A Recommendation System for Repairing Violations Detected by Static Architecture Conformance Checking

Ricardo Terra\(^1\), Marco Tulio Valente\(^1\), Krzysztof Czarnecki\(^2\), Roberto S. Bigonha\(^1\)

\(^1\)Universidade Federal de Minas Gerais, Brazil
\(^2\)University of Waterloo, Canada

Email: \{terra,mtov,bigonha\}@dcc.ufmg.br, kczarnec@gsd.uwaterloo.ca

Abstract

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 telecommunications 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.

1 Introduction

Software architecture erosion is one of the most evident manifestations of software aging \[9,34,36\]. 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 \[20,26,53\]. 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 \[9,12\].

To tackle the erosion process, the first task is to check whether the concrete architecture conforms to the intended one \[9,10,22,35\]. More specifically, the goal of an architecture conformance
process is to reveal the implementation decisions that denote architectural violations, i.e., 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 \cite{30}, intensional views \cite{27}, design tests \cite{5}, architecture description languages \cite{2}, and domain-specific languages \cite{11,18,47}.

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 a non-trivial 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 5,000 architectural divergences in three products of this product line \cite{19}. In previous work \cite{47}, 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 2,200 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 2,100 person-days just for coding and testing \cite{42}.

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 IDEs or simple program analysis tools, such as those that extract function-call information \cite{42}. 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, i.e., refactorings whose scope are limited to a few classes, has been proved to be a complex task \cite{43,44}.

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.

![Figure 1: Proposed architectural repair recommendation approach](image-url)
In a recent short paper, we discussed the preliminary design of our recommendation approach [48]. 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 ten 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.

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 [7, 38, 39]. 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 [17]), software artifacts that should be changed together (e.g., eRose [55]), and replacement methods for adapting code to a new library version (e.g., SemDiff [8]). 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:

\[
\text{Arch. Model} \times \text{Arch. Violation} \times \text{Arch. Defective Code} \rightarrow \text{Refactoring Recommendation}
\]

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 architec-
Architectural violation \((\text{Arch. Defective Code})\). ArchFix then returns a \textit{Refactoring Recommendation} that might be useful to repair such violation. In the remainder of this section, we discuss such elements in more detail.

\textbf{Architectural Model:} Kruchten defines software architecture using five concurrent \textit{views}, each one addressing a specific set of concerns of interest to different stakeholders \cite{25}. Particularly, our work is centered on the \textit{development view} (a.k.a. \textit{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 \cite{47}. 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 \textit{dep} field): calling methods or attributes (\textit{access}), declaring variables (\textit{declare}), creating objects (\textit{create}), extending classes or implementing interfaces (\textit{derive}), throwing exceptions (\textit{throw}), or using annotations (\textit{useannotation}). For example, the relations \((A, \text{create}, B)\) and \((A, \text{access}, B)\) indicate that class \(A\) creates and calls methods of an object of type \(B\), respectively.

The \textit{architectural model} considered by ArchFix is expressed in terms of architectural constraints, which are formalized as follows:

\[
\text{Module}_1 \ [\text{cannot|must}]-dep \ \text{Module}_2
\]

where \textit{dep} denotes the dependency type, i.e., \textit{dep} can be \textit{access}, \textit{declare}, \textit{create}, etc. As an illustrative example, assume that \(M_A\) and \(M_B\) are modules, i.e., sets of classes. A constraint in the form \(M_A \text{ cannot}-dep M_B\) indicates that types from module \(M_A\) \textit{cannot} establish a dependency of the \textit{dep} kind with types from module \(M_B\), e.g., \(\text{ViewLayer cannot}-access \text{ModelLayer}\). On the other hand, a constraint in the form \(M_A \text{ must}-dep M_B\) indicates that types from module \(M_A\) \textit{must} establish a dependency of the \textit{dep} kind with types from module \(M_B\), e.g., \(\text{DTO must}-implement \text{Serializable}\).

\textbf{Architectural Violation:} Basically, there are two types of violations in the static architecture of software systems: \textit{divergences} (when an existing dependency in the source code violates the architectural model) and \textit{absences} (when the source code does not establish a dependency that is prescribed by the architectural model) \cite{22,30,35}. In our approach, \textit{divergences} occur when architectural constraints of type \textit{cannot} are not respected by the source code. Conversely, absences occur when architectural constraints of type \textit{must} are not respected.

