Journal of Universal Computer Science, vol. 23, no. 8 (2017), 769-805
submitted: 28/6/16, accepted: 27/8/17, appeared: 28/8/17 © J.UCS

Introducing an Architectural Conformance Process in Continuous Integration

Arthur F. Pinto, Ricardo Terra
(Federal University of Lavras, Lavras, Brazil
fparthur@posgrad.ufla.br, terra@dcc.ufla.br)

Eduardo Guerra, Fernanda São Sabbas
(Instituto Nacional de Pesquisas Espaciais, São José dos Campos, Brazil
{eduardo.guerra, fernanda.saosabbas}@inpe.br)

Abstract: As software evolves, developers usually introduce deviations from the planned architecture, due to unawareness, conflicting requirements, technical difficulties, deadlines, etc. This occurs in systems with an explicit division of responsibility between groups of classes, such as modules and layers. Although there are architectural conformance tools to identify architectural violations, these tools are underused and detected violations are rarely corrected. To address these shortcomings, this article introduces an architectural conformance process into continuous integration. Thus, the conformance process is triggered by every code integration and, when no violations are detected, the code is integrated into the repository. The implemented tool, called ArchCI, supports the proposed solution using DCL (Dependency Constraint Language) as underlying conformance technique and Jenkins as the Continuous Integration server. We also evaluated the applicability of our proposed solution in a real-world Java project where we incrementally introduced 44 constraints through six releases. As the result, our process was able to detect 42 violations, which have always been fixed before the ensuing release.

Key Words: software architecture erosion, architectural conformance, continuous integration
Category: D.2, D.2.2, D.2.7, D.2.9

1 Introduction

Software architecture is defined as the set of design decisions that are critical to the success of complex software systems. This includes how systems are structured into components and constraints on how components must interact [Fowler, 2002, Garlan and Shaw, 1996]. Assume two components *View* and *Model* that contain classes that handle, respectively, user interface and database persistence. As a constraint, a *View* component should not access a *Model* component directly; such a call is a violation.

As a software evolves, developers usually introduce deviations from the planned architecture, due to unawareness, conflicting requirements, technical difficulties, deadlines, new requirements, etc. [Perry and Wolf, 1992, Terra and Valente, 2009]. More important, the accumulation of architectural violations leads to the phenomenon known as software architectural erosion prob-

lem [Nierstrasz and Lungu, 2012, de Silva and Balasubramaniam, 2012], which negatively affects software maintainability, reusability, scalability, portability, etc. [Passos et al., 2010].

Architecture conformance solutions detect architectural violations, i.e., they identify design decisions that do not respect a defined architectural constraint [Sangal et al., 2005, Verbaere et al., 2008, Terra and Valente, 2009, Murphy et al., 1995]. Nevertheless, (i) these tools are underused and (ii) detected violations are rarely corrected [Alwis and Sillito, 2009, Terra et al., 2015]. As an example, Knodel et al. identified almost 5,000 architectural violations in the domain of portable measurement devices after three years without architectural tool support [Knodel et al., 2008a]. Terra et al. identified more than 2,200 architectural violations in a human-resource management system, which most violations have not been fixed so far [Terra and Valente, 2009]. As the last example, Sarkar et al. estimated 2,100 person-days to reconstruct the original architecture of a large banking application [Sarkar et al., 2009].

In order to address the software architectural erosion problem, this article proposes an architectural conformance solution that introduces incrementally the architectural constraints and integrates their verification in Continuous Integration (CI) [Pinto and Terra, 2015]. This implies that the architectural conformance process is triggered at each code integration—presenting a solution to the "*architectural tools are underused*" problem—and performing a configured action when violations are detected, such as blocking the integration to the remote repository or sending a technical debt alert to the software architect—which helps to solve the "*detected violations are rarely corrected*" problem. More important, we implemented ArchCI—a tool that supports the proposed solution using DCL (Dependency Constraint Language) as underlying conformance technique and Jenkins as the Continuous Integration server—to evaluate our proposed solution in a real-world Java project.

The remainder of this article is organized as follows. Section 2 introduces basic concepts, such as version control systems, continuous integration practices, and architectural conformance processes. Section 3 describes our proposed architectural conformance solution. Section 4 presents the design and implementation of ArchCI. Section 5 evaluates our solution in a real-world project. Finally, Section 6 discusses related work and Section 7 presents our main contributions and future work.

2 Background

This section introduces basic concepts for the understanding of this study. Section 2.1 describes version control systems, Section 2.2 presents continuous integration practices, and Section 2.3 discusses architectural conformance processes.

2.1 Version Control

VCS (*Version Control Systems*) are used to manage different versions of developing artifacts in a project [Spinellis, 2005a, Spinellis, 2005b]. As their main contribution, they provide traceability of the changes, such as when they were made, the responsible developer, and the differences between versions. In the context of this article, since project artifacts (source code, configuration files, etc.) are stored in the VCS, it includes the whole input an architectural analysis tool would require.

VCS can be centralized or distributed [Hinsen et al., 2009]. Centralized VCS have code repositories where access and writing are restricted to a group of developers [Rama and Patel, 2010]. Distributed VCS, on the other hand, rely on the *peer-to-peer* architecture. Thus, each copy of a project contains all the history and background related to changes made and the project metadata. This ensures developers the ability to share the changes the way that best suits their needs [O'Sullivan, 2009].

Among the main version control tools—CVS, SVN, Git, and Mercurial—we chose Git[1] for the development of this project because it offers the possibility of development in a centralized and distributed way [O'Sullivan, 2009], as well as being one of the most widely used VCS tools nowadays comparing the number of Google searches and the number of questions in Stack Overflow [RhodeCode, 2017].

In this article, it is important to contextualize the following concepts [Spinellis, 2005a, Spinellis, 2005b]: (i) *tag*, a symbolic name assigned to a specific release or a branch; (ii) *branch*, a set of evolving source file versions, identified by a *tag*; (iii) *commit*, command that integrates the changes from a developer into a local repository *branch*; and (iv) *push*, command that integrates a series of commits from a developer into a remote repository *branch*.

2.2 Continuous Integration

CI is the software development practice that triggers processes to ensure the project's integrity as soon as members of a team incorporate changes to the software [Fowler and Foemmel, 2006]. In this way, it is easier to detect the flaws and errors in the early stages of the project, seeking a lower cost repair [Berg, 2012].

CI servers can be configured to check when changes are made in a repository [Duvall et al., 2007]. It usually retrieves the latest versions of the classes, compiles the code, and then runs tests for the integration process, displaying the results to maintainers [Bowyer and Hughes, 2006]. Therefore, in the context of this article, an action could be implemented in a CI server to check whether the code changes comply with the architectural constraints.

[1] http://git-scm.com/

Among the most relevant CI servers—Jenkins, TeamCity, and CruiseControl—Jenkins[2] achieves a wider reach in the open-source community [White, 2014]. As a consequence, it has a certain advantage in identifying and fixing bugs, as well as being responsible for implementing certain improvements.

2.3 Architectural Conformance

Architectural erosion is defined as the phenomenon that occurs when the implemented architecture of a software diverges from its planned architecture [de Silva and Balasubramaniam, 2012]. There are several techniques—through the process of *architectural monitoring* or the definition of architectural constraints—that can be used in order to prevent architectural erosion, such as Reflexion Models [Murphy et al., 1995], Dependency Structure Matrices [Sangal et al., 2005], Source Code Query Languages [Verbaere et al., 2008], Programming Language Extensions [Aldrich et al., 2002], Design Tests [Brunet et al., 2011], and Constraint Languages [Terra and Valente, 2009]. A constraint language known as *Dependency Constraint Language* (DCL) is used in this project. We chose DCL as the underlying conformance solution due mainly to three reasons: (i) the simple and self-explanatory syntax, (ii) the expressiveness to treat the architectural erosion problem, and (iii) an open-source implementation for Java systems.

DCL Language: DCL is a domain-specific declarative language, supports the definition of structural constraints between modules in object-oriented systems [Terra and Valente, 2009]. By defining structural constraints through DCL, it becomes possible to find two types of architectural violations: divergences (when a dependency that exists in the source code is not in accordance with the system's architectural model) and absences (absent dependencies in the source code, which are mandatory according to the architectural model). Basically, this model covers any form of relationship and dependency between classes that can be statically verified. DCL relies on *modules* and on *architectural constraints* between the defined modules, which are defined according to the syntax described in Figure 1.

