major	Minor	Moderate	Major
Misusage of persistence patterns	52	2	—	—	14	—
Misusage of domain-specific patterns	23	—	—	1	—	—
Bypassing layers	18	2	241	—	—	—
Unintended dependencies	27	—	5	—	—	5
Bypassing creational patterns	—	15	—	—	—	—
Context exploration	—	—	—	—	—	—
Misusage of interfaces	—	270	—	—	—	—
	120	289	246	1	14	5

the public interfaces of all layers. On the other hand, recommendations associated to the *misusage of persistence* and other *domain-specific patterns* have been mostly classified as having a *minor* complexity. They are usually associated to missing or incorrect use of annotations, which can be fixed more easily, by just inserting or removing a given annotation.

RQ #4 – How complex is to reject an incorrect refactoring provided by our recommendation system?

As also reported in Table IX, the results indicate that rejecting an incorrect recommendation has a higher complexity than accepting a correct one (19 out of 20 incorrect recs. require a *moderate* or *major* effort). However, we argue that the number of correct recommendations raised by our approach outperforms by a large margin, the number of incorrect ones (655 vs 20 recs. in our case studies).

5.7. Lessons learned

We identified five lessons learned from our experience on evaluating the use of `ArchFix` in the Geplanes and BrTCom systems. We learned that we could have avoided many incorrect recommendations if the `suitable_module` function indicated a set of modules with close similarity values, rather than a single module. As a related issue, we noticed that for some violations, the correct fix was a recommendation different than the triggered one. For this reason, we also argue that our current prioritization policy can be possibly changed to raise multiple recommendations, with an associated confidence, for particular violations.

Second, the architects highlighted that `ArchFix` does not target senior developers but mainly developers who recently joined the project. They typically refer to the lack of awareness about the architectural rules as the main cause of the violations. According to the architects, our approach can help less experienced developers to understand the system architecture by showing the correct repair procedure for the detected violations.

Third, the real value of conformance checking is to be part of the regular development and maintenance processes. As evidence, Knodel *et at.* demonstrate that teams supported by constructive compliance checking may insert about 60% less structural violations into the architecture [32]. In this paper, we claim that an architectural repair recommendation system—such as `ArchFix`—can also be integrated into the regular processes. In such way, besides the detection of points of violations, the regular process would also contemplate a mechanism to repair the detected violations.

Fourth, only 17 out of the 32 proposed recommendations, which emerged from our initial study [16], have been triggered in our case studies. However, we believe that the unused recommendations are generic enough to be triggered in other architecture erosion fixing contexts. For example, the unused recommendation $D1$ addresses violations due to the *misusage of interfaces*,

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

which were commonly detected in other case studies [7, 8, 16]. In fact, the unused recommendations instigate new case studies using `ArchFix`.

Last but not least, most recommendations were scored as having *moderate* or *major* complexity. Therefore, we claim that our approach may save developers' time when fixing architectural violations, although we have not measured this aspect in our evaluation. Instead, we assumed that the more complex the recommendation, more time is required to discover and to apply the suggested refactorings.

5.8. Threats to validity

Next, we identify threats to validity in our evaluation.

Internal validity. Because many steps of our evaluation require the involvement of architects, they could be affected negatively (e.g., tired or bored) during the experiment. In order to minimize this threat, we conducted one meeting for each task: constraints definition, violations validation, correctness assessment, and complexity assessment. More important, we carefully planned each meeting to last up to 50 min.

External validity. First, although we have used two industrial-strength web systems that have different architectures and constraints, we cannot claim that our approach will provide equivalent results in other systems (as usual in empirical studies in software engineering). Second, because the target systems presented a moderate number of violations (slightly over 1 violation/KLOC), we cannot claim that our approach provides the same precision in systems already facing a major architectural erosion process. Fundamentally, a major erosion process may impact the precision of basic functions, such as `suitable_module`. Third, because we have detected violations using DCL, we cannot claim that our approach provides equivalent coverage and precision when using other architecture conformance tools. More important, the proposed approach targets specific classes of violations, and many other types of violations may be present in a particular system. However, DCL was able to express all constraints proposed by the architects of two large and complex systems (BrTCom, as reported in this paper, and SGP, our training system, as reported in Section 3.1). Fourth, because we have used half of the proposed refactoring recommendations, it was not possible to evaluate the remaining recommendations. However, the unused recommendations would have been useful at least once in our training system [16].

