A Recommendation System for Tackling Software Architecture Erosion

Ricardo Terra[†‡], Marco Tulio Valente[†], Krzysztof Czarnecki[‡] and Roberto S. Bigonha[†]
[‡]*University of Waterloo, Canada*
[†]*Universidade Federal de Minas Gerais, Brazil*
Email: {terra,mtov}@dcc.ufmg.br, kczarnec@gsd.uwaterloo.ca, bigonha@dcc.ufmg.br

***Abstract*—In this paper, we describe a recommendation system that provides refactoring guidelines for maintainers when tackling an architectural erosion process. The paper formally describes 32 refactoring recommendations to fix architectural violations, describes a tool—called DCLfix—that triggers the proposed recommendations, and evaluates the application of this tool in two industrial-strength systems. For the first system, DCLfix has been able to recommend 31 refactorings that constitute the appropriate solution for 41 violations discovered as the result of an architecture conformance process. For the second system—a 728 KLOC customer care system used by a major telecommunication company—DCLfix has triggered 624 useful recommendations, as asserted by the system's architect.**

I. Introduction

Software architecture erosion is one of the most evident manifestations of software aging [1], [2]. Basically, 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 [3]–[5]. Among the causes of architectural erosion, we can mention deadline pressures, conflicting requirements, communication mismatches, unawareness of developers, and the lack of an explicit correspondence between architectural and programming language abstractions. Regardless the causes, when the erosion is neglected over the years, it can reduce the architecture to a small set of strongly-coupled and weakly-cohesive components, whose maintenance and evolution become increasingly more difficult and costly [6], [7].

To tackle the erosion process, the first task is to check whether the current architecture conforms to the intended one [7]–[10]. More specifically, the goal of an architecture conformance process is to reveal in the source code the implementation decisions that denote architectural violations, i.e., concrete statements, expressions, or declarations that do not match the constraints imposed by the intended architecture. For this purpose, several architecture conformance techniques and approaches have been proposed, including reflexion models [11], intensional views [12], design tests [13], architectural 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 a non-trivial and time-consuming task, because software erosion is mostly a silent process that accumulates over years. For example, Knodel et al. described their experience of applying an architecture conformance process to a real-world product line in the domain of portable measurement devices. As a result, they have identified almost 5,000 architectural divergences in three products of such product line [18]. In a previous work [16], we have described our own experience in applying conformance techniques to a human-resource management system. In this process, we have been able to discover more than 2,200 architectural violations. As a last example, Sarkar et al. reported their experience in remodularizing a large banking application, whose architecture has deteriorated into an unmanageable monolithic block. Reconstructing the original architecture of this system demanded 2,100 person-days just for coding and testing [6].

However, despite of its relevance and contrasting with the variety of techniques available for architecture conformance, the task of fixing architectural violations is usually performed in ad hoc ways. Usually, the only employed tools are the automatic refactorings provided by today's IDEs or simple program analysis tools, such as tools that extract function-call information [6]. In view of such circumstances, a solution based on recommendation system principles represents an interesting approach. Basically, such a solution can provide useful refactoring guidelines for developers and maintainers when fixing architectural violations. Therefore, it may represent a real improvement to the state of the practice in architecture repair. On the other hand, by definition, an approach based on recommendations does not have the ambition to provide a fully automatic solution for removing 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 [19].

In this paper, we first describe the design and implementation of `DCLfix`, a recommendation system whose main purpose is to provide refactoring guidelines for developers and maintainers when removing architectural violations. Specifically, `DCLfix` provides recommendations to remove architectural violations detected by the DCL language [16], a domain-specific language with a simple and self-explaining syntax for defining structural constraints between modules. As illustrated in Figure 1, the proposed recommendation engine has the following inputs: a set of DCL constraints (specified by a software architect or designer), a set of architectural violations (raised by the

DCLcheck conformance tool), and the source code of the system. Based on these inputs, the recommendation engine provides a set of recommendations to guide the process of removing the detected violations. For example, the system may suggest the use of a *Move Method* refactoring to fix a given violation, including a suggestion of a target class.

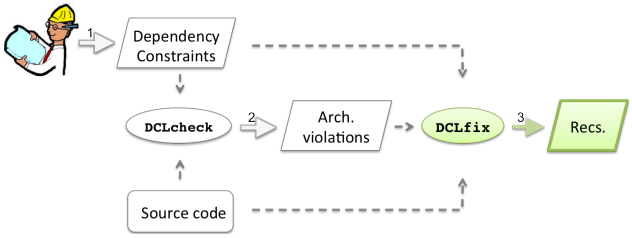

Figure 1. DCLfix recommendation engine

In a recent short paper, we discussed the preliminary design of our recommendation approach [20]. This paper extends our early 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 the recommendation engine, as an Eclipse plugin; (c) a detailed evaluation of DCLfix in two real-world systems. For the first system, DCLfix has been able to recommend 31 refactorings that constitute the appropriate solution for 41 violations discovered as the result of an architecture conformance process. For the second system—a 728 KLOC customer care system used by a major Brazilian telecommunication company—DCLfix has triggered 624 useful recommendations, as endorsed by the system's architect.

The remainder of this paper is structured as follows. Section II provides an overview on the DCL language. Section III presents a formal specification of the recommendations raised by DCLfix. Section IV describes the design and implementation of the DCLfix tool. Section V presents and discusses results on applying these recommendations in two real-world systems. Finally, Section VI presents related work and Section VII concludes the paper.