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

**Formal Definition:** Assume that \(M_A\) and \(M_B\) are modules, A and B are classes, and that \(\text{dep}\) denotes a dependency type, i.e., \(\text{dep}\) can be \text{access}, \text{declare}, \text{create}, \text{etc}. A violation of a constraint in the form \(M_A \text{ cannot-}\text{dep} M_B\) happens whenever

\[
\exists A \exists B [ A \in M_A \land B \in M_B \land \text{establishes}(A, \text{dep}, B)]
\]

where the predicate \(\text{establishes}\) checks whether there is a dependency of type \(\text{dep}\) from \(A\) to \(B\). Conversely, a violation of a constraint in the form \(M_A \text{ must-}\text{dep} M_B\) happens whenever

\[
\exists A \nexists B [ A \in M_A \land B \in M_B \land \text{establishes}(A, \text{dep}, B)]
\]

**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 \textit{architecturally defective code}) follows one of the code templates defined in Table 1. As an example, the template \texttt{new B(exp)} covers instantiations of a given class \(B\). As another example, consider the architectural constraint \(\text{UI cannot-}\text{access Bar}\) and the class \(\text{Screen}\) presented in Code 1.

```java
public class Screen {
    private Bar bar;
    ...
    public void init() {
        ...
        bar.foo(13, 'b');
    }
}
```

Code 1: Code template matching example

Assuming that \(\text{Screen}\) is located in module \(\text{UI}\), a violation \([\text{Screen, cannot, access, Bar}]\) is raised by the call \(\text{bar.foo}(13, 'b')\) located at line 6. This call matches the template \(\texttt{b.f(exp)}\) with \(b\) bound to the target object \(\text{bar}\), \(f\) bound to the method \(\text{foo}\), and \(\text{exp}\) 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 \textit{refactoring recommendation} is a sequence of refactoring operations, using the functions described in Table 2. Although most refactoring functions are familiar, Appendix A provides a detailed description of the following particular functions: promote.param, inline, and unwrap.return.
Table 1: Code templates

<table>
<thead>
<tr>
<th>Template</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>class A</td>
<td>Class implementation</td>
</tr>
<tr>
<td>class A derive B</td>
<td>Class that extends or implements type B</td>
</tr>
<tr>
<td>@B class A</td>
<td>Class with an annotation of type B</td>
</tr>
<tr>
<td>g (p){ S }</td>
<td>Implementation of a method g, with formal parameters p and body S</td>
</tr>
<tr>
<td>@B g (p){ S }</td>
<td>Implementation of a method g, with annotation @B, formal parameters p, and body S</td>
</tr>
<tr>
<td>g (B b){ S }</td>
<td>Implementation of a method g, with a formal parameter of type B, and body S</td>
</tr>
<tr>
<td>g (p){ T v = exp,b }</td>
<td>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)</td>
</tr>
<tr>
<td>g (p){ return new B(exp) }</td>
<td>Implementation of a method g, with formal parameters p, returning an object of type B created using parameters exp</td>
</tr>
<tr>
<td>g (p) throws B { S }</td>
<td>Implementation of a method g, with formal parameters p, and body S that can throw an exception of type B</td>
</tr>
<tr>
<td>B b; S</td>
<td>Declaration of a variable b of type B followed by statements S</td>
</tr>
<tr>
<td>B b = exp; S</td>
<td>Declaration of a variable b of type B, initialized with exp and followed by statements S</td>
</tr>
<tr>
<td>catch (B b) { S }</td>
<td>Implementation of a catch clause for exceptions of type B and with a body S</td>
</tr>
<tr>
<td>b.f(exp)</td>
<td>Invocation of a method f using the target object b and actual parameters exp</td>
</tr>
<tr>
<td>new B(exp)</td>
<td>Instantiation of an object of type B using parameters exp</td>
</tr>
</tbody>
</table>