Construct validity. In our evaluation, we relied on two chief architects (one per system) to define the constraints, to validate whether the detected violations are in fact true positives, to judge the correctness, and to assess the complexity of the recommendations. As typical in human-based classifications, our results might be affected by some degree of subjectivity. However, it is important to highlight that we interviewed the chief architects who designed the evaluated architectures, and are responsible for their maintenance and evolution. Therefore, they are the right experts to evaluate the correctness of a given recommendation. Moreover, our assessment of complexity is based on a fairly precise definition, which relies on the scope of the classes that might be inspected. Furthermore, instead of assessing each violation separately, we grouped similar recommendations to make the evaluation easier to the architects. More important, the architects' answers for the recommendations in the same group were always consistent.

6. RELATED WORK

We divided the related work into five groups: (i) *architectural models*, which describes different architectural views, including the one this paper is based on; (ii) *architecture conformance*, which is related to our work in terms of providing the input of our approach; (iii) *identification of refactoring opportunities*, because our approach relies on primitive refactorings to repair architectural violations; (iv) *remodularization approaches*, because they are potential alternatives to architecture

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

degradation; and (v) *recommendation systems*, because our approach is based on recommendation principles.

Architectural models. Kruchten defines software architecture using the following five concurrent views: *logical view* (which defines the conceptual decomposition of the system), *process view* (which partitions the software in independent tasks), *physical view* (which maps software tasks to hardware elements), *development view* (which partitions a system into physical modules or subsystems), and *use case view* (which defines the functions provided to the users by listing use cases) [28]. Each view addresses concerns of interest to different stakeholders. Particularly, our work is centered on the *development view* (a.k.a. *module view*). In practice, there are studies that refer to the development view as a high-level software model [11]. Similarly, there are studies that ascribe the violations tackled by our approach as due to *design erosion* rather than *architectural erosion* [4, 13].

Architecture conformance. Over the past decade, several techniques have been proposed to deal with the architecture erosion problem [9]. For instance, Silva and Balasubramaniam indicated a combination of strategies that might help to control architecture erosion [3]. Particularly, they argue that an *architecture restoration* strategy—such as `ArchFix`— should complement an *architecture conformance* strategy in order to extend the lifetime of the software.

Reflexion models (RMs) are based on the comparison of two models—representing a high-level and a low-level view of the target system [11, 35]. As a result, the technique highlights the detected differences in terms of *divergences* and *absences* in what the authors called a *reflexion model*. There are commercial tools that are based on RM principles [5, 6, 32]. Furthermore, there are also several works extending the original RM to support, for example, hierarchical structures [36], behavioral design [37], and software variants [38, 39].

Dependency structure matrices (DSMs) can be used to provide a scalable view of the established dependencies among classes of a system [40, 41]. A DSM is a weighted square matrix whose both rows and columns denote classes from an object-oriented system. The Lattix Dependency Manager (LDM) tool[††] provides a simple language to declare design rules that the target system implementation must follow (e.g., `A cannot-use B`) and visually represents the detected violations in a DSM. Currently, `ArchFix` relies on architectural violations detected by DCL [16, 31]. However, we claim that it would be straightforward to adapt our approach to the aforementioned conformance techniques and tools.

Identification of refactoring opportunities. Tsantalis and Chatzigeorgiou have proposed a semiautomatic approach to identify Move Method refactoring opportunities [42]. Their general goal is to tackle coupling and cohesion anomalies manifested in the form of the *Feature Envy* bad smell [29]. Similarly to our work, they employed the notion of Jaccard distance between an entity (attribute or method) and a class. Later, using an adaptation of program slicing techniques, they extended their tool to identify Extract Method refactorings [43].

O'Keeffe and Ó Cinnéide have proposed a search-based software maintenance tool that relies on search algorithms, such as hill climbing and simulated annealing, to suggest six inheritance-related refactorings [44]. They evaluated their approach in two packages of the `spec.benchmarks` and demonstrated that some programs can be automatically refactored to improve quality in terms of flexibility, reusability, and understandability.

In general terms, the ultimate goal of the aforementioned tools is to suggest refactorings that improve the internal quality of the code—for example, in terms of coupling and cohesion. On the other hand, the refactoring recommendation system we described in this paper aims to help developers to handle violations exposed as the result of an architecture conformance process.