Modules: A module is a set of classes. Assume, for instance, the following module definitions:

```
1: module View:           org.foo.view.*, org.foo.ui.Table
2: module DataStructure: org.foo.util.**,
3: module Remote:         java.rmi.UnicastRemoteObject+
4: module Frame:          "org.foo.[a-zA-Z0-9/.]*Frame"
```

[2] http://jenkins-ci.org/

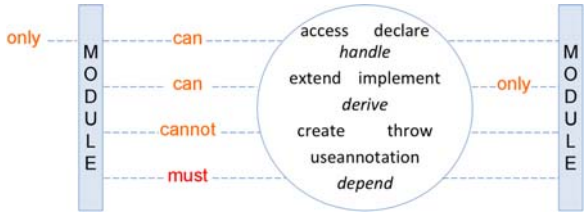


Figure 1: DCL Syntax

Module `View` includes all classes from package `org.foo.view` (operator *) and class `org.foo.ui.Table`. Module `DataStructure` includes all classes from package `org.foo.util` and all its subpackages (operator **). The + operator matches subtypes of a given type. For example, module `Remote` denotes all subclasses of `java.rmi.UnicastRemoteObject`. Last, module `Frame` is composed by any system class that matches that regular expression, i.e., classes whose qualified name starts with `org.foo.` and ends with `Frame`.

Constraints: Assume modules M_A and M_B—i.e., sets of classes—of a software system. Assume also that `dep` corresponds to the possible dependencies that can be specified through the DCL, such as *create*, *access*, *declare*, *handle*, etc. DCL provides four types of constraints: *cannot*, *can only*, *only can*, and *must*. A constraint on the form M_A `cannot`−*dep* M_B forbid classes from module M_A to depend on classes from module M_B. A constraint on the form `only` M_A `can`−*dep* M_B allows only classes from M_A to depend on classes from M_B. A constraint on the form M_A `can`−*dep*−`only` M_B allows classes from M_A to depend only on classes from M_B. The following example illustrates the definition of a few architectural constraints between modules:

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

The constraint at line 1 specifies that only objects from module `Factory` can create objects from module `Product`, which resembles a creational pattern. The constraint at line 2 specifies that the classes from module `Util` can establish dependencies only with the module `Util` itself and the default Java language library, which resembles a reusable component. The constraint at line 3 specifies that classes from module `View` cannot access classes in module `Model`, which resembles a layered architecture. And finally, constraint at line 4 specifies that all classes in module `Product` must implement the `Serializable` interface, which resembles the persistence of Data Transfer Objects (DTOs) [Fowler, 2002].

Semantics: Assume also that $\overline{M_A}$ is the complement of M_A, as well as $\overline{M_B}$ represents the complement of M_B. Therefore, the following semantic for finding violations can be assigned to M_A *cannot*-dep M_B constraint:

$$\exists \mathtt{A}\, \exists \mathtt{B}\, [\, \mathtt{A} \in \mathtt{M_A} \ \wedge \ \mathtt{B} \in \mathtt{M_B} \ \wedge \ \mathtt{dep(A,B)}\,]$$

In consequence, the semantic to *can only* and *only can* constraints can be defined based on *cannot* constraint function:

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

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

Finally, the semantic assigned to $\mathtt{M_A}$ *must*-dep $\mathtt{M_B}$ constraint is defined as:

$$\exists \mathtt{A}\, \nexists \mathtt{B}\, [\, \mathtt{A} \in \mathtt{M_A} \ \wedge \ \mathtt{B} \in \mathtt{M_B} \ \wedge \ \mathtt{dep(A,B)}\,]$$

Therefore, a violation occurs when an $A \in M_A$ does not establish a *dep* dependency with a $B \in M_B$.

3 The Architectural Conformance Solution

Although architectural conformance processes identify architectural violations, two problems still remain: "*architectural tools are underused*" and "*detected violations are rarely corrected*" [Alwis and Sillito, 2009, Terra et al., 2015]. On the one hand, the introduction of architectural conformance tools later in the project can reveal a high number of violations, which can discourage the team to fix them. On the other hand, if the constraints are introduced in the beginning of the project and do not evolve with the architecture, their usage would not be valuable for the team after some time.

To enable an efficient usage of architectural conformance tools as instruments to define, evolve, and guarantee the correct implementation of the defined architecture, this article proposes a solution to introduce an architectural conformance process in a development environment. More important, it also matches agile environments, where CI is a common practice. Since our solution enables an incremental definition of architectural constraints, it is compatible with patterns identified for handling architecture in agile teams [Guerra et al., 2015, Wirfs-Brock et al., 2015]. The architectural conformance might be also included as part of a continuous inspection process [Merson et al., 2014].

3.1 Introduction and Evolution of Constraints

An architectural conformance solution, as the one we are proposing, requires the definition of architectural constraints. A high number of constraints implies in the conformance process uncovering more violations. Nevertheless, we claim

the introduction of constraints should be done incrementally. For example, if the first version of the constraints contains a detailed set of constraints, the tool may detect a high number of violations and it would be hard for the team to prioritize and plan their correction in the next iterations. Also, it needs to be clear for the team the reason and importance of each constraint, otherwise developers will not prioritize the correction of detected violations.

Therefore, we propose a process to evolve the architectural constraints, as depicted in Figure 2. kin

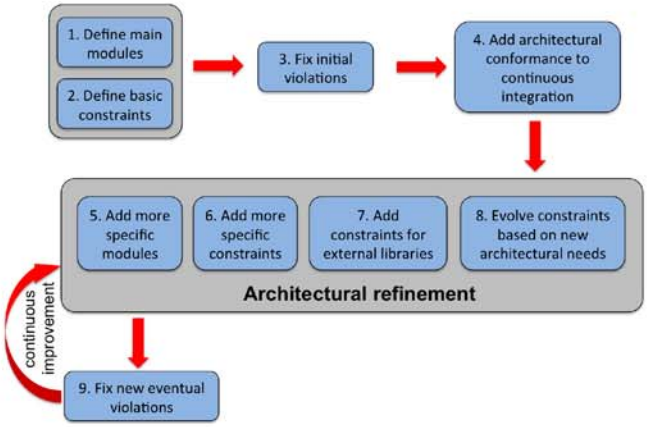


Figure 2: Process for Evolving Architectural Constraints Definition

Activities #1 and #2: The first version of the architectural constraints should be simple and define the modules that correspond to the most evident architectural components. In a layered architecture, the high-level layers—such as persistence, presentation, and business rules—are great candidates for these initial modules. As another example, the components of a framework project that represent the core hot spots and frozen spots might be used, e.g., the reference architecture for metadata-based frameworks suggests that metadata reading should be separated from metadata processing [Guerra et al., 2013]. In this context, since metadata reading might me exchanged to use other approaches, it represents the hot spot, and the processing represents the frozen spot. This explicit relationship between these classes is a good starting point for the architectural conformance constraints. As the last example, a module for the main external dependencies—such as the database access and the presentation framework—can also be defined, since it is also clear for the team that its use should be restricted to some classes. Since this initial architectural view is clear for the developers, it is easier for them to understand the defined constraints.

Activities #3 and #4: Finding some violations from these basic constraints in the current source code gives the idea of how the tool works and how it can be used to help them to keep the code aligned to the architecture. After fixing these initial violations, the initial set of constraints can be configured to be executed as part of the CI process. It ensures that code violating architectural constraints cannot be committed by a developer without notice (this environment is described in the next subsection).

Activities #5 to #8: After defining the initial modules and integrating the verification of constraints in CI, every integration of code is verified and every change in the module definition or in the constraints triggers an architecture conformance process. The team—guided by the architect or by the most experienced developers—should choose a part of the architecture that can benefit from more refined constraints. The following are examples of kinds of refinements and evolutions that can be targeted in one iteration:

- New modules might be introduced with its respective constraints. A new module might be the result of classes from a new feature that is being introduced in the architecture in this iteration or can be classes that were not included in the previous versions. As an example, a module "util" might not be included in the previous versions and might be included now. As another example, architecture now includes a class to handle asynchronous requests, then constraints should be also included (Activities #5 and #6).
- New modules that represent external libraries or frameworks might be introduced with their constraints to reflect their usage inside the architecture. It usually occurs when new libraries are introduced into the project or have not been included in the previous versions. For instance, a module for the Java Servlet API might be introduced with constraints that allow its usage (it is possible to specify the kind of dependency, such as access, declaration, inheritance, etc.) only on controller classes (Activity #7).
- Existing constraints might be refined to reflect more precisely the kind of dependence that is allowed. For instance, consider that a constraint prescribes that domain classes can depend on Java Persistence API (JPA). A more fine-grained constraint might refine this constraint defining that only domain classes can *use annotations* from JPA (Activity #8).
- Existing modules might be separated in more granular ones to enable the definition of more specific constraints. As an example, a module for controllers might be split into modules that contain web controllers and web services endpoints. Based on these new modules, more specific constraints might be defined. (Activity #8).