II. DCL in a Nutshell

The Dependency Constraint Language (DCL) is a statically checked domain-specific language that provides a fine-grained model to control the establishment of inter-module dependencies in object-oriented systems [16]. Particularly, the language provides constraints to capture two types of architectural violations: *divergences* (when an existing dependency in the source code violates the intended architecture) and *absences* (when the source code does not establish a dependency that is prescribed by the intended architecture). To capture divergences, DCL allows architects to specify that dependencies *only can*, *can only*, or *cannot* be established by specified modules. In addition, to capture absences, it is possible to specify that particular dependencies *must* be present in the source code.

To illustrate the use of DCL, assume the following constraints:

```
1: only Factory can-create Product
2: Util can-only-depend java
3: View cannot-access Model
4: Product must-implement Serializable
```

These constraints state that only classes in the Factory module can create objects from classes in the Product module (line 1), that classes in the Util module can only establish dependencies with classes from the Java API (line 2), that classes in the View module cannot access classes in the Model module (line 3), and that every class in the Product module must implement Serializable (line 4).

When defining constraints, DCL allows developers to specify dependencies caused by accessing methods and fields (access), declaring variables (declare), creating objects (create), extending classes (extend), implementing interfaces (implement), throwing exceptions (throw), or using annotations (useannotation). It is also possible to define constraints including any form of dependency (depend).

III. Refactoring Recommendations

The proposed refactoring recommendations aim to assist developers and maintainers to fix violations detected by DCLcheck. They have been proposed based on the investigation of possible fixes for more than 2,200 architectural violations we have detected in a previously evaluated system (our training set) [16].

In this section, the following syntax is used to define the recommendations:

dcl_constraint		
code_with_violation	$\Longrightarrow$	recommendation, if preconditions

This syntax should be interpreted as follows: whenever the dcl_constraint is violated by the particular code_with_violation and the preconditions hold, the recommendation is triggered. More specifically, dcl_constraint is a constraint defined in DCL and code_with_violation is the particular statement or expression in the source code where this constraint has been violated. A recommendation consists of a sequence of refactoring operations, using the functions described in Table I. This table also lists the auxiliary functions used to define the preconditions.

Based on the proposed syntax and functions, Table II shows a formal specification for the refactoring recommendations. The table formalizes recommendations to both divergences (recs. $D1$ to $D24$) and absences (recs. $A1$ to $A8$). Also, the recommendations are listed according to their priority, i.e., when two or more recommendations match a given violation, the system only triggers the recommendation with the highest priority. For formalization purposes, dcl_constraint in the Table II

Table I
REFACTORING (Ref) AND AUXILIARY (Aux) FUNCTIONS

Type	Function	Description
Aux	can(T_1, dep, T_2)	Checks whether type (class or interface) T_1 can establish a dependency of the dep kind with T_2
Aux	call_sites(f)	Returns the call sites of f
Aux	delegate(f)	Searches for a delegate method of f
Ref	extract(stm)	Extracts method with statements stm
Aux	factory(C, exp)	Searches for a factory for class C, accepting exp as input
Ref	gen_decl(C, f)	Declares a variable of class C to access f
Ref	gen_factory(C, exp)	Generates a factory for class C, accepting exp as input
Ref	move(f, M)	Moves the method f to the most suitable class into the module M
Ref	move(C, M)	Moves the class C to the module M
Aux	override(C, C′)	Checks whether class C overrides at least a method of its superclass C′
Ref	promote(f, v, exp)	Promotes variable v to a formal parameter of f; exp is used as the argument in the calls to f
Ref	propagate(exp, v, S)	Propagates exp to the uses of the variable v in the block of code S
Ref	remove(S)	Removes the block of code S
Ref	remove_treatment(S, EX)	Removes the catch clauses of the exception EX from the block of code S
Aux	replace(stm_1, stm_2)	Replaces block of code stm_1 by stm_2
Ref	replace_return(f, T, exp)	Modifies the return type of the method f to the exp type and transfer creations of T to the call sites
Aux	same_sign(f_1, f_2)	Checks if the method f_1 has the same signature of the method f_2
Aux	sub(T)	Returns the subtypes of type T (ascending by specialization)
Aux	suitable_module(E)	Returns the most suitable module for a source code entity E considering its context (see III-A)
Aux	super(T)	Returns the supertypes of type T (descending by specialization)
Aux	target(A)	Returns the target type of annotation A, which can be *type* or *method*
Aux	treat(S, EX)	Checks whether there is a catch clause of the exception EX inside the block of code S
Aux	type(v)	Returns the type of the variable v
Aux	typecheck(stm)	Checks whether code stm type checks
Aux	user_code()	Prompts the user for a block of code

considers the classes $\mathtt{A} \in \mathtt{M_A}$ and $\mathtt{B} \in \mathtt{M_B}$ and adopts derive as meaning both implements and extends.

The proposed recommendations also handle divergences detected by *only can* and *can only* rules, because they are defined in terms of *cannot* rules, as described next:

$$\mathtt{only\ M_A\ can\text{-}dep\ M_B} \Longrightarrow \mathtt{\overline{M_A}\ cannot\text{-}dep\ M_B}$$

$$\mathtt{M_A\ can\text{-}only\text{-}dep\ M_B} \Longrightarrow \mathtt{M_A\ cannot\text{-}dep\ \overline{M_B}}$$

where $\mathtt{M_A}$ and $\mathtt{M_B}$ are modules (i.e., sets of classes), $\mathtt{\overline{M_A}}$ denotes the complement of module $\mathtt{M_A}$ (i.e., all classes under analysis except those in $\mathtt{M_A}$) and dep denotes a dependency type (i.e., access, declare, create, etc).

Section V provides concrete examples of the use of the proposed recommendations in two real-world systems. In addition, a detailed description of each recommendation is available in a companion web site[1].