[††]http://www.lattix.com

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Remodularization approaches. When architecture erosion is neglected over the years, it can reduce the architecture to a set of strongly coupled and weakly cohesive components [45]. At a certain point, architecture conformance and repair techniques—such as `ArchFix`— present themselves mostly ineffective, and a complete remodularization can be the only solution [46].

Rama and Patel analyzed several remodularization efforts in order to define recurring modularization operators [47]. More specifically, they formalized the following operators: module decomposition/union, file/function/data transfer, and function to API promotion. Their case study on Linux shows that some operators are applied as the software evolves, and therefore, they argue that modularization is not necessarily a one shot process. Analogously, `ArchFix` can be continuously applied as the system evolves. However, in contrast to our approach, Rama and Patel do not provide tool support for applying the proposed operators.

Hierarchical clustering is another technique commonly proposed to evaluate alternative software decompositions [48, 49]. However, the effectiveness of clustering in reengineering tasks is often challenged. For example, using Eclipse as case study, Anquetil *et al.* reported that restructurings manually performed by developers do not necessarily improve modularity in terms of cohesion/coupling [50]. Therefore, this finding undermines the validity of clustering techniques that aim to minimize coupling and maximize cohesion. On the other hand, instead of quality metrics, `ArchFix` relies on structural similarity among source code entities, as computed by the `suitable_module` function.

Moghadam *et al.* have proposed a remodularization approach that automatically refactors the source code of a system toward a desired design, provided in the format of a UML class diagram [44]. The proposed approach automatically compares the current and desired models and express the *design differences* as a set of *detected refactorings*. In contrast, instead of a desired UML model, our approach aims to respect a set of constraints that defines an architectural model. Moreover, our evaluation was conducted in a real context using two industrial-strength systems.

In conclusion, most of the difficulties faced during remodularizations are caused by the accumulation of the architectural erosion process over the years. On the other hand, our approach aims to prevent large restructurings by providing suggestions to repair a violation as soon as it is detected.

Recommendation systems. Recommendation systems for software engineering (RSSEs) are an emerging research area [22]. For example, current RSSEs can recommend relevant source code fragments to help developers to use frameworks and APIs (CodeBroker [51], PARSEWeb [52], and Strathcona [25]), software artifacts that must be changed together (eRose [26]), parts of the software that should be tested more cautiously (Suade [53]), and replacement methods for adapting code to a new library version (SemDiff [27]). However, RSSEs usually present little or no information about the consequences of the recommended changes. An exception is the system proposed by Muşlu *et al.* to inform developers on the consequences of code transformations [54]. They have built an Eclipse plug-in—called *Quick Fix Scout*—that augments the Quick Fix dialog by adding the number of compilation errors that remain after each proposal's application (a technique they called speculative analysis). Their experiments demonstrate that developers complete compilation-error removal tasks 10% faster when using their tool. Although their technique originally focuses on compilation errors, the proposed idea can also be used to prioritize refactoring recommendations when more than one can be triggered to repair an architectural violation.

7. CONCLUSION

Architectural erosion is a recurrent problem faced by software architects. However, a clear dichotomy is perceived in the tools already designed to tackle this problem. On the one hand, there are several approaches and commercial tools proposed to uncover architectural violations [11, 12, 14, 16, 17]. On the other hand, the task of fixing the hundreds of violations raised after an architecture conformance process is normally conducted with limited tool support. To address this shortcoming, we described a recommendation system that provides refactoring guidelines for maintainers and developers when repairing architectural violations. The proposed system provides

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

recommendations for violations—divergences and absences—raised by static architecture conformance checking approaches. Furthermore, we conducted an evaluation with two industrial-strength systems that provided us with encouraging feedback on the applicability and correctness of our recommendations. Considering both systems, `ArchFix` has indicated the correct refactoring for 655 (79%) out of 828 violations detected as the result of an architecture conformance process. Moreover, the architects scored 82% of these recommendations as having moderate or major complexity.