Table 2: Refactoring functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract(stm)</td>
<td>Applies an Extract Method refactoring with the set of statements stm</td>
<td>lt</td>
</tr>
<tr>
<td>inline(exp,v,S)</td>
<td>Inlines exp in the uses of variable v in the block of code S (see Appendix A)</td>
<td>lt</td>
</tr>
<tr>
<td>move(f,M)</td>
<td>Moves method f to the most suitable class in module M, i.e., applies a Move Method refactoring</td>
<td>mv</td>
</tr>
<tr>
<td>move(C,M)</td>
<td>Moves class C to module M, i.e., applies a Move Class refactoring</td>
<td>mv</td>
</tr>
<tr>
<td>promote_param(f,v,exp)</td>
<td>Promotes variable v to a formal parameter of method f; exp is used as the additional argument in the calls to f (see Appendix A)</td>
<td>lt</td>
</tr>
<tr>
<td>replace(S1,S2)</td>
<td>Replaces block of code S1 with S2</td>
<td>lt</td>
</tr>
<tr>
<td>remove(S)</td>
<td>Removes block of code S, which is equivalent to replace(S,φ)</td>
<td>lt</td>
</tr>
<tr>
<td>remove_catch(Ex,S)</td>
<td>Removes the catch clause for exception Ex from the try-catch block S</td>
<td>lt</td>
</tr>
<tr>
<td>unwrap_return(f,T,exp)</td>
<td>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 (see Appendix A)</td>
<td>lt</td>
</tr>
</tbody>
</table>

As can be observed in Table 2, 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 Subsection 3.5. As an example,
suppose that a class \texttt{ProductReport} is implemented in module \texttt{View}, while the architectural model prescribes that all reports must be implemented in module \texttt{Report}. In this case, function \texttt{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 \texttt{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 2,200 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) [47]. 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, i.e., we detected 2,241 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:

\[
[A, \text{viol\_type}, \text{dep}, B] \quad \text{code\_template} \quad \Rightarrow \quad \text{recommendation, if preconditions}
\]

This recommendation syntax is interpreted as follows: whenever the indicated violation \([A, \text{viol\_type}, \text{dep}, B]\) is found in a piece of code that matches the \text{code\_template} and the \text{preconditions} hold, the \text{recommendation} is triggered (i.e., suggested to the user).

A \text{recommendation} consists of one or more \text{refactoring} functions, as defined in Table 2. Table 3 lists a set of auxiliary functions used to define the \text{preconditions}. Some of such functions define
a simple source code query language, including functions such as \texttt{call\_sites}(f), which returns the call sites of a given method \( f \), and \texttt{super}(T), which returns the supertypes of a given type \( T \). Other functions are slightly more complex, such as \texttt{can}(T_1, \texttt{dep}, T_2), which checks whether the establishment of a dependency \((T_1, \texttt{dep}, T_2)\) raises an architectural violation, and \texttt{typecheck}(stm), which checks whether a piece of code compiles without type errors. In addition, Appendix B provides a detailed description of the following non-trivial functions: \texttt{delegate}, \texttt{factory}, \texttt{gen\_decl}, and \texttt{gen\_factory}.