A. Suitable Module

Many recommendations (e.g., D18, D20, D21, A2, etc) include a suggestion to move methods or classes to more *suitable* modules, as computed by the function suitable_module. Essentially, the implementation of 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, which is a statistical measure for the similarity between two sets. To calculate this coefficient, we assume that a given source code entity (method, class, or module) is represented by the set of dependencies it establishes with other program elements. This assumption is based on the fact that our recommendations have been proposed to handle violations in DCL constraints, which basically are defined to capture divergences or absences in the dependencies established by a given program element.

Based on such assumptions, the similarity between the source code entities E_1 and E_2 is defined by:

$$sim(E_1, E_2, dep) = \frac{|Deps(E_1, dep) \cap Deps(E_2, dep)|}{|Deps(E_1, dep) \cup Deps(E_2, dep)|}$$

where dep denotes the dependency type (e.g., access, create, etc) that has motivated the similarity calculation. In addition, $Deps(E, dep)$ is a set whose elements are pairs $[dep, t]$, denoting the existence of a dependency of type dep between e and a given type t, where $e \in E$. When $dep = *$, we do not consider a particular dependency type when calculating $Deps$, i.e., dependencies of any type are included in the resulting set.

Suppose that a violation is detected in the module E_1 involving a dependency of type dep. Suppose also that the recommendation for this violation requires moving the class responsible by the violation to another module E_2. In this case, E_2 is defined as the module m that provides the following maximal value:

$$\max\ \{\ \max_{\forall m} sim(E_1, m, dep),\ \max_{\forall m} sim(E_1, m, *)\ \}$$

In other words, the most suitable module is the module with the highest Jaccard coefficient—as returned by the sim function—considering two posssible sets of dependencies: (1) only dependencies of type dep; (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 source module E_1 in this case misses a dependency of type dep prescribed in a *must* constraint (i.e., $Deps(E_1, dep) = \emptyset$).

[1] http://www.dcc.ufmg.br/~terra/dclfix

Table II
REFACTORING RECOMMENDATIONS

A cannot-declare B, where $\texttt{A} \in \texttt{M}_\texttt{A} \wedge \texttt{B} \in \texttt{M}_\texttt{B}$		
B b; S	$\Longrightarrow$ replace([B], [B′]), if $\texttt{B}' \in \texttt{super(B)} \wedge \texttt{typecheck}(\texttt{[B' b ; S]}) \wedge \texttt{B}' \notin \texttt{M}_\texttt{B}$	$D1$
B b; S	$\Longrightarrow$ replace([B], [B′]), if $\texttt{B}' \in \texttt{sub(B)} \wedge \texttt{typecheck}(\texttt{[B' b ; S]}) \wedge \texttt{B}' \notin \texttt{M}_\texttt{B}$	$D2$
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 $\texttt{B}' \in \texttt{super(B)} \wedge \texttt{typecheck}(\texttt{[catch(B' b)\{ S \}]}) \wedge \texttt{B}' \notin \texttt{M}_\texttt{B}$	$D5$
A cannot-access B		
b.f	$\Longrightarrow$ replace([b.f], [D; c.g]), if $\texttt{g} = \texttt{delegate(f)} \wedge \texttt{D} = \texttt{gen\_decl(type(c), g)} \wedge \texttt{type(c)} \notin \texttt{M}_\texttt{B}$	$D6$
b.f	$\Longrightarrow$ replace([b.f], [D; c.g]), if $\texttt{same\_sign(f, g)} \wedge \texttt{D} = \texttt{gen\_decl(type(c), g)} \wedge \texttt{type(c)} \notin \texttt{M}_\texttt{B}$	$D7$
b.f	$\Longrightarrow$ g = extract([b.f]), move(g, M), if $\texttt{M} = \texttt{suitable\_module(g)} \wedge \texttt{can(A, access, M)}$	$D8$
b.f	$\Longrightarrow$ remove([b.f]), if $\overline{\texttt{M}_\texttt{A}} = \emptyset$	$D9$
g { T v = exp_b }	$\Longrightarrow$ promote(g, v, [exp_b]), if $\forall \texttt{C} \in \texttt{call\_sites(g)}, \texttt{can(C, access, B)}$	$D10$
A cannot-create B		
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [FB.getB(exp)]), if $\texttt{FB} = \texttt{factory(B, [exp])} \wedge \texttt{can(A, access, FB)}$	$D11$
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [null]), if $\overline{\texttt{M}_\texttt{A}} = \emptyset$	$D12$
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [FB.getB(exp)]), if $\texttt{FB} = \texttt{gen\_factory(B, [exp])} \wedge \texttt{can(A, access, FB)}$	$D13$
g { return new B(exp) }	$\Longrightarrow$ replace_return(g, B, [exp]), if $\forall \texttt{C} \in \texttt{call\_sites(g)}, \texttt{can(C, create, B)}$	$D14$
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 $\texttt{can(A, declare, B)} \wedge \texttt{S}' = \texttt{user\_code()}$	$D16$
g (p) throws B	$\Longrightarrow$ replace([B], [B′]), if $\texttt{B}' \in \texttt{super(B)} \wedge \texttt{B}' \notin \texttt{M}_\texttt{B}$	$D17$
g (p) throws B	$\Longrightarrow$ move(g, M), if $\texttt{M} = \texttt{suitable\_module(g)} \wedge \texttt{M} \neq \texttt{M}_\texttt{A}$	$D18$
A cannot-derive B		
A derive B	$\Longrightarrow$ replace([B], [B′]), if $\texttt{B}' \in \texttt{super(B)} \wedge \texttt{typecheck}(\texttt{[A derive B']}) \wedge \neg\texttt{override(B, B')} \wedge \texttt{B}' \notin \texttt{M}_\texttt{B}$	$D19$
A derive B	$\Longrightarrow$ move(A, M), if $\texttt{M} = \texttt{suitable\_module(A)} \wedge \texttt{can(A, derive, B)}$	$D20$
A cannot-useannotation B		
@B A	$\Longrightarrow$ move(A, M), if $\texttt{M} = \texttt{suitable\_module(A)} \wedge \texttt{M} \neq \texttt{M}_\texttt{A}$	$D21$
@B A	$\Longrightarrow$ remove([@B]) if $\texttt{M}_\texttt{A} = \texttt{suitable\_module(A)}$	$D22$
@B g(p)	$\Longrightarrow$ move(g, C), if $\texttt{C} \in \texttt{suitable\_module(g)}$	$D23$
@B g(p)	$\Longrightarrow$ remove([@B]) if $\texttt{M}_\texttt{A} \neq \texttt{suitable\_module(A)}$	$D24$
A must-derive B		
A	$\Longrightarrow$ replace([A], [A derive B]), if $\texttt{M}_\texttt{A} = \texttt{suitable\_module(A)} \wedge \texttt{typecheck}(\texttt{[A derive B]})$	$A1$
A	$\Longrightarrow$ move(A, M), if $\texttt{M} = \texttt{suitable\_module(A)} \wedge \texttt{M} \neq \texttt{M}_\texttt{A}$	$A2$
A must-throw B		
g (p){ S }	$\Longrightarrow$ replace([g (p){ S }] , [g (p) throws B { S }]), remove_treatment(S, B) if treat(B, S)	$A3$
g (p){ S }	$\Longrightarrow$ move(g, M) if $\texttt{M} = \texttt{suitable\_module(g)} \wedge \texttt{M} \neq \texttt{M}_\texttt{A}$	$A4$
A must-useannotation B		
A	$\Longrightarrow$ move(A, M), if $\texttt{M} = \texttt{suitable\_module(A)} \wedge \texttt{M} \neq \texttt{M}_\texttt{A} \wedge \texttt{target(B)} = \text{type}$	$A5$
A	$\Longrightarrow$ replace([A], [@B A]), if $\texttt{M}_\texttt{A} = \texttt{suitable\_module(A)} \wedge \texttt{target(B)} = \text{type}$	$A6$
g(p)	$\Longrightarrow$ move(g, M), if $\texttt{M} = \texttt{suitable\_module(A)} \wedge \texttt{M} \neq \texttt{M}_\texttt{A} \wedge \texttt{target(B)} = \text{method}$	$A7$
g(p)	$\Longrightarrow$ replace([g(p)], [@B g(p)]), if $\texttt{M}_\texttt{A} = \texttt{suitable\_module(g)} \wedge \texttt{target(B)} = \text{method}$	$A8$