Ideas for future work include (i) refining and extending our current list of recommendations and considering the possibility to raise multiple recommendations, with an associated priority, for particular architectural violations; (ii) evaluating `ArchFix` on other systems; (iii) designing an architecture-repair recommendation language, in order to allow maintainers to extend `ArchFix` with their own domain-specific refactorings; (iv) conducting a crossover study to compare the effort required to apply the correct refactoring tasks with and without `ArchFix`; (v) refining the suitable module heuristic; and (vi) improving our current prioritization heuristic through elements of a learning system or by means of speculative analysis [54].

The `ArchFix` tool—including its complete source code—is publicly available at `http://github.com/rterrabh/DCL`.

ACKNOWLEDGEMENTS

Our research has been supported by CAPES, FAPEMIG, and CNPq. We would like to thank the software architects Gessé Dafé (BrTCom) and Rógel Garcia (Geplanes) for the valuable collaboration in the case studies.

APPENDIX

In this appendix, we provide detailed descriptions and illustrative examples on the nontrivial functions used by `ArchFix` to recommend architectural refactorings. Appendix A addresses nontrivial *refactoring* functions as introduced in Table II, whereas Appendix B addresses nontrivial *auxiliary* functions as introduced in Table III.

APPENDIX A: REFACTORING FUNCTIONS

Function `promote_param(f, v, exp)`: Promotes variable `v` to a formal parameter of method `f`; `exp` is the corresponding argument in the calls to `f`. To illustrate, assume a constraint in the form `Facade cannot−access Foo`, but `Facade` is allowed to *declare* `Foo` as presented in Figure A.1*(before)*. Therefore, a violation [`Facade, cannot, access, Foo`] occurs at line 9 in this figure.

Assuming that `MainClass` is allowed to access `Foo`, a potential refactoring to repair such violation may be defined as follows: `promote_param( setup(Foo), d, [foo.getDate()] )`. As can

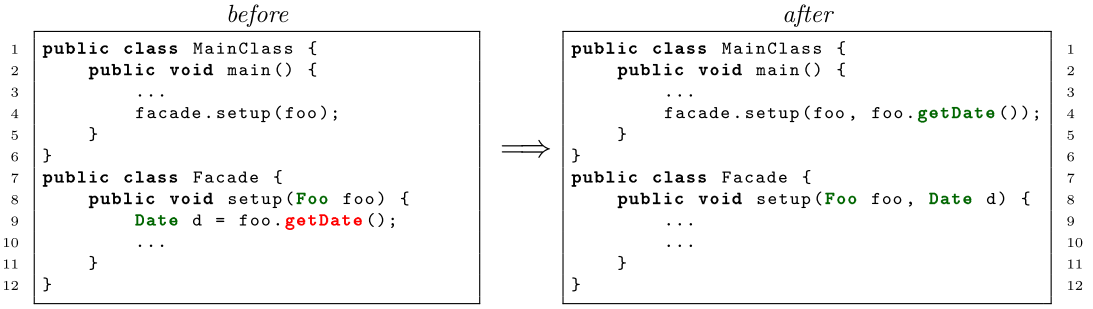

Figure A.1. `promote_param( setup(Foo), d, [foo.getDate()] )`.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

be observed in Figure A.1*(after)*, after this refactoring, (i) variable `d` is promoted to a formal parameter of `setup(Foo)` (line 8), and (ii) calls to `setup(Foo, Date)` are adjusted to include the actual parameter `foo.getDate()` (line 4).

Function `inline(exp, v, S)`: inlines `exp` in the uses of variable `v` in the block of code `S`. To illustrate, assume a constraint in the form `Bar cannot−declare Foo`, but it is allowed to *access* `Foo` as presented in Figure A.2*(before)*. Therefore, a violation `[Bar, cannot, declare, Foo]` occurs at line 3 in this figure.

before

```
1 public class Bar {
2     public void init(Context ctx) {
3         Foo foo = ctx.getFoo();
4         int x = foo.getX();
5         int y = foo.getY();
6         ...
7     }
8 }
```

⟹

after

```
1 public class Bar {
2     public void init(Context ctx) {
3
4         int x = ctx.getFoo().getX();
5         int y = ctx.getFoo().getY();
6         ...
7     }
8 }
```

Figure A.2. `inline( [ctx.getFoo()], foo, S )`.

Because class `Bar` is allowed to *access* `Foo`, a potential refactoring to repair such violation is as follows: `inline( [ctx.getFoo()], foo, S )`, where `S` is the body of method `init(Context)`. As can be observed in Figure A.2*(after)*, this refactoring removes the declaration of variable `foo` (line 3) and inlines the expression `ctx.getFoo()` to its previous uses (lines 4–5).