Activity #9: The introduction of new constraints might reveal violations. When it occurs, the team has three options: (a) rethink the constraint; (b) refactor the code; or (c) consider the violation a technical debt. Sometimes the violation reveals that there might be some exceptions to the constraint, and the team should rethink if that constraint is valid (case *a*). Another possibility is to refactor the code to make it compatible to the defined architectural constraints (case *b*). Last, the refactoring might be time-consuming, and the team can assign that violation as a technical debt (case *c*). In this case, the tool generates warnings for a constraint violation instead of blocking the commit. Independently the option the team chooses, they can return to the architectural refinement steps as long as they need.

The criterion used by the team to define the order in which new constraints are added is not highly relevant. We claim it is important to review and refine the architectural constraints periodically. The feedback received from constraints verification at each commit, combined with the team practice to review them in each iteration, is the key to have success in the adoption of this process.

Besides, there is no fixed constraint about when new constraints should be added. As an advice, when a new kind of component is introduced in the architecture, the addition of its architectural constraints can define more precisely its role and avoid its misuse. If a refactoring demands the decoupling between two kinds of components, the addition of a constraint that reflects the desired decoupling might help to detect targets for refactoring. Usually, the team should discuss what adds more benefit to the team. The architectural constraints can even not be related to the features implemented in the iteration. For instance, the iteration can focus on the addition of new reports and the new constraints are related to the introduction of security constraints.

This process might be introduced in the beginning of a project or in an ongoing one. In case of its usage in a project that does not have its architectural design documented, the introduction of the constraints can also work as a formalization and standardization of the architecture. It is recommended, however, that architects introduce the constraints at a pace that they can fix their uncovered violations. Thus, in an ongoing project with a large code base, we recommend to introduce constraints slower than in a project that is using them since the beginning.

3.2 Development Environment

This section describes roles and responsibilities of actors involved in the development environment, and the proper application of the tools used in the architectural conformance process, as illustrated in Figure 3.

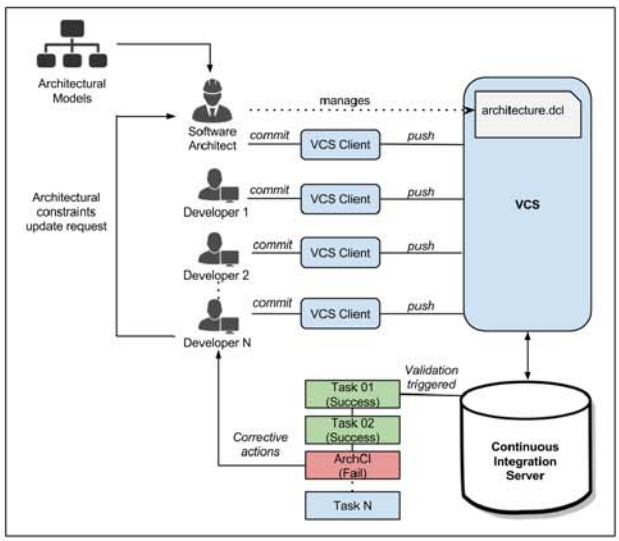


Figure 3: Development Environment

Based on the project's architectural models, software architects are responsible to specify the system's architectural constraints. They are also responsible for the evolution of these constraints, managing and updating them when needed.

The `.dcl` file (with the defined constraints) is stored in the same remote VCS repository where the project source code is. This allows an architectural conformance tool—such as ArchCI—to validate the system architecture every time a developer performs an integration (*push*) of the *committed* code in his/her VCS client to the remote VCS repository. Usually, the architectural conformance tool verifies the added and modified classes of that integration. It only verifies the entire project again when the architect changes the `.dcl` file.

The CI server triggers the architectural conformance process in every code integration.[3] When violations are not detected, the code is successfully integrated. However, when violations are detected, the developer cannot proceed with the integration, except when the violations are from constraints assigned as technical debt. In this case, an alert is sent to the developer responsible for the violation asking him/her to take corrective actions, as well as to the project manager or architect, informing him/her about such event. The developer can fix the violation in following integrations or may notify the software architect of the need of an architectural evolution.

[3] When other commit-blocking mechanisms are applied in development teams, the software engineer is responsible to define the dependence of the performed tasks on the CI server. Since tasks are usually independent, when a task in a CI server fails, it does not impact other running tasks. Even though, if there is any other task that modifies the committed code, a more conservative approach would recommend to setup the architectural conformance process after those tasks.

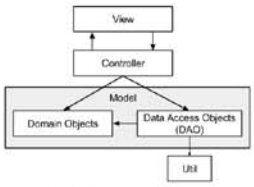
In a nutshell, we argue that our proposed environment maintains the source code always in convergence with the planned architecture (which contributes directly to the software quality) by the following reasons:

- The architectural conformance process is performed on every code integration with no need of installations on developers' machines, only on the CI server. This addresses the "*architectural tools are underused*" problem.
- It allows the integration only when no violations were detected, otherwise it will constantly alert developers to fix such violations. This addresses the "*detected violations are rarely corrected*" problem.

3.3 Illustrative Example

This section provides an example of the proposed process (Figure 2) using `myAppointments` [Passos et al., 2010], a simple personal information management system implemented to illustrate architectural conformance techniques. The system follows the well-known *Model-View-Controller* (MVC) architectural pattern [Fowler, 2002] and has primary functions such as create, search, update, and delete contacts.

Initial Release (r0): In this initial release, we develop `myAppointments` using textual user interface (UI) and no database persistence. Following the proposed process, we define main modules (Activity #1) and basic constraints (Activity #2) as illustrated in Figure 4.



(a) Architecture

```
1  %Modules
2  module View:        ma.view.*
3  module Model:       ma.model.**
4  module Domain:      ma.model.domain.*
5  module Util:        ma.util.*
6  module DAO:         "ma.model.[a-zA-Z/.]*DAO"
7
8  %Constraints
9  View cannot-depend Model              --block
10 Domain can-depend-only $java          --block
11 Util cannot-depend $system            --debt
12 DAO can-depend-only Domain, Util      --block
```

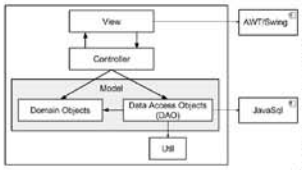
(b) DCL Architectural Constraints

Figure 4: myAppointments r0 Architecture

These basic constraints avoid well-known conceptual problems. When the *View* layer accesses the *Model* components directly (line 9, AC1), this bypassing

of MVC layers impacts on the system maintainability. The system reusability is jeopardized when *Domain Objects* depend on classes outside the Java API (line 10, AC2) or when the *Util* package depends on project classes (line 11, AC3). Also, the system extensibility is reduced when DAO classes depend on other classes besides *Domain Objects* and package *Util* (line 12, AC4). When integrating the code, ArchCI detects 38 violations. Therefore, we fix initial violations (Activity #3) except for 35 violations of the constraint AC3 we prefer to assign as technical debt (i.e., we postponed the fixing task). Last, we add architectural conformance to CI (Activity #4).

First Release (r1): In this first release, we incorporate a graphical UI and database persistence to `myAppointments`. Following the proposed process, we refine the `myAppointments` architecture (Activities #5 to #8), as seen in Figure 5.



(a) Architecture

```
1  %Modules
2  module Controller: ma.controller.*
3  module View:       ma.view.*
4  module Model:      ma.model.**
5  module Domain:     ma.model.domain.*
6  module Util:       ma.util.*
7  module DAO:        "ma.model.[a-zA-Z/.]*DAO"
8  module SQL:        java.sql.**
9  module UI:         java.awt.**, javax.swing.**
10
11 %Constraints
12 View cannot-depend Model                    --block
13 Domain can-depend-only $java                --block
14 Util cannot-depend $system                  --block
15 DAO can-depend-only Domain, Util, SQL       --block
16 only DAO, DB can-depend SQL                 --block
17 only View can-depend UI                     --block
```

(b) DCL Architectural Constraints

Figure 5: myAppointments r1 Architecture

We add two more specific modules (Activity #5) to SQL and graphical UI (lines 8-9, respectively). Although we do not add specific constraints (Activity #6), we add two constraints for external libraries (Activity #7), which promote maintainability by restricting the modules that can depend on SQL and UI (AC5-6, lines 16-17, respectively). We also evolve constraints based on new architectural needs (Activity #8) by updating AC4 since DAO classes can also depend on database services (line 15) and by removing the technical debt of AC3 (line 14). Since we change the `.dcl` file, a new verification is performed in the entire project. Thus, we fix five new and 35 postponed AC3 violations from *r0*.