IV. The DCLfix Tool

We have implemented a prototype tool called DCLfix that provides recommendations for architectural violations—as defined in Table II. Our tool has been implemented as an extension of the DCLcheck Eclipse plug-in and therefore exploits several preexisting data structures, such as the graph of existing dependencies, the defined architectural constraints, and the detected violations.

The current implementation has two main modules:

- *Auxiliary and Refactoring Functions*: This module implements the functions listed in Table I, such as searching for design patterns (e.g., factory), checking whether the refactored code type checks, and calculating the most suitable module, according to the sim function described in Section III-A. Some functions have already been implemented in DCLcheck (e.g., function can) and hence DCLfix simply reuses them.

- *Recommendation Engine*: Following the specification shown in Table II, this module is responsible for triggering the appropriate refactoring recommendation for a particular violation (if applicable). It first obtains information about the architectural violation—such as the violated constraint and the code where the violation is located—and then indicates the most appropriate refactoring.

As an example, suppose a constraint of the form only Factory can−create Product. Suppose also a class Client that creates an instance of Product. When the maintainer requests a recommendation, DCLfix indicates the most appropriate refactoring. For example, Figure 2 illustrates the DCLfix interface when providing a recommendation for the previously mentioned constraint. Basically, the provided recommendation suggests replacing the direct instantiation of the Product with the respective factory method (which corresponds to the recommendation $D11$ in Table II).

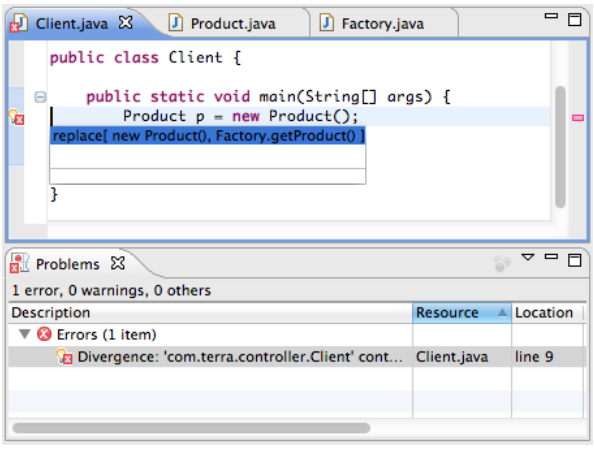

Figure 2. DCLfix interface

V. Evaluation

A. Research Question

We designed a study to address the following overarching research question:

RQ – Do developers consider our refactoring recommendations useful when repairing software architecture violations using DCL?

B. Subjects

Our evaluation relies on two Java-based systems. The first one is an open-source strategic management system, called Geplanes[2]. The system handles strategic management activities, including management plans, goals, performance indicators, actions, etc. The second one is the 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 TCom. This large-size and complex system handles a full range of customer related services, including account activation, claims and inquiries, offers, etc. Table III shows information about the size of both systems.