Function `unwrap_return(f, T, exp)`: Considering a method `f` that returns `new T(exp)`, this function modifies this statement to just return `exp` and moves the instantiations of the wrapper type `T` to the respective call sites. To illustrate, assume a constraint in the form `Model cannot−create Foo` as presented in Figure A.3*(before)*. Therefore, a violation `[Model, cannot, create, Foo]` occurs at line 10 in this figure.

before

```
1  public class MainClass {
2      public void main() {
3          Foo foo = m.retrieve();
4          ...
5      }
6  }
7  public class Model {
8      public Foo retrieve() {
9          ...
10         return new Foo(obj);
11     }
12 }
```

⟹

after

```
1  public class MainClass {
2      public void main() {
3          Foo foo = new Foo(m.retrieve());
4          ...
5      }
6  }
7  public class Model {
8      public ObjType retrieve() {
9          ...
10         return obj;
11     }
12 }
```

Figure A.3. `unwrap_return( retrieve(), Foo, [obj] )`.

Assuming that class `MainClass` is allowed to *create* `Foo`, a potential refactoring to repair this violation is as follows: `unwrap_return( retrieve(), Foo, [obj] )`. As can be observed in Figure A.3*(after)*, (i) the return statement is refactored to return just `obj` (line 10); (ii) the return type of `retrieve` is refactored to `ObjType` (i.e., the type of `obj`) (line 8); and (iii) an object of `Foo` is created to wrap the results returned by the calls to `retrieve` (line 3).

APPENDIX B: AUXILIARY FUNCTIONS

Function `delegate(f)`: searches a delegate method for `f`, i.e., a method that just encapsulates a call to `f` [55]. Basically, our heuristic to find delegate methods consists in finding a method that (i) only invokes method `f` and (ii) returns the same type of `f`. For instance, as presented in Figure B1, `Persistence::persist(Bar)` is a delegate method for `Bar::save(Connection)` because it only forwards the call (line 5).

Function `factory(C, exp)`: searches for a factory method for class `C`, accepting `exp` as input. Basically, our heuristic to find a factory method for a class `C` consists in finding a method that

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

```
1  public class Persistence {
2      Connection conn = ...;
3      ...
4      public int persist(Bar bar) {
5          return bar.save(conn);
6      }
7  }
```

Figure B1. `delegate(Bar::save(Connection))` = `Persistence::persist(Bar)`.

just returns an object of type `C` created using `exp`. For instance, as presented in Figure B2, `DAOFactory::getBar(int)` is a factory method for `Bar` (lines 3–5).

```
1  public class DAOFactory {
2      ...
3      public int getBar(int max) {
4          return new Bar(max);
5      }
6  }
```

Figure B2. `factory(Bar, {[5]})` = `DAOFactory::getBar(int)`.

Function `gen_decl(f)`: returns a declaration `D` for a variable `c` of type `C`, where `C` is the class that defines the method `f`. The simplest scenario happens when `f` is a static method and this function returns the static reference to the class `C`. However, when `f` is not static, this function proceeds as follows: (i) if `C` is a singleton, it uses the `getInstance` method, for example, `C c = C.getInstance()`; (ii) when C has a factory, it obtains an instance from the factory, for example, `C c = Factory.getC()`; and (iii) otherwise the function creates a null-initialized stub object to make the call, for example, `C c = new C(...)`.

Function `gen_factory(C, exp)`: generates a factory for class `C`, accepting `exp` as input. Basically, it creates a class using the template illustrated on the left of Figure B3, where `<T>` is the target type and `<exp_t>` is the list of expression types. For instance, the subfigure on the right illustrates the generated factory class for `gen_factory(Bar, {['a'], [5]})`.

Factory Template

```
1  public class <T>Factory {
2      private <T> <t>;
3
4      private <T>Factory() { }
5
6      public <T> get<t>( <exp_t> ) {
7          if (<t> == null) {
8              <t> = new <T>( <exp_t> );
9          }
10         return <t>;
11     }
12 }
```

$\Longrightarrow$

Factory of class `Bar`

```
1  public class BarFactory {
2      private Bar bar;
3
4      private BarFactory() { }
5
6      public Bar getBar( char c, int i ) {
7          if (bar == null) {
8              bar = new Bar(c,i);
9          }
10         return bar;
11     }
12 }
```

Figure B3. `gen_factory(Bar, {['a'], [5]})`.