Final remarks: Table 1 summarizes the evolution of `myAppointments` w.r.t. the architectural perspective. Although we rely on a toy example, this section illustrates how the proposed process could be applied in real-world development scenarios.

Table 1: `myAppointments`' Architecture Evolution

Release	Description	Constraints Added	# Modules	# Constraints	# Violations
r0	initial release	AC1, AC2, AC3, AC4	5	4	38
r0'	refined release r0	-	-	-	35 (debt)
r1	UI and persistence	AC4 (modified), AC5, AC6	8	+2 (added) 1 (modified)	40
r1'	refined release r1	-	-	-	0

4 ArchCI

We could not find an architectural conformance tool that implements our solution in CI. We therefore developed ArchCI, a tool that supports our solution using DCL as the underlying architectural conformance technique and Jenkins as the CI server.

4.1 Features

This section describes the six main features of our supporting tool:

- *Architectural Checking*: The integrations are performed in a temporary branch. When ArchCI does not detect violations, the code is automatically merged to the project's main branch. However, when violations are detected, the integration is denied and ArchCI performs the action configured in the specific violation case.
- *Atomicity*: Only integrations that are in full convergence with the defined architectural constraints are automatically accepted and merged by the server.
- *Incremental Checking*: ArchCI scans only the classes that have had changes since the last integration, ensuring a better tool performance. The exception occurs when the `.dcl` is changed, which requires ArchCI to verify the entire project again.

– *Architecture Evolution*: The architectural checking considers the DCL specifications stored in the remote repository (`architecture.dcl`). However, in case of an integration having changes in the `.dcl` file—to include or modify constraints and modules—ArchCI verifies the entire project again considering the DCL specifications *to be* integrated into the repository. Therefore, ArchCI has consistency, i.e., it never misses any violation.

– *Local Usage*: Since ArchCI performs the architectural conformance checking only at the time of code integration, developers could adopt a local conformance tool to ensure fewer violations in attempts of code integration to the remote repository. We suggest, to a better integration with our proposed process, the `dclcheck` tool [Terra and Valente, 2009]—a plug-in for Eclipse IDE that can run local verifications based on `.dcl` file as well.

– *Technical Debt Support*: When the development team does not want to handle certain violations at the moment, they can declare them as technical debts by adding `−−debt` soon after the constraint definition. Doing that, ArchCI generates warnings instead of blocking the code integration.

4.2 Underlying CI Environment

Jenkins: The proposed solution requires an underlying CI server. For this reason, Jenkins server is used to program tasks that ensure the architectural conformance in the integrations performed by developers. Each task that is created in Jenkins refers to specific activities, such as architectural verification or even sending e-mails to specific developers. Through the integration of the proposed tool with Jenkins platform, it is possible to define tasks that trigger the architectural verification process, where each task corresponds to a particular software project. In the same way, it could be easily implemented a task to send daily e-mails notifying the developers responsible for the integrations that contained violations. These tasks can be triggered by events originating from facts, such as time or even external triggers, allowing the adaptation to different tools and platform operations.

Gerrit: In order to provide an environment for revision of code integrations, the Gerrit platform is used in this project.[4] For the effectiveness of the architectural validation and verification process, the platform has its own code repository and every integration (*push*) must take place at a *branch* (adopting the prefix `HEAD : refs/for/` followed by the target *branch* name), which, although it was not defined in the repository, is provided exclusively by Gerrit to integration revision. Each integration performed by the developer becomes available for future

[4] https://www.gerritcodereview.com

analysis after its rejection or approval of its unification (*merge*) to the respective target *branch*. In addition, Gerrit provides custom scripts, called *hooks*, which are triggered when certain actions take place on the server.

4.3 Design and User Interface

As illustrated in Figure 6, the implementation of ArchCI follows an architecture with five main modules. In a nutshell, a push request triggers the Jenkins action implemented in module *CI Bridge*. This task, implemented by method `perform`, invokes (1) method `getDependencies` from *Dependency Extractor* to extract dependencies from classes that are being pushed, (2) method `parseModulesAndConstraints` from *Constraint Definition Parser* to obtain the up-to-date architectural constraints, and (3) method `validate` from *Architectural Conformance Checker* to verify whether the dependencies from (1) respect the constraints from (2). Each module is detailed as follow.

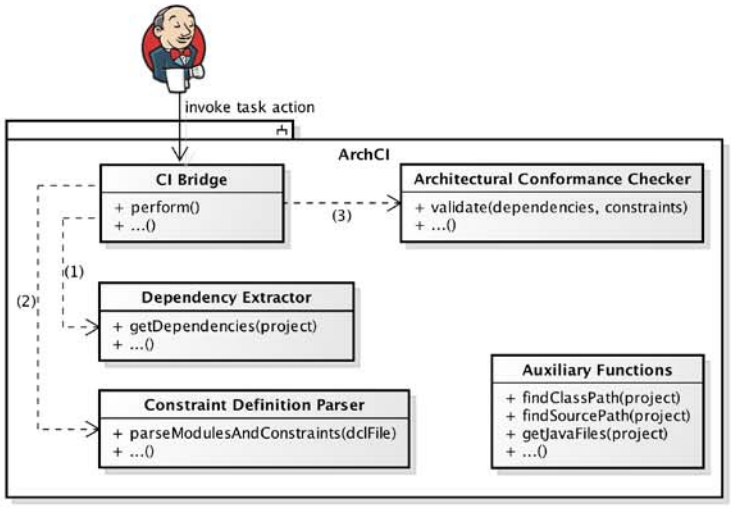


Figure 6: ArchCI architecture

– *Dependency Extractor*: Module responsible for obtaining the project dependencies, as well as manipulating them. It provides functions that analyze each element of the classes to be validated, analyzing the dependency type at which the particular element refers to. For its implementation, we adapted `dclcheck` [Terra and Valente, 2009], an Eclipse plug-in. Thus, we had to remove all dependencies and pieces of code that were entirely Eclipse IDE exclusive, remaining only the dependencies to the AST (*Abstract Syntax Tree*)

libraries provided by the Eclipse JDT (*Java Development Tools*). All dependencies to Java Model—a set of classes that represent and model a project internally in the Eclipse IDE—had to be completely discarded and, for this reason, almost every piece of code associated with the manipulation of the classes to be validated had to be rewritten. Each class to be validated had to be loaded externally, turned into a compilation unit and, subsequently, its environment had to be defined stating the project path and structure, as well as the necessary libraries to compile the given project. At last, methods and functions from the AST class were used, so that each element could be understood and then, the project dependencies could be determined.

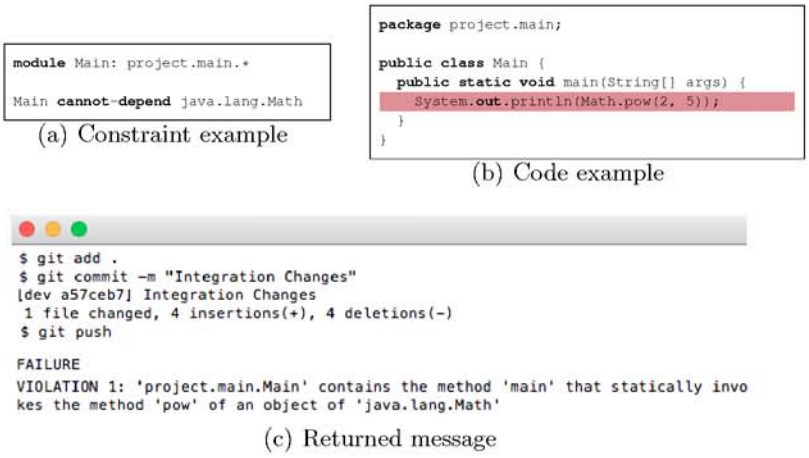
- *Constraint Definition Parser*: Module responsible for *parsing* the specified project architecture modules and constraints stored in file `architecture.dcl`. When the DCL file with the constraints to be validated is loaded, this module executes the function to interpret the name of each specified module along with the defined constraints and, finally, stores them separately in sets of modules and constraints.