### Table 3: Auxiliary functions

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

### 3.3 Recommendations

Table 4 specifies the recommendations using the aforementioned syntax and functions. The table formalizes recommendations to address both divergences (recs. \( D_1 \) to \( D_{24} \)) and absences (recs. \( A_1 \) to \( A_8 \)). 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 \( M_A^{-1} \) is the complement of the module returned by \( M_A \), i.e., 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 \([A, \text{cannot}, \text{throw}, B]\). In this particular case, a recommendation for removing the \texttt{throws} clause (rec. \( D_{15} \)) has a higher priority than the recommendation to handle the exception internally (rec. \( D_{16} \)). In this case, the order was defined considering the complexity of the recommended refactorings. More specifically, it is simpler to remove a \texttt{throws} clause than to include a catch statement and the respective exception handling code.
<table>
<thead>
<tr>
<th>Table 4: Refactoring Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, cannot.declare]</td>
</tr>
<tr>
<td>B b; S</td>
</tr>
<tr>
<td>B b; S</td>
</tr>
<tr>
<td>B b = exp; S</td>
</tr>
<tr>
<td>g(B b) {S}</td>
</tr>
<tr>
<td>catch(B b) {S}</td>
</tr>
<tr>
<td>[A, cannot.access, B]</td>
</tr>
<tr>
<td>b.f</td>
</tr>
<tr>
<td>b.f</td>
</tr>
<tr>
<td>b.f</td>
</tr>
<tr>
<td>b.f</td>
</tr>
<tr>
<td>g(p){T v = exp_b}</td>
</tr>
<tr>
<td>[A, cannot.create, B]</td>
</tr>
<tr>
<td>new B(exp)</td>
</tr>
<tr>
<td>new B(exp)</td>
</tr>
<tr>
<td>new B(exp)</td>
</tr>
<tr>
<td>g(p){return new B([exp])}</td>
</tr>
<tr>
<td>[A, cannot.throw, B]</td>
</tr>
<tr>
<td>g(p) throws B {S}</td>
</tr>
<tr>
<td>g(p) throws B {S}</td>
</tr>
<tr>
<td>g(p) throws B {S}</td>
</tr>
<tr>
<td>g(p) throws B {S}</td>
</tr>
<tr>
<td>[A, cannot.derive, B]</td>
</tr>
<tr>
<td>class A derive B</td>
</tr>
<tr>
<td>class A derive B</td>
</tr>
<tr>
<td>[A, cannot.useannotation, B]</td>
</tr>
<tr>
<td>B class A</td>
</tr>
<tr>
<td>B class A</td>
</tr>
<tr>
<td>B g(p){S}</td>
</tr>
<tr>
<td>B g(p){S}</td>
</tr>
<tr>
<td>[A, must.throw, B]</td>
</tr>
<tr>
<td>g(p){S}</td>
</tr>
<tr>
<td>g(p){S}</td>
</tr>
<tr>
<td>[A, must.derive, B]</td>
</tr>
<tr>
<td>class A</td>
</tr>
<tr>
<td>class A</td>
</tr>
<tr>
<td>[A, must.useannotation, B]</td>
</tr>
<tr>
<td>class A</td>
</tr>
<tr>
<td>class A</td>
</tr>
<tr>
<td>g(p){S}</td>
</tr>
<tr>
<td>g(p){S}</td>
</tr>
</tbody>
</table>
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 GWT (Google Web Toolkit) should avoid the use of generic types (e.g., `java.util.Collection`) on GWT 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.

**D11**: Replace a `new` operator with a call to the `get` method of a Factory \( F \). It addresses the situation where developers—due to unawareness or forgetfulness—create directly objects of classes that have a well-defined factory.

**D16**: 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`.

**D20**: Move a class \( A \) to a most suitable module \( M \). It is particularly useful when developers mistakenly implement a class in the wrong module, e.g., a `ProductReport` class in the `View` layer.

**A3**: 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 web site.

\[\text{http://www.dcc.ufmg.br/~terra/spe2013}\]
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 \([A, \text{must, useAnnotation, B}]\), 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**

<p>| Input: Architectural model (arch), violation (viol), and defective code (code) |</p>
<table>
<thead>
<tr>
<th>Output: Refactoring recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: <code>potentialRecs ← getRecommendations(viol)</code></td>
</tr>
<tr>
<td>2: <strong>for each</strong> <code>rec</code> <strong>in</strong> <code>potentialRecs</code> <strong>do</strong></td>
</tr>
<tr>
<td>3:   <strong>if</strong> <code>(code matches rec.code_template and ⟨code, viol, arch⟩ satisfies rec.preconditions)</code> <strong>then</strong></td>
</tr>
<tr>
<td>4:     <strong>return</strong> <code>rec.recommendation</code></td>
</tr>
<tr>
<td>5:   <strong>end if</strong></td>
</tr>
<tr>
<td>6: <strong>end for</strong></td>
</tr>
<tr>
<td>7: <strong>return</strong> ϕ</td>
</tr>
</tbody>
</table>

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 4, 14 out of 32 recommendations (e.g., D18, D20, D21, A4, 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 code entities to make a recommendation. In order to measure this similarity, we use the Jaccard Similarity Coefficient [40], 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:

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

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 $dep = \ast$, we do not consider a particular dependency type when calculating $Deps$, i.e., 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:

$$\max \{ \max_{\forall m} \text{sim}(M_1, m, dep), \max_{\forall m} \text{sim}(M_1, m, \ast) \}$$

In other words, the most suitable module is the module with the highest Jaccard coefficient—as returned by the $\text{sim}$ function—considering two possible sets of dependencies: (1) only dependencies of type $dep$; (2) all dependencies, independently of their type (i.e., $dep = \ast$). 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 $dep$, i.e., $Deps(M_1, dep) = \emptyset$.