REFERENCES

1. Parnas DL. Software aging. *16th International Conference on Software Engineering (ICSE)*, 1994; 279–287.
2. Perry DE, Wolf AL. Foundations for the study of software architecture. *Software Engineering Notes* 1992; **17**(4):40–52.
3. de Silva L, Balasubramaniam D. Controlling software architecture erosion: a survey. *Journal of Systems and Software* 2012; **85**(1):132–151.
4. van Gurp J, Bosch J. Design erosion: problems and causes. *Journal of Systems and Software* 2002; **61**:105–119.
5. Knodel J, Muthig D, Naab M, Lindvall M. Static evaluation of software architectures. *10th European Conference on Software Maintenance and Reengineering (CSMR)*, 2006; 279–294.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

6. Lindvall M, Muthig D. Bridging the software architecture gap. *Computer* 2008; **41**(6):98–101.
7. Sarkar S, Ramachandran S, Kumar G, Iyengar MK, Rangarajan K, Sivagnanam S. Modularization of a large-scale business application: a case study. *IEEE Software* 2009; **26**:28–35.
8. Knodel J, Popescu D. A comparison of static architecture compliance checking approaches. *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007; 12.
9. Passos L, Terra R, Diniz R, Valente MT, Mendonça N. Static architecture-conformance checking: an illustrative overview. *IEEE Software* 2010; **27**(5):82–89.
10. Ducasse S, Pollet D. Software architecture reconstruction: a process-oriented taxonomy. *IEEE Transactions on Software Engineering* 2009; **35**(4):573–591.
11. Murphy G, Notkin D, Sullivan K. Software reflexion models: bridging the gap between source and high-level models. *3rd Symposium on Foundations of Software Engineering (FSE)*, 1995; 18–28.
12. Mens K, Kellens A, Pluquet F, Wuyts R. Co-evolving code and design with intensional views: a case study. *Computer Languages, Systems & Structures* 2006; **32**(2-3):140–156.
13. ao Brunet J, Guerreiro D, Figueiredo J. Structural conformance checking with design tests: an evaluation of usability and scalability. *27th International Conference on Software Maintenance (ICSM)*, 2011; 143–152.
14. Aldrich J, Chambers C, Notkin D. ArchJava: connecting software architecture to implementation. *22nd International Conference on Software Engineering (ICSE)*, 2002; 187–197.
15. Hou D, Hoover H. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering* 2006; **32**(6):404–423.
16. Terra R, Valente MT. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience* 2009; **32**(12):1073–1094.
17. Eichberg M, Kloppenburg S, Klose K, Mezini M. Defining and continuous checking of structural program dependencies. *30th International Conference on Software Engineering (ICSE)*, 2008; 391–400.
18. Knodel J, Muthig D, Haury U, Meier G. Architecture compliance checking - experiences from successful technology transfer to industry. *12th European Conference on Software Maintenance and Reengineering (CSMR)*, 2008; 43–52.
19. Steimann F, Thies A. From public to private to absent: refactoring Java programs under constrained accessibility. *23rd European Conference on Object-Oriented Programming (ECOOP)*, 2009; 419–443.
20. Soares G, Gheyi R, Massoni T. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering* 2013; **39**(2):147–162.
21. Terra R, Valente MT, Czarnecki K, Bigonha R. Recommending refactorings to reverse software architecture erosion. *16th European Conference on Software Maintenance and Reengineering (CSMR), early research achievements track*, 2012; 335–340.
22. Robillard M, Walker R, Zimmermann T. Recommendation systems for software engineering. *IEEE Software* 2010; **27**(4):80–86.
23. Resnick P, Varian HR. Recommender systems. *Communications of the ACM* 1997; **40**(3):56–58.
24. Burke R. Hybrid web recommender systems. In *The adaptive web*, Vol. 4321, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007; 377–408.
25. Holmes R, Walker RJ, Murphy GC. Approximate structural context matching: an approach to recommend relevant examples. *IEEE Transactions on Software Engineering* 2006; **32**(12):952–970.
26. Zimmermann T, Zeller A, Weissgerber P, Diehl S. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 2005; **31**(6):429–445.
27. Dagenais B, Robillard MP. Recommending adaptive changes for framework evolution. *30th International Conference on Software Engineering (ICSE)*, 2008; 481–490.
28. Kruchten P. The 4+1 view model of architecture. *IEEE Software* 1995; **12**(6):42–50.
29. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Boston, 1999.
30. Romesburg H. *Cluster Analysis for Researchers*. Lulu Press: North Carolina, 2004.
31. Terra R, Valente MT. Towards a dependency constraint language to manage software architectures. *2nd European Conference on Software Architecture (ECSA)*, 2008; 256–263.
32. Knodel J, Muthig D, Rost D. Constructive architecture compliance checking – an experiment on support by live feedback. *24th International Conference on Software Maintenance (ICSM)*, 2008; 287–296.
33. Unphon H, Dittrich Y. Software architecture awareness in long-term software product evolution. *Journal of Systems and Software* 2010; **83**(11):2211–2226.
34. Feilkas M, Ratiu D, Jurgens E. The loss of architectural knowledge during system evolution: an industrial case study. *17th IEEE International Conference on Program Comprehension (ICPC)*, 2009; 188–197.
35. Murphy G, Notkin D, Sullivan K. Software reflexion models. *IEEE Transactions on Software Engineering* 2001; **27**(4):364–380.
36. Koschke R, Simon D. Hierarchical reflexion models. *10th Working Conference on Reverse Engineering (WCRE)*, 2003; 36–47.
37. Ackermann C, Lindvall M, Cleaveland R. Towards behavioral reflexion models. *20th International Symposium on Software Reliability Engineering (ISSRE)*, 2009; 175–184.
38. Koschke R, Frenzel P, Breu APJ, Angstmann K. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal* 2009; **17**:331–366.
39. Frenzel P, Koschke R, Breu APJ, Angstmann K. Extending the reflexion method for consolidating software variants into product lines. *14th Working Conference on Reverse Engineering (WCRE)*, 2007; 160 –169.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