- *Architectural Conformance Checker*: Module with functions to ensure the project architectural conformance through verification and validation of architectural deviations based on the defined architectural constraints. Each dependency extracted by module *Dependency Extractor* is checked against the constraints obtained by module *Constraint Definitions Parser* in order to find architectural violations.

- *Auxiliary Functions*: Module responsible for providing functions to general tasks, e.g., tracking the libraries and file paths needed to resolve the dependencies.

- *CI Bridge*: Module containing features related to CI practices and functions required to integrate the code to Jenkins server, which includes functions for customizing the *build*, getting the *workspace* with the code to be integrated, identifying classes to be validated, etc. Since ArchCI is structured as a Maven project, it could operate as a Jenkins *plug-in*. Subsequently, we included dependencies to classes that enable the manipulation of elements and components related to Jenkins tasks. At last, we created classes containing the *build* description and containing methods to get information provided by the server job.

During the CI process, when ArchCI detects violations, it returns error messages. An example of a message is shown in Figure 7(c), based on the example of dependency constraint from Figure 7(a) and the code from Figure 7(b).



(a) Constraint example

(b) Code example

(c) Returned message

Figure 7: ArchCI reporting a violation

5 Case Study

This section reports a case study performed on a real-world project named `LEONA` built by the National Institute for Space Research in Brazil, which has almost 14 KLOC in its last release and involves three developers. In this case study, we applied the proposed solution by introducing the architectural constraints iteratively. Particularly, we rely on ArchCI in the development environment to block the commits that contain violations to the prescribed DCL constraints.

This section is organized as follows. Section 5.1 defines the research questions. Section 5.2 describes our study design. Section 5.3 presents the project and context in which the case study was conducted. Section 5.4 describes the evolution of the architectural constraints as the software evolves. Section 5.5 reports and analyzes the violations detected in each release. Section 5.6 conducts a refactoring task on the defined constraints. Section 5.7 discusses how the constraints evolved and the benefits they brought into the project, and misuses of the DCL made by the developers. Finally, Section 5.8 lists threats to validity.

5.1 Research Questions

This case study aims to investigate how the proposed solution performs in a real-world project. Therefore, we defined the following three research questions:

- **RQ1**: Is the proposed solution suitable to be applied in a real-world project?
- **RQ2**: Is ArchCI suitable to support the proposed process?
- **RQ3**: Is our proposed solution effective to detect architectural violations?

5.2 Study Design

This case study consists of implementing the proposed process with the support of the tool ArchCI in a real-world project.

System choice: We look for a software project that satisfies the following four requirements: (a) a project that follows a development process with small iterations to enable the introduction of new constraints on each one; (b) a project whose development team provides free access to the source code, architectural specification, and detected violations; (c) a project whose one senior developer is willing to introduce our proposed solution; and (d) a Java project in a Git repository to enable ArchCI integration.

Data collect: For each release, we need the development team to provide us with: (a) the complete DCL specification (modules and constraints); (b) the number of violations found for each constraint; and (c) the source code. Neither formal interview nor questionnaire was applied to the participant developers. Instead, we opted to monitor the process being applied, interacting with the developer when necessary. We will take notes of those informal conversations to discuss them (when relevant).

Qualitative Analysis: The analysis performed to answer the research questions considers how the constraints evolved through the iterations and which violations were found during this process. Despite the retrieved data is quantitative, our analysis will combine them with qualitative information, such as contextual information about the project features, the motivation of the introduced constraints, the desired architecture, and the observations performed during the study.

Software Visualization: To support the data analysis, we rely on a Dependency Structure Matrix (DSM) [Sullivan et al., 2001, Sangal et al., 2005]. A DSM is a weighted square matrix whose rows and columns denote software elements, such as classes, packages, or modules. The value represented in the cell $[e_1, e_2]$ is the number of dependencies from e_1 to e_2. Basically, this representation is used to better visualize the location of the violations detected in each release of the system.

Refactoring: Concluding the case study, we will review and refactor the final DCL specification written by the development team. Our goal is to remove unnecessary constraints, merge related constraints, make constraints stricter, and replace constraints with more suitable dependency types in order to motivate the development team to keep using our proposed solution.

5.3 Subject System

LEONA is the Transient Luminous Event and Thunderstorm High Energy Emission Collaborative Network in Latin America, which researches Transient Luminous Events (TLEs). As part of this project, it was built a software system to monitor remote stations located throughout Brazil to perform observations that obtain images for analysis and studies of TLEs.

As illustrated in Figure 8, LEONA is based on the MVC architectural pattern [Reenskaug, 2003], but it also has additional layers on its *Model* component. *Model* has classes that represent the *Domain Objects*, the *Services* with business rules, and classes that implement the *DAO* pattern for persistence services. The system uses external libraries in the implementation of such components, such as Esfinge QueryBuilder [Guerra, 2014] for persistence layer, Quartz [Cavaness, 2006] for task scheduling, and VRaptor [Cavalcanti, 2013] for web MVC implementation.

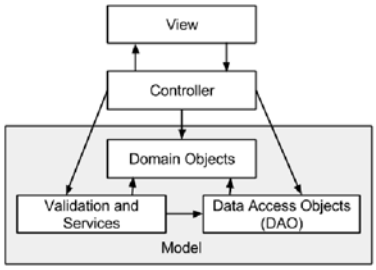


Figure 8: LEONA Architectural Main Components

The case study was initiated in a moment that a refactoring was being performed in the system. Although the system was being built using a CI process, the team decided to create a new code base and migrate module-by-module, refactoring the code or adding features in the process. The first release considered in the case study was the first portion of code migrated to the new code base. In this release, they introduced the first set of constraints. In each following release, new features were migrated and new constraints were introduced. The features of each release are as following:

- (r0) Register User, Hashing Password, and User Login.
- (r1) Register Station, Search Station by ID, Save Station Image, Search Station Image, Verify Station Status, Edit Station, and List Station Available Services.
- (r2) Register Observation, Schedule Observation, List Observations, Search Observation, and Save Observation Log.

– (r3) Communicate with Camera PanTilt, Turn On Camera, and Turn Off Camera.
– (r4) Move Camera PanTilt Azimuth, Move Camera PanTilt Elevation, Reset PanTilt, Start Remote Image Capture, Stop Remote Image Capture, and Search Images on Station.
– (r5) Save images on Database and Show images.

Feature "Search Images on Station" in r4 and all features in r5 were created in this new code base. As a convention, this article uses a prime symbol with the release identification to refer to a release where the detected architectural violations were corrected. As an example, r3' is a release where the architectural violations found in r3 were corrected.

The project team was composed of three developers, but only one of them was responsible to add the constraints and correct the violations. The project team did not have a member with an explicit role of "architect", and the most important decisions were discussed with the team with the support of an external collaborator. Therefore, the definition of constraints, their respective implementation in DCL, and the implementation of the proposed process were implemented by an experienced developer. He received some guidance in the creation of the first release of DCL, how he should increment the DCL following the process, and in the setup of ArchCI in the development environment. After that, we observed how the constraints were being implemented and decided not to interfere. The developer had some guidance again only after finishing release r5, the last one considered in the case study.

5.4 Software Architecture Evolution

This section presents how architectural constraints evolved during the case study, where the complete set of constraints can be found in Appendix A. Table 2 presents the system releases and their architectural constraints. Column "Current" presents the constraints present in that release, i.e., the constraints of the last release plus the added ones minus the removed ones; column "Added" presents the new constraints that were introduced in the same release; and column "Removed" presents the constraints that were removed in the same release. From the releases where the constraints violations were corrected, only r2' is present on this table because it is the only one of these releases that suffered a change on the constraints.

The constraint evolution through the releases (i.e., the main changes in a release) is summarized as follows:

– (r0) The first set of constraints focused on the relationship between the core component types of the system architecture. It also contained constraints

Table 2: Constraint Evolution

Release	Current	Constraints Added	Removed
r0	AC1-AC8	-	-
r1	AC1-AC15	AC9-AC15	-
r2	AC1-AC19	AC16-AC19	-
r2'	AC1-AC18	-	AC19
r3	AC1-AC18, AC20-AC24	AC20-AC24	-
r4	AC1-AC18, AC20-AC29	AC25-AC29	-
r5	AC1-AC3,AC7-AC10, AC12-AC18,AC20-AC44	AC30-AC44	AC4-AC6,AC11

about which component could access the more evident system libraries: VRaptor, Esfinge, and Swing.