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 DCL language [46,47], which is a domain-specific language for defining structural constraints between modules in Java. Compared to other approaches, DCL relies on a simple syntax and supports architecture compliance by construction, in order to proactively prevent architecture decay [21]. 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, e.g., a function that checks whether a type can establish a particular dependency with another type (function $\text{can}$, Table 3).

ArchFix follows an architecture with three main modules:

- Recommendation Engine: Based on the specification shown in Table 4, 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 Section 3.4.
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 marker\textsuperscript{2} that represent architectural violations.

As illustrated in Figure\textsuperscript{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\textsuperscript{3}.

- **Refactoring Functions**: This module is responsible for applying the refactorings listed in Table\textsuperscript{2}. In the current stage of its implementation, ArchFix is mostly a recommendation engine, i.e., 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, i.e., a class that introduces a violation in

\textsuperscript{2}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 IDE.
the form \([\text{ProductController}, \text{cannot}, \text{declare}, \text{HibernateDAO}]\). When the maintainer requests a recommendation to fix such violation, ArchFix indicates the most appropriate refactoring (see Figure 3). The provided recommendation suggests replacing the declaration of the unauthorized type \(\text{ProductHibernateDAO}\) with its interface \(\text{IProductDAO}\) (which corresponds to recommendation \(D1\) in Table 4). This recommendation is particularly useful to handle violations due to references to a concrete implementation of a service, instead to its general interface.

![Figure 3: ArchFix interface](image)

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 the Methodology section (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\(^3\). The system handles strategic management activities, including management plans, goals, performance indicators, actions, etc. The second one is a large and complex customer care platform of a major Brazilian telecommunication company. Due to a non-disclosure 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, etc. Table 5 shows information on the size of both systems.

<table>
<thead>
<tr>
<th></th>
<th>Geplanes</th>
<th>BrTCom</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>21,799</td>
<td>728,814</td>
</tr>
<tr>
<td>Subsystems</td>
<td>1</td>
<td>146</td>
</tr>
<tr>
<td>Packages</td>
<td>25</td>
<td>2,289</td>
</tr>
<tr>
<td>Classes</td>
<td>278</td>
<td>4,724</td>
</tr>
<tr>
<td>Interfaces</td>
<td>1</td>
<td>1,893</td>
</tr>
<tr>
<td>External libraries</td>
<td>47</td>
<td>58</td>
</tr>
</tbody>
</table>

5.3 Methodology

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

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. Since the constraints were provided in natural language, we translated the informal definitions to DCL and validated them with the architects in a 30-minutes individual meeting.

![Figure 4: Methodology followed in the evaluation](image)

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. Al-

\(^3\)http://www.softwarepublico.gov.br
though Geplanes’ meeting lasted 25 minutes, BrTCom’s meeting lasted almost one hour 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. Since Geplanes triggered few violations (only 41 in total), this classification was possible during a 30-minutes 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. Since the violations were very similar, we randomly selected six to be scored by the architect. In this way, in two 30-minutes meetings with the architect, it was possible to classify 89 recommendations that represent the whole set of recommendations triggered for the BrTCom system.

5.3.3 Complexity Evaluation

We conducted a 30-minutes 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-minutes 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 6 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. The reflexion model is mainly related to the MVC-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 6, we have found 41 architectural violations and ArchFix was able to provide recommendations for all of them.

Correctness (RQ #2): Table 6 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%).

4We 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.
As reported in Table 6, 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. A6 and A8). For example, as can be observed in Code 2, class AnomaliaCrud is missing annotation @Controller required by the underlying MVC-based framework. Therefore, this absence represents a violation of constraint GP6, as illustrated by an arrow with an ‘!’ 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.

```java
package br.com.linkcom.sgm.controller.crud;

public class AnomaliaCrud extends SGMCrudController<AnomaliaFiltro, Anomalia, Anomalia> {
    private FerramentaAnomaliaService fas = new FerramentaAnomaliaService();
    ...
}
```

Code 2: Controller class AnomaliaCrud

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—$\text{sim}(\text{Verbo, Entities, *}) = 0.4216$—, whereas the current module of the class (Controllers) has a significant lower similarity—$\text{sim}(\text{Verbo, Controllers, *}) = 0.0763$.

```java
package br.com.linkcom.sgm.controller;

@Entity
@SequenceGenerator (name = "sq_verbo", sequenceName = "sq_verbo")
public class Verbo {
    private Integer id;
    private String nome;
    ...
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO, generator="sq_verbo")
    public Integer getId () {
        return id;
    }
}
```

Code 3: “Controller” class Verbo

**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
viations 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.

5.5 BrTCom Results

The results for BrTCom are discussed next.