Table III
Target systems

	Geplanes	TCom
LOC	21,799	728,814
Subsystems	1	146
Packages	25	2,289
Classes	278	4,724
Interfaces	1	1,893
External libraries	47	58

C. Methodology

To provide an answer to our research question, we performed the following tasks:

1) Architectural constraints definition: First, the systems' architects defined the architectural constraints in natural language. Next, we translated this preliminary definitions to DCL, and validated the resulting constraints with the architects.

2) Architecture conformance process: Using as input the constraints defined in the previous step, we executed the DCLcheck tool to discover architectural violations in both systems. We also validated with the architects whether the indicated divergences and absences are in fact true violations, because some violations may in fact represent exceptions to general rules.

3) Usefulness evaluation: Using as input the violations raised in the previous step, we executed the DCLfix tool to provide refactoring recommendations. Finally, we checked with the architects the usefulness of the provided recommendations. For each violation, we showed the triggered

[2]http://www.softwarepublico.gov.br

Table IV
GEPLANES RESULTS

Constraint		# violations	Recommendations (*useful – pr. useful – not useful*)	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.annotation.DescriptionProperty	18	A6(18−0−0)	18−0−0
GP5	MAController **must-useannotation** linkcom.neo.controller.DefaultAction	1	A6(1−0−0)	1−0−0
GP6	MAController **must-useannotation** linkcom.neo.controller.Controller	2	A6(1−0−1)	1−0−1
GP7	GService **must-useannotation** linkcom.neo.bean.annotation.ServiceBean	1	A6(1−0−0)	1−0−0
GP8	**only** Entities **can-useannotation** javax.persistence.Entity	1	D21(1−0−0)	1−0−0
GP9	**only** MAController **can-useannotation** linkcom.neo.controller.Input	1	D24(1−0−0)	1−0−0
GP10	*$system* **cannot-create** GService, GDAO, MAController	5	D12(2−3−0)	2−3−0
GP11	GDAO **cannot-create** QBuilder	7	D11(2−0−0); D13(0−0−5)	2−0−5
		41		**31−3−7**

recommendation to the architects who classified them as *useful*, *partially useful*, or *not useful*. We instructed the architects to classify a recommendation as *useful* when it is the appropriate solution to the detected violation, as *not useful* when it is definitively not part of the architectural fix, and as *partially useful* when the recommendation is only part of the required refactoring (e.g., a violation whose fixing involves replacing an annotation by a new one, but DCLfix has only suggested inserting the new annotation, without a suggestion to remove the existing one).

It is important to mention that we have only considered the recommendations defined in Table II, which have been collected from a previous architecture conformance process we have been involved with [16]. In other words, we do not refine or extend the available recommendations when evaluating the Geplanes and TCom systems.

D. Geplanes Results

The results achieved after aplying our methodology to Geplanes are discussed next.

Task #1: As reported in Table IV, Geplanes' architect has defined 15 architectural constraints, mainly related to rules prescribed by the MVC-based framework used by Geplanes' current implementation. Specifically, 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 an example of the first group of constraints, constraint GP5 specifies that subclasses of MultiActionController must have at least one method being annotated by the DefaultAction annotation; as an example of the second group of constraints, GP8 forbids classes that do not belong to the Entities module to be annotated as Entity; and as an example of the last group of constraints, GP10 forbids any class to create subclasses of GenericService, GenericDAO, or MultiActionController.

It was not possible to translate four constraints to DCL and therefore they were disregarded. We could not translate such constraints because DCL only supports constraints at the class level, and these particular restrictions operate at the field or method level. For example, they prescribe that only some methods can receive a particular annotation or that classes must declare a variable of a specific type only once.

Task #2: Using as input the constraints defined and validated in the previous step, we have executed the DCLcheck tool. Initially, the tool reported 74 architectural violations. After that, Geplanes' architect carefully analyzed the reported violations and decided to refine four constraints. More specifically, this revision was motivated by a minor misunderstanding originally made in the definition of some modules. After the revision, we executed again the conformance tool and 41 violations were raised. Table IV shows the number of violations raised for each of the constraint considered in the study.

Task #3: We executed the DCLfix tool to provide refactoring recommendations for each violation discovered in the previous task. Next, we showed the violations to Geplanes' architect, who ranked them as *useful*, *partially useful*, and *not useful*. Table IV shows the recommendations triggered for each violation and the results of the architect's classification. As can be observed, a total of 31 recommendations have been ranked as *useful* (75%), three recommendation have been ranked as *partially useful* (7%), and seven recommendations have been considered *not useful* (18%).

Most of the 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 had been detected (recs. $A6$ and $A8$). For the 27 recommendations provided for such constraints, only two were classified as *not useful*. For example, in the case of GP1, a recommendation to move the class to another module (rec. $A8$) was not accepted because according to the architect the class should be turned abstract instead.