- (r1) This release focused on the use of Java libraries to handle files. A constraint on the use of database libraries by Model classes was also added.
- (r2) This release added constraints about the use of library classes and the class used to manage the user session. A constraint about the relationship between services and controllers was also included.
- (r2') By verifying the constraints of the previous release, a violation detected by AC19 revealed that this constraint was no longer suitable for the application architecture and hence it was removed.
- (r3) This release added constraints regarding classes that handles HTTP requests.
- (r4) This release added constraints regarding networking and application services, such as security and logging.
- (r5) In this release, some constraints were removed and replaced by more specific ones. Some large modules were split into smaller modules. There were also some constraints that were complemented by others.

As stated in the proposed process, the constraints introduced in each iteration could not be related to new features. This is usually true when the constraints are related to architectural layers or to the usage of libraries that are orthogonal to the system features. For instance, in r1 the features are related to managing stations in the system and the new constraints focused on libraries for handling Java files. The new constraints were related not only to classes that were added for new features but also for the existing ones.

Based on Table 2 and on the description of each increment, we can understand how the process was applied to the case study. The initial version of the DCL had a smaller number of modules and constraints, related to the main software layers and libraries in the architecture. The next versions refined the modules definition, making them more granular, and added constraints that refer to other services, such as handing files and web requests.

During the case study, constraints were added but also removed in releases r2 and r5. In r2, the removal was in a recently introduced constraint (AC19). When violations have been detected, the developer realized that it was not suitable for the system architecture. In r5, however, four constraints introduced in r0 and r1 were removed due to a refinement to make constraints stricter.

Moreover, Table 2 indicates most constraints search for divergences—which restrict the spectrum of dependencies established by a module—and only one constraint searches for absences—which verifies a mandatory relationship. It occurs because the same architectural layer can perform tasks of different natures, e.g., classes from module Service can access files, invoke remote operations, and query the database. Although we can use a constraint to forbid other modules to perform such operations (searching for divergences), we cannot consider mandatory the usage of these libraries (searching for absences).

5.5 Architectural Violations

This section analyses the architectural violations found during the proposed process in the case study. Table 3 presents each release with its respective date, number of modules, constraints, violations, violations per constraints, changes that occur in that release, and overall lines of code (LOC). The column "Change" describes the kind of modification performed in that release: (i) "DCL and Code Increment" means that new features were migrated to the new code base and new constraints were added in the DCL; (ii) "Violation Correction" means that the changes in that release only corrected violations detected in the previous release; and (iii) "DCL Correction"—a situation that occurred only once when a wrong constraint was removed (refer to the previous section). In this case study, 42 violations were corrected through the releases by using the proposed process. More important, the last release of the system (r5) has not presented any violation despite the refinement of several constraints.

Figure 9 presents a visualization of the latest system release (r5) in a customized Dependency Structure Matrix (DSM).[5] A number x in a blue matrix cell means that the module in the column depends x times of the module in the row. A number y in a green matrix cell means the opposite, i.e., that the module

[5] Since we are only interested in the dependencies established by the system classes, we intentionally omitted rows with no establishment of dependencies and columns related to external modules.

Table 3: Evolution of the architectural constraints through the releases

Date	Release	Modules	Constraints	Violations	Violations per Constraint	Change	LOC
14-Aug-15	r0	9	8	9	AC5(9)	DCL and Code Increment	8,490
14-Aug-15	r0'	9	8	0		Violation Correction	8,493
26-Aug-15	r1	17	15	5	AC8(3), AC11(2)	DCL and Code Increment	9,730
27-Aug-15	r1'	17	15	0		Violation Correction	9,725
09-Sep-15	r2	21	19	6	AC2(1), AC5(2), AC8(3)	DCL and Code Increment	10,908
09-Sep-15	r2'	21	18	0		DCL Correction	10,908
21-Sep-15	r3	26	23	13	AC18(13)	DCL and Code Increment	11,830
21-Sep-15	r3'	26	23	0		Violation Correction	11,838
07-Oct-15	r4	31	28	9	AC5(9)	DCL and Code Increment	12,642
07-Oct-15	r4'	31	28	0		Violation Correction	12,640
28-Oct-15	r5	43	42	0		DCL and Code Increment	13,732

in the row depends y times of the module in the column. More important, red cells report the number of dependencies that represent violations that occur. For instance, on the column "Service" and row "br.com.caelum.VRaptor", the value "r2(1)" means that, in release r2, there was one dependency from "Service" to "br.com.caelum.vraptor" that represents a violation.

By observing the DSM, we noticed that violations in the same constraint occurred recurrently on different releases. For instance, AC5 states that only "DAO" can depend on "Esfinge". However, module "Service" has recurrently established dependency with "org.esfinge.querybuilder.QueryBuilder". When new features were migrated and refactored in the new code base, developers eventually moved violations that were in the old code base. According to the responsible developer, other violations were also prevented because developers learned with previously reported violations.

The orange cells would not be allowed based on constraint AC19. However, this constraint was introduced on release r2 and removed on release r2'. This is an interesting event because it shows that a violation does not always mean that there is a problem in the code, but it can reveal that the constraint is unsuitable for the architecture.

5.6 Refactoring of Constraints

The fact that the constraints were able to successfully identify architectural violations does not mean that they were defined in the most suitable way. Table 4 presents the refactoring set of 14 constraints (RAC1–RAC14) that summarizes the 44 constraints defined in the latest system release (AC1–AC44).

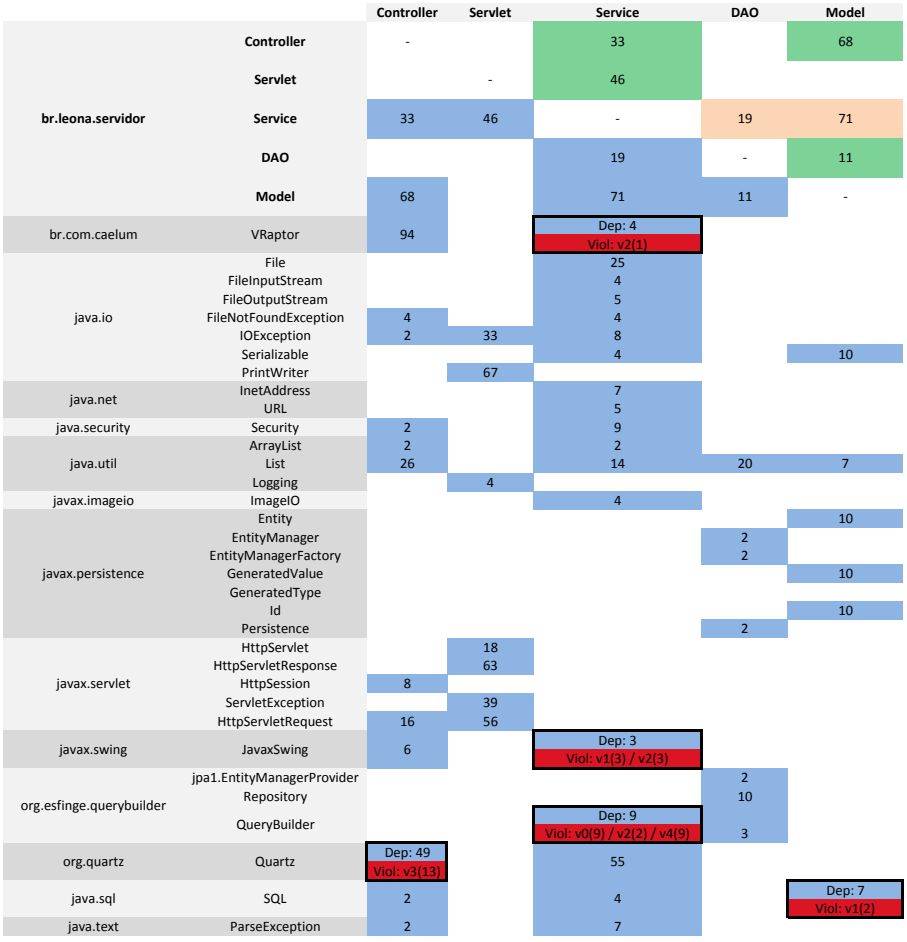


Figure 9: LEONA DSM with the violations detected through the releases

This refactoring is part of the last step of the case study and aims to highlight the good practices on constraints definitions. The refactoring on the constraints turned the existing constraints more suitable and no new violations were found during this process. We performed the following kind of changes:

1. *Merge several related constraints into a single one*: There were nine modules that only Service can depend on. Since there was one constraint for each module, we joined nine constraints in a single one (RAC5). Despite the DCL specification could be simpler by grouping modules with the same constraints and related to the same architectural feature, this does not make the constraints incorrect or less effective.

