DCLfix: A Recommendation System for Repairing Architectural Violations

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

[1]Universidade Federal de Minas Gerais, Brazil

[2]University of Waterloo, Canada

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

Abstract. *Architectural erosion is a recurrent problem in software evolution. Despite this fact, the process is usually tackled in ad hoc ways, without adequate tool support at the architecture level. To address this shortcoming, this paper presents a recommendation system—called DCLfix—that provides refactoring guidelines for maintainers when tackling architectural erosion. In short, DCLfix suggests refactoring recommendations for violations detected after an architecture conformance process using DCL, an architectural constraint language.*

1. Introduction

Software architecture erosion is a recurrent problem in software evolution. The phenomenon designates the progressive gap normally observed between two architectures: the *planned architecture* defined during the architectural design phase and the *concrete architecture* defined by the current implementation of the software system [3]. Although the causes for this architectural gap are diverse—ranging from conflicting requirements to deadline pressures—when the process is accumulated over years, architectural erosion can transform software architectures into unmanageable monoliths [5].

Although several architecture conformance approaches have been proposed to detect architectural violations (e.g., reflexion models, intensional views, design tests, query languages, and architecture description languages [3]), there has been less research effort dedicated to the task of repairing such violations. As a consequence, developers usually perform the task in ad hoc ways, without tool support at the architectural level. We argue that the task of repairing architectural violations can no longer be addressed in ad hoc ways because *architecture repair* is as important as *architecture checking*.

To address this shortcoming, this paper presents `DCLfix`, a recommendation system that provides refactoring guidelines for maintainers to repairing architectural erosion. Specifically, it suggests refactoring recommendations for violations detected as the result of an architecture conformance process using DCL, an architectural constraint language.

The remainder of this paper is structured as follows. Section 2 presents the design and implementation of the `DCLfix` tool, including background and examples. Section 3 discusses related tools and Section 4 presents final remarks.

2. The `DCLfix` tool

As illustrated in Figure 1, `DCLfix`—based on a set of DCL constraints (specified by the software architect), a set of architectural violations (raised by the `DCLcheck` conformance

tool), and the source code of the system—provides a set of refactoring recommendations to guide the process of removing the detected violations. For instance, in order to repair a particular violation, DCLfix may suggest the use of a *Move Class* refactoring, including the indication of the most suitable module.

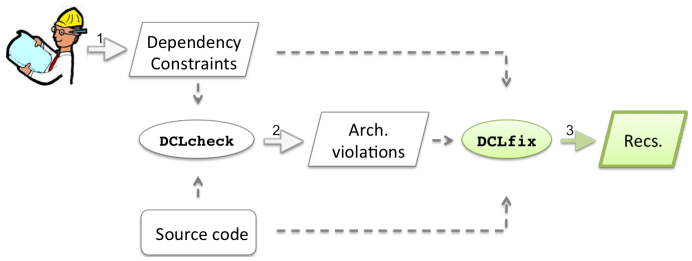


Figure 1. DCLfix recommendation engine

We first provide an overview of DCL (Subsection 2.1). Next, we introduce the refactoring recommendations triggered by DCLfix (Subsection 2.2). Finally, we present the design and implementation of DCLfix followed by examples extracted from real case studies (Subsections 2.3 and 2.4).

2.1. DCL

The Dependency Constraint Language (DCL) is a domain-specific language that allows architects to restrict the spectrum of structural dependencies, which can be established in object-oriented systems [6]. Particularly, the language allows architects to specify that dependencies *only can*, *can only*, *cannot*, or *must* be established by specified modules. In DCL, a module is a set of classes. Moreover, it also allows architects to define the type of the dependency (e.g., `access`, `declare`, `create`, `extend`, etc.). In order to explain the differences, let us assume the following constraints:

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

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

In a previous paper [6]—where a complete description of DCL can be found—we have also described the DCLcheck tool that checks whether DCL constraints are respected by the source code of the target system. DCLfix operates on the violations detected by this tool in order to provide refactoring recommendations.

2.2. Refactoring Recommendations

To provide recommendations, DCLfix relies on a set of 32 refactoring recommendations formalized in previous papers [8, 7]. Table 1 shows a subset of the recommendations. As an example, consider a violation in which an unauthorized class $A \in M_A$ has created an

object of a class $\texttt{B} \in \texttt{M}_\texttt{B}$. In this case, DCLfix might trigger recommendation D11 that suggests the replacement of the *new* operator with a call to the *get* method of a Factory class.