Regarding the violations due to constraint GP10, DCLfix suggested removing the new operator responsible by the violation (rec. $D12$), since the required objects

Table V
TCOM RESULTS

Constraint		# violations	Recommendations *(useful – pr. useful – not useful)*	Total
TC1	DTO **must-implement** java.io.Serializable	63	A1(50−0−0); A2(0−0−13)	50−0−13
TC2	SAO **must-extend** tcom.server.sao.AbstractSAO	1	A2(0−1−0)	0−1−0
TC3	Controller **must-useannotation** tcom.client.controller.Controller	1	A6(1−0−0)	1−0−0
TC4	DataSource **must-useannotation** tcom.client.datasource.annotation.DataSource	1	A6(1−0−0)	1−0−0
TC5	**only** tcom.server.persistence.dao.BaseJPADAO **can-create** DAO	13	D11(13−0−0)	13−0−0
TC6	**only** DAO **can-throw** tcom.server.persistence.dao.common.DAOException	15	D15(11−0−0); D16(2−0−0)	13−0−0
TC7	**only** ControllerLayer, DataSourceLayer **can-useannotation** AnnotationCtrlDS	20	D21(2−0−0); D22(18−0−0)	20−0−0
TC8	**only** ServiceImpl, SAOLayer **can-depend** SAO	5	D20(1−0−0)	1−0−0
TC9	*$system* **cannot-create** Controller, DataSource	3	D12(3−0−0)	3−0−0
TC10	ControllerLayer **cannot-create** java.util.Date	84	D13(0−84−0)	0−84−0
TC11	ScreenWrappers **cannot-useannotation** java.lang.annotation.Annotation+	18	D21(0−0−5); D22(13−0−0)	13−0−5
TC12	*$system* **cannot-depend** java.lang.System	14	D9(14−0−0)	14−0−0
TC13	ServiceAsync **cannot-declare** UnallowedAbstractTypes	270	D2(270−0−0)	270−0−0
TC14	Server **cannot-depend** ClientUtil	279	D7(225−0−0)	225−0−0
		787		**624−85−18**

are supposed to be provided by dependency injection techniques. For this particular constraint, three out of the five recommendations have been considered *partially useful* by the architect because, besides removing the instantiation, the architect has also indicated the need to create a setter method. For the constraint GP11, DCLfix was able to find a factory class in two cases (rec. $D11$). In the remaining five violations, the tool suggested the creation of a new factory (rec. $D13$), which has not been considered useful by the architect. In fact, the architect ascribed these violations to a missing functionality in the application development framework used by Geplanes, that does not provide means to create query builder objects parameterized by wrapper types, such as Integer.

E. TCom Results

The results for TCom are discussed next.

Task #1: TCom's architect has defined 18 architectural constraints. However, four constraints did not raise any architectural violation and therefore they are not discussed in this subsection. Table V lists the constraints with at least one violation. Constraints TC1 and TC2 require that classes from particular modules derive from a specified base type. Similarly, constraints TC3 and TC4 require that Controller and DataSource classes receive particular annotations. Constraint TC5 specifies the factory class for Data Access Objects (DAOs) and constraint TC6 prescribes that only DAOs can throw DAOException. Complementing constraints TC3 and TC4, constraint TC7 forbids annotations from the AnnotationCtrlDS module to be used outside Controller and DataSource modules.

Constraint TC8 specifies the only two modules that can establish dependencies with the SAO module. Constraint TC9 forbids any class to create Controller or DataSource classes, because objects of such classes can only be created by the underlying application development framework. Similarly, constraint TC10 prescribes that the Controller layer must not create Date objects, in order to avoid time synchronization inconsistencies because Controller objects are located on the client side. Constraint TC11 forbids ScreenWrappers to receive any annotation and constraint TC12 forbids any class to establish dependencies with the Java API System class (to avoid for example calls to System.out.println, since TCom is a web-based system). Due to a pattern strongly recommended by the GWT framework, constraint TC13 forbids ServiceAsync classes to declare abstract types such as Collection and Map. Finally, constraint TC14 forbids server classes to use Util classes designed for use only on the client side.

Task #2: After the architect refined the initially defined constraints, the number of violations decreased from 978 to 787 (as reported in Table V). The refinements were mainly related to test classes that should not have been included in the conformance process.

Task #3: DCLfix has triggered recommendations for 727 violations (92%). As presented in Table V, 624 recommendations have been ranked as *useful* (85%), 85 recommendations as *partially useful* (13%), and 18 recommendations as *not useful* (2%).

Constraint TC1 raised violations in 63 classes. For 50 classes, DCLfix suggested the right refactoring according to TCom's architect, which is making the class implement Serializable (rec. A1). However, for 13 classes DCLfix suggested instead moving the class responsible by the violation to another module (rec. A2). Basically, most of such classes are Data Transfer Objects (DTOs), which by their nature rely extensively on types from the Java API. For this reason, DCLfix—based on the most suitable module calculated by the sim function described in Section III-A—has improperly recommended moving the classes to the Constants module, whose classes are also heavily based on Java's built-in types.

DCLfix suggested accurate refactorings for the 13 violations of constraint TC5. Essentially, the recommendations

in such cases prescribe the use of an existing factory method (rec. D11). Furthermore, the conformance process discovered 15 methods that throw DAOException, but are not located in DAO modules as required by constraint TC6. In the case of 11 methods, DCLfix suggested the right refactoring according to TCom's architect, which basically consists in removing the exception from the throws clause (rec. D15). In addition, for two methods, DCLfix suggested removing the throws clause and handling the exception internally (rec. D16), which was the right refactoring in this particular context. Finally, there were only two cases where DCLfix could not provide a recommendation because the repair prescribes the raise of a different exception type and the current recommendations are not prepared to suggest replacing an exception by another one.

Constraint TC7 raised 20 violations and DCLfix provided *useful* recommendations for all of them. Basically, DCLfix could determine that 18 violations had been raised in classes already located in their most suitable module—as returned by the sim function when called without any dependency type filter. Based on this finding, the recommendation engine indicated the right refactoring, which prescribes removing the annotation defined in the constraint (rec. $D22$). Moreover, DCLfix could also determine that two classes were located in the wrong modules. Because most of the dependencies established by such classes were with DataSource types, such as DataSourceRecordIdentifier, the engine precisely recommended moving the classes to the DataSource module (rec. D21).