Table 4: Constraints After Refactoring

Ref	Constraints After Refactoring	From Constraints
RAC1	only Controller can-depend Controller,JavaxSwing,HttpSession, VRaptor	AC1,AC2*,AC8, AC17
RAC2	only Controller,Service can-throw FileNotFound,ParseException	AC9,AC16
RAC3	only Controller,Service,Servlet can-throw IOException	AC10
RAC4	only Controller,Servlet can-depend HttpServletReq	AC24
RAC5	only Service can-depend File,FileIS,InetAddress,ImageIO, Quartz,BufferedImage,FileOS,FTP,NetURL	AC12,AC13,AC14, AC15,AC18,AC25, AC29,AC31,AC32
RAC6	only Servlet can-depend ServletEx,HttpServlet,HttpServletRes	AC21,AC22,AC23
RAC7	only Service can-declare Security	AC28
RAC8	only DAO can-depend Persistence,EntityManager,EMF, EsfingeEMP,EsfingeRepo,EsfingeQB	AC30,AC37,AC38, AC39,AC40,AC43
RAC9	only Model can-useannotation Entity,PersistenceGT, PersistenceId,PersistenceGV	AC33,AC34,AC35, AC36
RAC10	Controller must-useannotation VRaptor	AC2*
RAC11	Service cannot-depend Controller, Servlet	AC3
RAC12	Model must-implement Serializable	AC7
RAC13	only Model can-implement Serializable	AC41
RAC14	Servlet,DAO cannot-depend SQL	AC42

* *Partially*

2. *Replace a constraint with a more suitable or a broader dependency type*: Another problem of the defined DCL constraints was the overuse of "can-declare" and "cannot-declare" types of constraint. These constraints verify a weak relationship between the classes, i.e., just declarations of the other, and it was not the most appropriate dependency type to be used in the DCL. As an example, RAC7 states that Model must implement Serializable, which corrects AC7 that incorrectly forbade *declare* dependencies. In other constraints, these types could be easily exchanged for "can-depend" or "cannot-depend", which refer to a broader dependency type. As another example, RAC6 states that only Servlet can depend on three modules, which corrects AC21–AC23 that incorrectly stated only *declare* dependencies. Similarly, RAC9 states that only Model can use annotation of four modules, which corrects the AC33–AC36 that also incorrectly stated only *declare* dependencies. As the last example, AC2 specifies that only Controller can declare VRaptor. RAC1 specifies it using a broader dependency type (depend rather than declare) and RAC10 complements it ensuring that Controller uses annotations (a more suitable dependency type) from VRaptor. Therefore, after the case study, the DCL specification was refactored to express the most appropriate dependency type and no additional violation was detected. Even though there potentially were other types of violations (creates, accesses, etc.) along with the forbidden declarations, when the developer was fixing the forbidden declarations, implicitly they also fixed the other types of violations as well.

3. *Make the constraints stricter*: There are constraints that could be more precise in the way they were defined. For instance, AC43 stated that Controller cannot declare Serializable. However, the developer noted that—besides changing the dependency type to *implement*—the constraint could restrict the implementation of Serializable from Model only (RAC13). Another issue occurred when developers were incorrectly using constraints on the form *can dependency-type only*. For instance, AC19, which was included in r2 and removed in r2'—stated that Service can depend only on Controller. The developer argued that such constraint resulted in an explosion in the number of violations. However, we mentioned to him that the right side of the constraint should contain $java, which refers to any class from the Java API, otherwise it raises countless errors.

4. *Removal of irrelevant constraints w.r.t. the architecture*: The developer has noted that some constraints are not particularly important from the architectural perspective—namely AC20, AC26, AC27, and AC44—and removed them. For instance, AC27 stated that only Service could declare ArrayList, which is not architecturally relevant. A similar scenario occurs with AC6 (removed in r5) when the developer noted that module View no longer existed.

5.7 Discussion

In this section, we directly answer the case study research questions.

RQ1 - Is the proposed solution suitable to be applied in a real-world project? Based on the proposed process, the constraints were increased and refined through the releases. As the proposed process uses an iterative constraint definition, the experience of defining constraints and verifying them in the current code base is relevant for further definitions. Thus, after each release, the developers are more mature about how to define the constraints and how they can add value to the team.

The introduction of the proposed process increased the awareness of the architectural constraints by the developers. The constant review of the set of constraints enforces developers to frequently rethink the architectural solutions and constraints. This simple awareness, according to a developer, helped to avoid the introduction of new violations, especially on new features. It could be observed in release r5, where all new code features were introduced without violating any constraint.

RQ2 - Is ArchCI suitable to support the proposed process? In the case study, the usage of ArchCI can be considered successful. The integration of the tool in the existing CI process was natural and the centralization did not demand

the installation on each developer machine. More important, it prevented the architectural conformance checking to be ignored in a commit by a developer.

The feature of blocking commits with violations was considered suitable to be used with the proposed process. Since the introduction of new constraints and new pieces of code occurred in small increments, it was feasible to correct the violations to enable the code commit. As reported in Table 3, the corrections were released soon after they were detected. According to the team, performing the correction did not impact significantly the project activities and hence there was no need to use the technical debt feature.

RQ3 - Is our proposed solution effective to detect architectural violations? As we can see in Table 3, ArchCI detected 42 architectural violations in five constraints through the releases. A new release correcting the violations was almost always released on the same day. The reasonable number of violations detected in each release make viable for it to be handled in a short term.

5.8 Threats to Validity

Next, we identify threats to validity in our case study.

Internal Validity: Several steps of this case study involved one specific developer of the team. Although one could question the representativeness of qualitative results (might be biased or based on a single opinion), the development team has no relationship with any of us and such developer guaranteed that his opinion reflects the team's one.

External Validity: First, despite the successful implementation of the process in this case study, we cannot claim that our approach will provide equivalent results in other systems (as usual in case studies in software engineering). Although the evaluation using other systems would be desirable, this kind of study is hard to apply in a large scale since it needs the process to be applied on a system with a non-trivial architecture for several releases (which is the reason we chose LEONA).

Second, our case study relies only on a Java Web application (one particular architectural style) and therefore we cannot claim equivalent results to other kinds of applications, especially based on innovative architectures. However, we argue that LEONA architecture follows a well-known reference architecture that uses several architectural patterns and common practices.

Conclusion Validity: The development team of our case study is composed of two full-time and one part-time developers. Although there is no evidence the process would not be suitable for a larger team, we acknowledge we cannot conclude it.

6 Related Work

We divided the related work in three groups: (i) *architecture conformance checking*, since our approach needs to identify architectural violations; (ii) *architecture repair*, since our approach can block integrations of code with violation; and (iii) *CI approaches*, since our approach monitors code integration.

Architecture Conformance Checking: Several techniques have been proposed to deal with the architecture erosion problem [Passos et al., 2010]. In a survey, Silva and Balasubramaniam indicated a combination of strategies that might help control architecture erosion [de Silva and Balasubramaniam, 2012]. They expressed an opinion that an *architecture conformance* approach to point out violations together with an *architecture repair* approach to assist developers in fixing them are likely to extend the lifetime of the software. Our proposed process somehow follows their idea and ensures the planned architecture by forcing developers to fix violations before integrating the code.

Reflexion Models (RM) compares the source code to planned architecture of a system [Murphy et al., 1995, Murphy et al., 2001]. As a result, the technique highlights the detected differences in terms of *divergences* and *absences* in what the authors called a *reflexion model*. It is a popular technique with several commercial tools [Knodel et al., 2006, Lindvall and Muthig, 2008, Knodel et al., 2008b] and studies extending it to support, for example, hierarchical structures [Koschke and Simon, 2003], behavioral design [Ackermann et al., 2009], and software variants [Koschke et al., 2009, Frenzel et al., 2007].

Dependency Structure Matrices (DSMs) provide a scalable view of the dependencies among classes of a system [Sullivan et al., 2001, Sangal et al., 2005]. A DSM is a weighted square matrix whose rows and columns denote classes from an object-oriented system. The LDM tool[6] 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.

ArchLint [Maffort et al., 2016, Maffort et al., 2013a, Maffort et al., 2013b] implements a lightweight approach for architecture conformance that does not require a manual specification of the architecture. Constraints are mined from version repositories using a combination of static and historical source code analysis. Goldstein and Segall [Goldstein and Segall, 2015] implemented a similar approach for the automatic detection of architectural violations based, in turn, on predefined and user-defined patterns, which does not require prior knowledge of the intended architecture. Their article somehow inspires ours since they proposed to leverage their solution as part of the nightly build process used by

[6] http://www.lattix.com

development teams to achieve continuous automatic validation of the software architecture.