Table 1. Subset of Refactoring Recommendations

A cannot-declare B		
B b; S	$\Longrightarrow$ replace([B], [B′]), if $\texttt{B}' \in \texttt{super(B)} \wedge$ typecheck([B′ b ; S]) $\wedge\ \texttt{B}' \notin \texttt{M}_\texttt{B}$	$D1$
A cannot-create B		
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [FB.getB(exp)]), if $\texttt{FB} = \texttt{factory(B, [exp])} \wedge$ can(A, access, FB)	$D11$
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [null]), if $\overline{\texttt{M}_\texttt{A}} = \emptyset$	$D12$
A must-derive B		
A	$\Longrightarrow$ replace([A], [A derive B]), if $\texttt{M}_\texttt{A} = \texttt{suitable\_module(A)} \wedge$ typecheck([A derive B])	$A3$
A	$\Longrightarrow$ move(A, M), if $\texttt{M} = \texttt{suitable\_module(A)} \wedge\ \texttt{M} \neq \texttt{M}_\texttt{A}$	$A4$
A must-useannotation B		
A	$\Longrightarrow$ replace([A], [@B A]), if $\texttt{M}_\texttt{A} = \texttt{suitable\_module(A)} \wedge \texttt{target(B)} = \text{type}$	$A6$

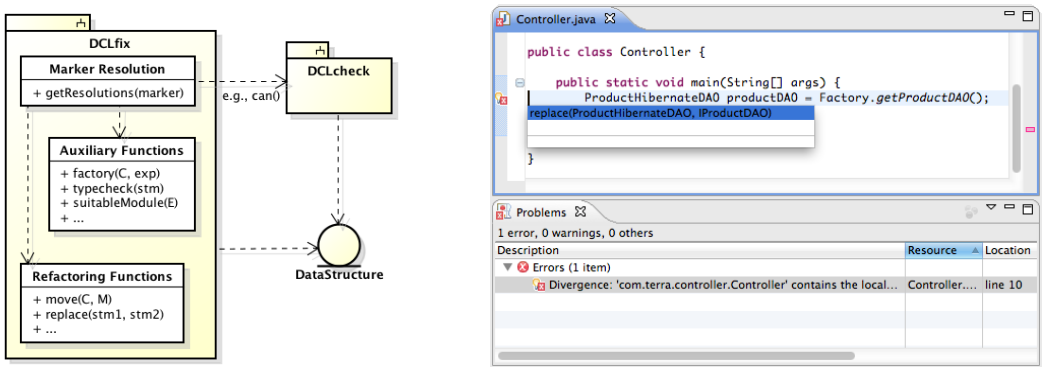
Many recommendations (e.g., A3, A4, and A6) rely on the suitable_module function. This function considers a class or a module as the set of dependencies that it establishes with other program elements. Based on the *Jaccard Similarity Coefficient*—a statistical measure for the similarity between two sets—it returns the module of the system with the highest similarity. For example, a class that relies extensively on GUI types is likely to have View as its suitable module. Due to space restrictions, this paper does not provide the description of other refactoring and auxiliary functions used in Table 1, such as move, replace, and typecheck. A complete description of these functions and also the entire set of refactoring recommendations can be found at [7].

2.3. Internal Architecture and Interface

We have implemented DCLfix as an extension of the DCLcheck Eclipse plug-in. As illustrated in Figure 2a, DCLfix exploits preexisting data structures, such as the graph of existing dependencies, the defined architectural constraints, and the detected violations. Moreover, DCLfix also reuses functions implemented in DCLcheck, e.g., to check whether a type can establish a particular dependency with another type.

The current DCLfix implementation follows an architecture with three main modules:

- *Recommendation Engine*: This module is responsible for determining the appropriate refactoring recommendation for a particular violation. More specifically, DCLfix has been designed as a marker resolution because DCLcheck marks architectural violations on the source code (see Figure 2b). In short, it first obtains information about the architectural violation (e.g., violated constraint and code location), and then, using the auxiliary functions, searches for potential refactoring recommendations.
- *Auxiliary Functions*: This module implements the auxiliary functions used in the preconditions of refactoring recommendations, such as checking whether the refactored code type checks (typecheck), searching for design patterns (e.g., factory), and calculating the most suitable module (suitable_module).



(a) Architecture (b) Interface

Figure 2. `DCLfix` architecture (2a) and interface (2b)

- *Refactoring Functions*: This module is responsible for applying the refactorings in the source code, e.g., the `replace` and `move` functions. This module is still under development.

As an example, consider a constraint of the form `Controller cannot−depend HibernateDAO`. This constraint prevents the Controller layer from manipulating directly Hibernate Data Access Objects (DAOs). Assume also a class `Controller` that declares a variable of a type `ProductHibernateDAO`. When the developer requests a recommended fix for such violation, `DCLfix` indicates the most appropriate refactoring (see Figure 2b). The provided recommendation suggests replacing the declaration of the unauthorized type `ProductHibernateDAO` with its interface `IProductDAO` (which corresponds to recommendation `D1` in Table 1). This recommendation is particularly useful to handle violations due to references to a concrete implementation of a service, instead of its general interface.

2.4. Applications

In our previous paper [7], we have evaluated the application of our tool in two industrial-strength systems: (i) Geplanes, an open-source strategic management system, in which `DCLfix` triggered correct refactoring for 31 out of 41 violations; (ii) TCom, a large customer care system used by a telecommunication company, in which `DCLfix` triggered correct refactoring for 624 out of 787 violations.