Recommendations (RQ #1): Table 7 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), etc. ArchFix raised 727 recommendations for the 787 violations we have detected using the constraints in Table 7 (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.

Table 7: Recommendations and correctness evaluation (BrTCom)

<table>
<thead>
<tr>
<th>Violations</th>
<th>Number</th>
<th>Recommendations</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DTOs, must, implement, java.io.Serializable]</td>
<td>63</td>
<td>A3(50−0−0); A4(0−0−13)</td>
<td>50−0−13</td>
</tr>
<tr>
<td>[SAOs, must, extend, brtcom.server.sao.AbstractSAO]</td>
<td>1</td>
<td>A4(0−1−0)</td>
<td>0−1−0</td>
</tr>
<tr>
<td>[Controllers, must, useannotation, brtcom.client.controller.Controller]</td>
<td>1</td>
<td>A6(1−0−0)</td>
<td>1−0−0</td>
</tr>
<tr>
<td>[DataSources, must, useannotation, brtcom.client.datasource.DataSource]</td>
<td>1</td>
<td>A6(1−0−0)</td>
<td>1−0−0</td>
</tr>
<tr>
<td>[brtcom server dao, brtcom.dao.jpaDAO, cannot, create, DAOs]</td>
<td>13</td>
<td>D11(13−0−0)</td>
<td>13−0−0</td>
</tr>
<tr>
<td>[DAOs, cannot, throw, brtcom.server.dao.DAOException]</td>
<td>15</td>
<td>D15(11−0−0); D16(2−0−0)</td>
<td>13−0−0</td>
</tr>
<tr>
<td>[CtrlLayer, DLSLayer], cannot, useannotation, CtrlDSAnnotations]</td>
<td>20</td>
<td>D21(2−0−0); D22(18−0−0)</td>
<td>20−0−0</td>
</tr>
<tr>
<td>[Services, SAOLayer], cannot, depend, SAOs</td>
<td>5</td>
<td>D20(1−0−0)</td>
<td>1−0−0</td>
</tr>
<tr>
<td>[System, cannot, create, [Controllers, DataSources]]</td>
<td>3</td>
<td>D12(3−0−0)</td>
<td>3−0−0</td>
</tr>
<tr>
<td>[CtrlLayer, cannot, create, java.util.Date]</td>
<td>84</td>
<td>D13(0−84−0)</td>
<td>0−84−0</td>
</tr>
<tr>
<td>[ScreenWrappers, cannot, useannotation, JavaLangAnnotations]</td>
<td>18</td>
<td>D21(0−0−0); D22(13−0−0)</td>
<td>13−0−5</td>
</tr>
<tr>
<td>[System, cannot, depend, java.lang.System]</td>
<td>14</td>
<td>D9(14−0−0)</td>
<td>14−0−0</td>
</tr>
<tr>
<td>[ServicesAsync, cannot, declare, UnallowedAbstractTypes]</td>
<td>270</td>
<td>D2(270−0−0)</td>
<td>270−0−0</td>
</tr>
<tr>
<td>[ServerLayer, cannot, depend, ClientUtil]</td>
<td>279</td>
<td>D7(225−0−0)</td>
<td>225−0−0</td>
</tr>
</tbody>
</table>

787 violations, 624−85−18
Correctness (RQ #2): As can be observed in Table, 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, i.e., 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, 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, since 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.

\footnote{A similar recommendation (A5) was triggered in Geplanes’ case study and it was considered as having moderate complexity. Basically, the recommendations involves moving a DTO, 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.}
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 8 classifies the detected violations according to a set of architectural defect types initially proposed by Knodel and Popescu [22] and later extended by us [47]. As a general fact, the architects ascribed most of the detected violations to the lack of awareness about the architectural model [52] and copy-and-paste procedures [12].