Regarding the violations due to constraints TC9 (three violations) and TC12 (14 violations), the recommendation engine indicated the right refactoring in each case, i.e., removing the new operator (rec. D12) and the references to the System class (rec. D9), respectively. Finally, the highest number of useful recommendations for a single constraint has been raised for the 270 violations associated to constraint TC13. For each detected violation, DCLfix could suggest a more specialized type (rec. D2). For instance, most of the violations occurred because the used type was the interface List and DCLfix properly suggested replacing it with ArrayList.

F. Discussion

Based on the experience gained with the Geplanes and TCom case studies, this subsection includes a critical analysis on the recommender approach proposed in this paper. Our analysis is based on the following criteria:

Applicability: The evaluation provided us with encouraging feedback on the application of our approach. Considering two real-world systems, we could trigger recommendations for 92% of the detected violations. Moreover, the architects ranked 85% of the raised recommendations as the most appropriate architectural repair solution. As a practical result of our evaluation, the architects of both systems have opened a maintenance request in the issue management platform of the evaluated systems requesting a correction for the detected violations and suggesting the use of the recommendations provided by DCLfix.

Coverage: Table VI provides an overview of the recommendations used in both systems, including the usefulness classification made by the respective software architects. First of all, we can observe that 17 out of the 32 proposed recommendations have been triggered at least once in our case studies (the recommendations not used in any of the considered systems have not been listed in this table). However, we truly believe that the unused recommendations are generic enough to be triggered in other architecture erosion fixing contexts, since they have emerged in our first conformance experience using the DCL language [16]. For example, the unused recommendation D1—which suggests replacing the declaration of an unauthorized type (e.g., ProductHibernateDAO) with one of its supertypes (e.g., IProductDAO)—has proven itself very useful in our training system.

Table VI
RECOMMENDATIONS TRIGGERED IN THE CASE STUDIES AND THEIR CLASSIFICATION AS USEFUL – PR. USEFUL – NOT USEFUL

$D2$	270−0−0	$D15$	11−0−0	$A1$	50−0−0
$D7$	225−0−0	$D16$	2−0−0	$A2$	0−1−13
$D9$	14−0−0	$D20$	1−0−0	$A5$	1−0−1
$D11$	15−0−0	$D21$	3−0−5	$A6$	24−0−1
$D12$	5−3−0	$D22$	31−0−0	$A8$	2−0−0
$D13$	0−89−0	$D24$	1−0−0		

Usefulness classification: As presented in Table VI, 13 out the 17 recommendations triggered in the case studies have been massively classified as useful by the respective software architects. Particularly, TCom's architect has highlighted the 270 violations fixed by recommendation $D2$ as one of the "most important recommendations" provided by DCLfix. According to the architect, by not following this recommendation (associated to the correct use of the GWT framework), the current implementation of TCom experiences an overhead in terms of the size of the generated Javascript code which has important consequences both in CPU performance and network bandwidth consumption.

On the other hand, recommendations D12, D13, D21, and A2 have presented the lowest usefulness rate. Particularly, recommendations D21 and A2 have been mostly ranked as *not useful*. In fact, this result is explained by our current policy on recommending just the best result provided by the suitable_module function. More specifically, we noticed in many cases that the correct recommendation was in fact the module with the second best Jaccard's coefficient, whose value was very close to the highest calculated coefficient. For this reason, we are investigating a revision in our current priorization policy, to include second or third-hand recommendations in particular contexts.

G. Threats to Validity

We must state at least four threats inherent to the reported evaluation. First, although we have used two industrial-strength systems, we cannot claim that our approach will provide equivalent results in other systems. Second, since the subject systems have presented a moderate number of violations, we cannot claim that our approach will provide the same precision in systems already facing a major architectural erosion process. Third, since the proposed architectural fix recommendations are tightly coupled to our previously designed constraint language (DCL), we are limited to the violations detected by this language. For instance, four constraints in the Geplanes case study could not be translated to DCL. On the other hand, DCL has proved itself to be able to express all constraints proposed for two large and complex systems (TCom, as reported in this paper, and SGP, a 220 KLOC system employed in a previous paper [16]). Finally, since we have used a little over half of the proposed refactoring recommendations, it was not possible to evaluate the applicability of the unused recommendations.

VI. Related Work

We divided the related work into two groups: refactoring recommendation tools and remodularization approaches.

Refactoring Recommendation Tools: Tsantalis and Chatzigeorgiou have proposed a semi-automatic approach to identify Move Method refactoring opportunities [21]. Later, using an adaptation of program slicing techniques, they extended their tool to also identify Extract Method refactorings [22]. O'Keeffe and O'Cinneide have proposed a search-based software maintenance tool that relies on search algorithms, such as Hill Climbing and Simulated Annealing, to automatically suggest six inheritance-related refactorings (Push Down Field/Method, Pull Up Field/Method, and Extract/Collapse Hierarchy) [23]. In general terms, the ultimate goal of the above-mentioned tools is to suggest refactorings that can improve the internal quality of the code—for example, in terms of coupling and cohesion. On the other hand, the refactoring recommendation engine we have proposed in this paper aims to help developers handle the massive number of violations usually discovered as the result of an architecture conformance process [16], [18].