40. Sullivan KJ, Griswold WG, Cai Y, Hallen B. The structure and value of modularity in software design. *9th International Symposium on Foundations of Software Engineering (FSE)*, 2001; 99–108.
41. Sangal N, Jordan E, Sinha V, Jackson D. Using dependency models to manage complex software architecture. *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005; 167–176.
42. Tsantalis N, Chatzigeorgiou A. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 2009; **99**:347–367.
43. Tsantalis N, Chatzigeorgiou A. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 2011; **84**(10):1757–1782.
44. O'Keeffe MK, Cinnéide M. Search-based software maintenance. *10th European Conference on Software Maintenance and Reengineering (CSMR)*, 2006; 249–260.
45. Borchers J. Invited talk: reengineering from a practitioner's view – a personal lesson's learned assessment. *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011; 1–2.
46. Hochstein L, Lindvall M. Combating architectural degeneration: a survey. *Information and Software Technology* 2005; **47**(10):643–656.
47. Rama GM, Patel N. Software modularization operators. *26th International Conference on Software Maintenance (ICSM)*, 2010; 1–10.
48. Anquetil N, Lethbridge T. Experiments with clustering as a software remodularization method. *6th Working Conference on Reverse Engineering (WCRE)*, 1999; 235–255.
49. Mitchell BS, Mancoridis S. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering* 2006; **32**(3):193–208.
50. Anquetil N, Laval J. Legacy software restructuring: analyzing a concrete case. *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011; 279–286.
51. Ye Y, Fischer G. Reuse-conducive development environments. *Automated Software Engineering* 2005; **12**:199–235.
52. Thummalapenta S, Xie T. PARSEWeb: a programmer assistant for reusing open source code on the web. *22nd International Conference on Automated Software Engineering (ASE)*, 2007; 204–213.
53. Nagappan N, Ball T, Zeller A. Mining metrics to predict component failures. *28th International Conference on Software Engineering (ICSE)*, 2006; 452–461.
54. Muşlu K, Brun Y, Holmes R, Ernst MD, Notkin D. Speculative analysis of integrated development environment recommendations. *27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012; 1–15.
55. Fowler M. *Patterns of Enterprise Application Architecture*. Addison-Wesley: Boston, 2002.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/spe