<table>
<thead>
<tr>
<th>Architectural Defect Type</th>
<th>Constraints</th>
<th># viols.</th>
<th># recs.</th>
<th># correct recs.</th>
<th>precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misusage of persistence patterns</td>
<td>GP1–3; TC1</td>
<td>68</td>
<td>68</td>
<td>54</td>
<td>0.79</td>
</tr>
<tr>
<td>Misusage of domain-specific patterns</td>
<td>GP4–7; TC2–4</td>
<td>25</td>
<td>25</td>
<td>23</td>
<td>0.92</td>
</tr>
<tr>
<td>Bypassing layers</td>
<td>GP8–9; TC6–8; TC14</td>
<td>321</td>
<td>261</td>
<td>261</td>
<td>1.00</td>
</tr>
<tr>
<td>Unintended dependencies</td>
<td>GP10; TC9; TC11–12</td>
<td>40</td>
<td>40</td>
<td>32</td>
<td>0.80</td>
</tr>
<tr>
<td>Bypassing creational patterns</td>
<td>GP11; TC5</td>
<td>20</td>
<td>20</td>
<td>15</td>
<td>0.75</td>
</tr>
<tr>
<td>Context exploration</td>
<td>TC10</td>
<td>84</td>
<td>84</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>Misusage of interfaces</td>
<td>TC13</td>
<td>270</td>
<td>270</td>
<td>270</td>
<td>1.00</td>
</tr>
</tbody>
</table>

828 768 655

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 8, 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 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 (page 5), 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 OO systems.

RQ #2 – Does the proposed approach provide correct recommendations for repairing architectural violations?
As can be observed in Table 8, 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 D2—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 9, 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 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.

Table 9: Classification of the detected violations (focusing on complexity)

<table>
<thead>
<tr>
<th>Architectural Defect Type</th>
<th>Complexity</th>
<th>minor</th>
<th>correct recs.</th>
<th>incorrect recs.</th>
<th>minor</th>
<th>moderate</th>
<th>major</th>
<th>minor</th>
<th>moderate</th>
<th>major</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misusage of persistence patterns</td>
<td>minor</td>
<td>52</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>14</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Misusage of domain-specific patterns</td>
<td>minor</td>
<td>23</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bypassing layers</td>
<td>minor</td>
<td>18</td>
<td>2</td>
<td>241</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Unintended dependencies</td>
<td>minor</td>
<td>27</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bypassing creational patterns</td>
<td>minor</td>
<td>-</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Context exploration</td>
<td>minor</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Misusage of interfaces</td>
<td>minor</td>
<td>-</td>
<td>270</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>120</td>
<td>289</td>
<td>246</td>
<td>1</td>
<td>14</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

As also reported in Table 9, 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 al. demonstrate that teams supported by constructive compliance checking may insert about 60% less structural violations into the architecture [21]. 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.

Forth, only 17 out of the 32 proposed recommendations, which emerged from our initial study [47], 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, which were commonly detected in other case studies [22, 42, 47]. 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: Since 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 minutes.

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, since 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, since 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, since 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 [47].

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, since our approach relies on primitive refactorings to repair architectural violations; (iv) remodularization approaches, because they are potential alternatives to architecture degradation; and (v) recommendation systems, since 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) [25]. 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 [30]. Similarly, there are studies that ascribe the violations tackled by our approach as due to design erosion rather than architectural erosion [5,53].

Architecture Conformance: Over the past decade, several techniques have been proposed to deal with the architecture erosion problem [35]. For instance, Silva and Balasubramaniam indicated a combination of strategies that might help to control architecture erosion [9]. 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 (RM) are based on the comparison of two models—representing a high-level and a low-level view of the target system [30][31]. 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 reflexion model principles [20][21][26]. Furthermore, there are also several works extending the original reflexion model to support, for example, hierarchical structures [24], behavioral design [1], and software variants [15][23].

Dependency Structure Matrices (DSMs) can be used to provide a scalable view of the established dependencies among classes of a system [41][45]. A DSM is a weighted square matrix whose both rows and columns denote classes from an object-oriented system. The 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 [46][47]. 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 semi-automatic approach to identify Move Method refactoring opportunities [50]. Their general goal is to tackle coupling and cohesion anomalies manifested in the form of the Feature Envy bad smell [13]. 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 [51].

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 [33]. 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.

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 [6]. 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 [16].