Remodularization Approaches: Rama and Patel have analyzed several software modularization projects in order to define recurring modularization patterns, called modularization operators by the authors [24]. More specifically, they have formalized the following operators: module decomposition/union, file/function/data transfer, and function to API promotion. However, they do not provide tool support for applying the proposed operators. Hierarchical clustering is another technique commonly proposed to evaluate alternative software decompositions [25], [26]. However, the effectiveness of clustering in reengineering tasks is often challenged. For example, using Eclipse as case study, Anquetil et al. have recently shown that software remodularizations do not necessarily improve modularity in terms of cohesion/coupling. Therefore, this finding undermines the validity of techniques as clustering centered on a function of similarity that aims to minimize coupling and maximize cohesion [27]. Glorie et al. have reported an experiment in which clustering and formal concept analyses have failed to produce an acceptable partitioning of a monolithic medical imaging application [28]. They have reported, for example, that some of the extracted clusters have components that according to domain experts were not semantically related.

Chern and De Volter have claimed that the static-dynamic coupling—i.e., the degree to which changes in a program's modular structure imply changes in its dynamic behavior—is a major obstacle to software remodularization efforts [29]. To alleviate this kind of coupling, they have proposed a new language, called SubjectJ, that allows class implementations to be split across different files. However, it is not clear whether the new classes provide the key benefits of modularity, such as parallel development, comprehensibility, and changeability.

VII. Conclusions

Architectural erosion is a recurrent problem faced by software architects. Despite several approaches have been proposed in the literature to uncover architectural violations, less research effort has been dedicated to properly repair the detected violations. To tackle this lack of effort, we have proposed an architecture-based recommendation system that provides refactoring guidelines for maintainers when repairing architectural violations.

The proposed recommendation system provides recommendations for architectural violations—divergences and absences—raised by DCL constraints. More important, we have conducted an evaluation with two industrial-strength systems that provided us with encouraging feedback on the application of our approach. Considering both systems, `DCLfix` has been able to indicate the most appropriate refactoring recommendation for 655 out of 828 violations detected as the result of an architecture conformance process.

Future work involves the refinement of the refactoring recommendations and then the evaluation of `DCLfix` in other systems. In addition, we have plans to design an architectural recommendation language—which exploits our existing auxiliary and refactoring functions—and hence allowing maintainers to produce their own domain-specific refactoring recommendations.

The `DCLfix` tool—including its source code—is publicly available at `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 (TCom) and Rógel Garcia (Geplanes) for the valuable collaboration in the case studies.

REFERENCES

[1] D. L. Parnas, "Software aging," in *16th International Conference on Software Engineering (ICSE)*, 1994, pp. 279–287.

[2] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[3] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, pp. 105–119, 2002.

[4] J. Knodel, D. Muthig, M. Naab, and M. Lindvall, "Static evaluation of software architectures," in *10th European Conference on Software Maintenance and Reengineering (CSMR)*, 2006, pp. 279–294.

[5] M. Lindvall and D. Muthig, "Bridging the software architecture gap," *Computer*, vol. 41, no. 6, 2008.

[6] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, and S. Sivagnanam, "Modularization of a large-scale business application: A case study," *IEEE Software*, vol. 26, pp. 28–35, 2009.

[7] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, 2012.

[8] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007, p. 12.

[9] L. Passos, R. Terra, R. Diniz, M. T. Valente, and N. Mendona., "Static architecture-conformance checking: An illustrative overview," *IEEE Software*, vol. 27, no. 5, pp. 82–89, 2010.

[10] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[11] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *3rd Symposium on Foundations of Software Engineering (FSE)*, 1995, pp. 18–28.

[12] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views: A case study," *Computer Languages, Systems & Structures*, vol. 32, no. 2-3, pp. 140–156, 2006.

[13] J. Brunet, D. Guerreiro, and J. Figueiredo, "Structural conformance checking with design tests: An evaluation of usability and scalability," in *27th International Conference on Software Maintenance (ICSM)*, 2011, pp. 143–152.

[14] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: connecting software architecture to implementation," in *22nd International Conference on Software Engineering (ICSE)*, 2002, pp. 187–197.

[15] D. Hou and H. J. Hoover, "Using SCL to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, 2006.

[16] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software: Practice and Experience*, vol. 32, no. 12, pp. 1073–1094, 2009.

[17] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 391–400.

[18] J. Knodel, D. Muthig, U. Haury, and G. Meier, "Architecture compliance checking - experiences from successful technology transfer to industry," in *12th European Conference on Software Maintenance and Reengineering (CSMR)*, 2008, pp. 43–52.

[19] F. Steimann and A. Thies, "From public to private to absent: Refactoring Java programs under constrained accessibility," in *23rd European Conference on Object-Oriented Programming (ECOOP)*, 2009, pp. 419–443.

[20] R. Terra, M. T. Valente, K. Czarnecki, and R. Bigonha, "Recommending refactorings to reverse software architecture erosion." in *16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track*, 2012, pp. 335–340.

[21] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 99, pp. 347–367, 2009.

[22] ——, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[23] M. K. O'Keeffe and M. Ó. Cinnéide, "Search-based software maintenance," in *10th European Conference on Software Maintenance and Reengineering (CSMR)*, 2006, pp. 249–260.

[24] G. M. Rama and N. Patel, "Software modularization operators," in *26th International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[25] N. Anquetil and T. Lethbridge, "Experiments with clustering as a software remodularization method," in *6th Working Conference on Reverse Engineering (WCRE)*, 1999, pp. 235–255.

[26] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[27] N. Anquetil and J. Laval, "Legacy software restructuring: Analyzing a concrete case," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011, pp. 279–286.

[28] M. Glorie, A. Zaidman, A. van Deursen, and L. Hofland, "Splitting a large software repository for easing future software evolution - an industrial experience report," *Journal of Software Maintenance*, vol. 21, no. 2, pp. 113–141, 2009.

[29] R. Chern and K. D. Volder, "The impact of static-dynamic coupling on remodularization," in *23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2008, pp. 261–276.