Table 2 summarizes some results obtained from the evaluation of the aforementioned systems, including the constraint description, the number of raised violations, and the triggered refactoring recommendations. In order to illustrate the recommendations provided by `DCLfix`, we have chosen one constraint from Geplanes (`GP4`) and three constraints from TCom (`TC1`, `TC5`, and `TC9`).

As a first example, constraint `GP4` states that every class in the `Entities` module must be annotated by `DescriptionProperty`. This constraint ensures a rule prescribed by the underlying framework in which every persistent class has to set a description property. As the result of an architectural conformance process, `DCLcheck` has detected some classes in the `Entities` module without such annotation. Because these classes were located in their correct module (as certified by `suitable_module` function), `DCLfix` has correctly suggested adding the class-type annotation to them (rec. `A6`).

Table 2. Geplanes and TCom results

Constraint		# Violations	Correct Recs.
GP4	Entities **must-useannotation** linkcom.neo.bean.annotation.DescriptionProperty	18	A6 (18 cases)
TC1	DTO **must-implement** java.io.Serializable	63	A3 (50 cases)
TC5	**only** tcom.server.persistence.dao.BaseJPADAO **can-create** DAO	13	D11 (13 cases)
TC9	*$system* **cannot-create** Controller, DataSource	3	D12 (3 cases)

As a second example, constraint TC1 prescribes the serialization of Data Transfer Object (DTOs). For 50 out of 63 violations, DCLfix has suggested adding the implementation of Serializable (rec. A3). For the other violations, DCLfix improperly triggered the recommendation A4 because the suitable_module function considered them as Constant instead of DTO classes. The reason is that both DTO and Constant classes rely heavily on Java's built-in types and therefore are structurally very similar.

As another example, constraint TC5 specifies a factory class for DAOs. DCLcheck has indicated instantiations of DAO objects outside the factory. In this case, DCLfix was able to find the factory and suggested replacing the instantiation with a call to the factory (rec. D11). As a last example, constraint TC9 forbids any class of the system to create objects of Controller or DataSource classes. In fact, these objects must be created by dependency injection techniques and thus no class of the system is allowed to create them. As a result, DCLfix has correctly suggested the removal of the instantiation statements (rec. D12).

3. Related Tools

Recommendation Systems for Software Engineering (RSSEs) are ready to become part of industrial software developers' toolboxes [4]. Such systems usually help developers to find information and make decisions whenever they lack experience or cannot handle all available data. For example, since frameworks are usually large and difficult to understand, Strathcona [2] is a tool that recommends relevant source code fragments to help developers to use frameworks and APIs. Our approach is also realized as a recommendation system, but our focus is following the planned architecture, instead of using a framework.

As another example, SemDiff [1] recommends replacement methods for adapting code to a new library version, i.e., it finds suitable replacements for framework elements that were accessed by a client program but removed as part of the framework's evolution. Analogously, DCLfix provides suitable replacements for implementation decisions that denote violations in the software evolution.

As a last example, eRose [9] identifies program elements (classes, methods, and fields) that usually are changed together. For instance, when developers want to add a new preference to the Eclipse IDE and then change fKeys[] and initDefaults(), eRose would recommend changing also the Plugin.properties file, because, according to versioning system, they are always changed together. However, despite of a trend towards the use of recommendation systems in software engineering, we are not aware of recommendation systems whose precise goal is to help developers and maintainers in tackling the architectural erosion process.

4. Final Remarks

Architectural erosion is a recurrent problem in software evolution. Although many approaches and commercial tools have been proposed to detect architectural violations, there has been less research effort dedicated to the task of repairing violations. Developers usually perform the task of fixing violations in ad hoc ways, without tool support at the architectural level.

To overcome these difficulties, we have developed `DCLfix`—a solution based on recommendation system principles—that provides refactoring guidelines for developers when repairing architectural violations. It prevents developers to waste a long time on determining the proper fix or to introduce new violations while fixing one. Even though the good results obtained in our previous evaluation with two industrial-strength systems [7], we are still evaluating `DCLfix` with other systems and our plan is to allow developers to extend `DCLfix` with their own domain-specific refactorings 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.

References

[1] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *30th International Conference on Software Engineering (ICSE)*, pages 481–490, 2008.

[2] R. Holmes, R. Walker, and G. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[3] L. Passos, R. Terra, R. Diniz, M. T. Valente, and N. Mendonça. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89, 2010.

[4] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.

[5] 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*, 26:28–35, 2009.

[6] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009.

[7] R. Terra, M. T. Valente, K. Czarnecki, and R. Bigonha. A recommendation system for repairing architectural violations. In *28th International Conference on Software Maintenance (ICSM)*, pages 1–10, 2012. (Submitted to).

[8] 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*, pages 335–340, 2012.

[9] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.