Rama and Patel analyzed several remodularization efforts in order to define recurring modular-

---

6http://www.lattix.com
ization operators [37]. 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 [4, 28]. 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 [3]. 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 towards a desired design, provided in the format of a UML class diagram [33]. 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 [39]. For example, current RSSEs can recommend relevant source code fragments to help developers to use frameworks and APIs (CodeBroker [54], PARSEWeb [49], and Strathcona [17]), software artifacts that must be changed together (eRose [55]), parts of the software that should be tested more cautiously (Suade [32]), and replacement methods for adapting code to a new library version (SemDiff [8]). 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 [29]. 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 \cite{2,11,27,30,47}. 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 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, \textit{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 \textit{moderate} or \textit{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 \textit{ArchFix} on other systems; (iii) designing an architecture-repair recommendation language, in order to allow maintainers to extend \textit{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 \textit{ArchFix}; (v) refining the current prioritization heuristic through elements of a learning system or by means of speculative analysis \cite{29}.

The \textit{ArchFix} tool—including its complete source code—is publicly available at 
\url{http://github.com/rterrabh/DCL}

Acknowledgments

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

Appendix

In this appendix, we provide detailed descriptions and illustrative examples on the non-trivial functions used by \textit{ArchFix} to recommend architectural refactorings. Appendix A addresses non-trivial \textit{refactoring} functions as introduced in Table 2 whereas Appendix B addresses non-trivial \textit{auxiliary} functions as introduced in Table 3.
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 6 (before). Therefore, a violation `[Facade, cannot, access, Foo]` occurs at line 9 in this figure.

```java
before
public class MainClass {
    public void main() {
        facade.setup(foo);
    }
    public class Facade {
        public void setup(Foo foo) {
            Date d = foo.getDate();
            ...
        }
    }
}

after
public class MainClass {
    public void main() {
        facade.setup(foo, foo.getDate());
    }
    public class Facade {
        public void setup(Foo foo, Date d) {
            ...
        }
    }
}
```

Figure 6: `promote(param(setup(Foo), d, [foo.getDate()]))`

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 be observed in Figure 6 (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 7 (before). Therefore, a violation `[Bar, cannot, declare, Foo]` occurs at line 3 in this figure.

```java
before
public class Bar {
    public void init(Context ctx) {
        Foo foo = ctx.getFoo();
        int x = foo.getX();
        int y = foo.getY();
        ...
    }
}

after
public class Bar {
    public void init(Context ctx) {
        int x = ctx.getFoo().getX();
        int y = ctx.getFoo().getY();
        ...
    }
}
```

Figure 7: `inline([ctx.getFoo()], foo, S)`

Since 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 7 (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 8 (before). Therefore, a violation `[Model, cannot, create, Foo]` occurs at line 10 in this figure.

![Figure 8: unwrap_return(retrieve(), Foo, [obj])](image)

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 8 (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).

### B Auxiliary Functions

Function `delegate(f)`: Searches a delegate method for `f`, i.e., a method that just encapsulates a call to `f` [14]. 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 9, `Persistence::persist(Bar)` is a delegate method for `Bar::save(Connection)` because it only forwards the call (line 5).

![Figure 9: delegate(Bar::save(Connection)) = Persistence::persist(Bar)](image)
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 just returns an object of type C created using `exp`. For instance, as presented in Figure 10, `DAOFactory::getBar(int)` is a factory method for Bar (lines 3–5).

```java
public class DAOFactory {
    ...  
    public int getBar(int max) {
        return new Bar(max);
    }
}
```

Figure 10: `factory(Bar, {5}) = DAOFactory::getBar(int)`

Function `gen_decl(C, f)`: Declares a variable of class C to access 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, e.g., `C c = C.getInstance();` (ii) when C has a factory, it obtains an instance from the factory, e.g., `C c = Factory.getC();` (iii) otherwise the function creates a null-initialized stub object to make the call, e.g., `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 11 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]})`.

```
public class Factory {
    private <T> t;
    private Factory() {} 
    public <T> get<T>( <exp_t> ) {
        if (<t> == null) {
            <t> = new <T>( <exp_t> );
        }
    return <t>;
    }
}
```

```
public class BarFactory {
    private Bar bar;
    private BarFactory() {} 
    public Bar getBar( char c, int i ) {
        if (bar == null) {
            bar = new Bar(c,i);
        }
    return bar;
    }
}
```

Figure 11: `gen_factory(Bar, {[‘a’],[5]})`

References