Besides the aforementioned architectural conformance techniques, there are others based on source code query languages [de Moor et al., 2007], unit tests [Brunet et al., 2009, Brunet et al., 2011], and custom checks of existing static analysis tools [Merson, 2013, Merson et al., 2014]. Currently, our proposed approach relies on architectural violations detected by DCL [Terra and Valente, 2009], which was previously described in Section 2.3. However, we argue that it is straightforward to adapt our approach to the aforementioned conformance techniques and tools (e.g., RMs or DSMs). For instance, one could rely on ArchLint when architectural documentation (if available) is outdated or when dealing with walking architectures [Unphon and Dittrich, 2010].

Although aforementioned architectural conformance techniques aim to point out architectural violations, nothing guarantees developers fix them. Our proposed process, on the other hand, aims to avoid software architecture erosion by only integrating code that has been successfully passed through an architectural conformance checking. In this way, the code base is always in conformance with the planned architecture.

Architecture Repair: When an IT company relies on an architectural conformance approach, 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 product line in the domain of portable measurement devices [Knodel et al., 2008a]. As a result, they identified almost 5,000 architectural divergences in three products of this product line. In a previous work [Terra and Valente, 2009], we described our own experience in applying conformance techniques to a human-resource management system. In this process, we were able to detect more than 2,200 architectural violations. As the last example, Sarkar et al. reported their experience in modularizing a large banking application [Sarkar et al., 2009]. Reconstructing the original architecture of this system demanded 2,100 person-days just for coding and testing. These arguments favor approaches (like the one we are proposing) that do not allow the integration of architecturally questionable code.

In a certain point, architecture repair techniques could be used to modify the source code in order to reestablish the conformance with the planned architecture. In these cases, developers could apply refactorings [Fowler, 1999]—e.g., *Extract Method*, *Move Method*, and *Move Class*—or rely on more sophisticated solutions, such as ArchFix that formalizes a set

of architectural repair recommendations to fix violations raised by static architecture conformance checking approaches [Terra et al., 2015]. However, when architecture erosion is neglected over the years, the software architecture becomes a set of strongly-coupled and weakly-cohesive components [Borchers, 2011]. In these scenarios, architecture conformance and repair techniques present them mostly ineffective and a complete modularization can be the only solution [Hochstein and Lindvall, 2005, Rama and Patel, 2010, Anquetil and Lethbridge, 1999, Mitchell and Mancoridis, 2006, O'Keeffe and Cinnéide, 2006]. These conditions also favor approaches (like the one we are proposing) that favor incremental architectural evolution and avoid the accumulation of architectural erosion over the years.

Continuous Integration Approaches: We claim the originality and applicability of our proposed architectural conformance process. However, existing tools could be adapted to follow our process like ArchCI does. Deissenboeck et al. [Deissenboeck et al., 2008] list several tool support for continuous quality control—such as Sotograph and iPlasma—that fit in their *System analysis workbenches* category, as ArchCI does.

As an example, although a general-purpose static analysis tool, Sonargraph-Architect [Hello2morrow, 2017] provides an architecture Domain Specific Language (DSL) that can be used by developers to design their architectures. Next, a Jenkins plug-in [Kellner et al., 2017] allows developers to define if the build should be marked as "unstable" or "failed" when Sonargraph-Architect finds architectural violations. It is not exactly what our process contemplates, but it could be (with a reasonable effort) adapted to.

As another example, SonarQube is an open-source platform to track and manage the quality of a project [Campbell and Papapetrou, 2013]. It provides information about test coverage metrics, dependency matrix, compliance to good code practices, technical debt, etc. In addition to providing tools and metrics for quality analysis at different moments of development, SonarQube provides integration with IDEs and CI servers. In the software architecture perspective, by establishing architectural rules in the SonarQube, it becomes possible to obtain information about the conformance as part of the CI process. Based on this idea, Merson et al. [Merson et al., 2014] defined custom rules on different aspects of development as well as to set architectural constraints, and later exported them to use in quality analysis tools, code review, or even CI servers. The authors demonstrated how to use the API from Checkstyle—an open-source tool focused on static Java code analysis—to verify the conformance of the code with manually implemented rules. The rule set, which runs on Checkstyle, could be easily integrated with SonarQube.

Although these studies have features related to the evaluation of software quality during the CI process, none focuses directly on an instrument for ar-

chitectural conformance checking. As mentioned by Merson [Merson, 2013], it is possible to create custom rules for this purpose, but his approach demands an excessive work and effort, requiring the creation of a class for each rule. Under these circumstances, a simple rule definition, combined with verification and validation as part of the CI process, emphasizes the originality of the proposed solution on this article.

7 Conclusion

The software architecture erosion problem fractures significant characteristics of software systems, such as maintainability, reusability, scalability, portability, etc. Our proposed solution addresses this problem by using a stringent conformance process into CI, i.e., we allow code integrations when no violations are detected. Besides formalizing an architectural conformance process, we implemented tool support and conducted an evaluation of a real-world project.

The contribution and novelty of our architectural conformance process are sixfold: (i) it addresses the "*architectural tools are underused"* problem and promotes the use of conformance tools since it is integrated with CI servers; (ii) it addresses the "*detected violations are rarely corrected"* problem since it mostly allows code integrations when no violations are detected (except for item v); (iii) it contemplates both continuous and evolutive aspects of architectural constraints, besides an always up-to-date formal architectural definition; (iv) it is fully supported by the ArchCI tool; (v) it supports Industry reality allowing violations that do not need to be immediately corrected to be configured as technical debt; and (vi) it was successfully applied in a real-world project.

As future work, it is intended to: (i) apply the proposed solution in other real-world development scenarios to evaluate its applicability, expressiveness, and performance; (ii) evaluate the tool usability, e.g., best way to perform the checking, best way to present violations, as well as most important features for acceptance of the developers, regarding different approaches for reporting violations; and (iii) define severity levels for the architectural constraints, which would allow developers to set predetermined actions for each severity level, e.g., block the integration for violations that affect security and send an e-mail alert for violations that affect performance.

Acknowledgements

This work is supported by CNPq (grant 445562/2014-5) and FAPESP (grant 2014/16236-6). We acknowledge the valuable contribution of Nicolas Fontes, as the developer who applied the proposed process and collected the data in the case study.

References

[Ackermann et al., 2009] Ackermann, C., Lindvall, M., and Cleaveland, R. (2009). Towards behavioral reflexion models. In *20th International Symposium on Software Reliability Engineering (ISSRE)*, pages 175–184.

[Aldrich et al., 2002] Aldrich, J., Chambers, C., and Notkin, D. (2002). ArchJava: Connecting software architecture to implementation. In *24th International Conference on Software Engineering (ICSE)*, pages 187–197.

[Alwis and Sillito, 2009] Alwis, B. d. and Sillito, J. (2009). Why are software projects moving from centralized to decentralized version control systems? In *2nd Cooperative and Human Aspects on Software Engineering (CHASE)*, pages 36–39.

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

[Berg, 2012] Berg, A. (2012). *Jenkins Continuous Integration Cookbook*. Packt Publishing.

[Borchers, 2011] Borchers, J. (2011). Invited talk: Reengineering from a practitioner's view – a personal lesson's learned assessment. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 1–2.

[Bowyer and Hughes, 2006] Bowyer, J. and Hughes, J. (2006). Assessing undergraduate experience of continuous integration and test-driven development. In *28th International Conference on Software Engineering (ICSE)*, pages 691–694.

[Brunet et al., 2009] Brunet, J., Guerreiro, D., and Figueiredo, J. (2009). Design tests: An approach to programmatically check your code against design rules. In *31st International Conference on Software Engineering (ICSE), New Ideas and Emerging Results Track*, pages 255–258.

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

[Campbell and Papapetrou, 2013] Campbell, A. and Papapetrou, P. (2013). *SonarQube in Action*. Manning.

[Cavalcanti, 2013] Cavalcanti, L. (2013). *VRaptor - Desenvolvimento ágil para web com Java*. Casa do Código. (in Portuguese).

[Cavaness, 2006] Cavaness, C. (2006). *Quartz Job Scheduling Framework: Building Open Source Enterprise Applications*. Prentice Hall.

[de Moor et al., 2007] de Moor, O., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Sereni, D., and Tibble, J. (2007). Keynote address: .QL for source code analysis. In *7th International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 3–14.

[de Silva and Balasubramaniam, 2012] de Silva, L. and Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151.

[Deissenboeck et al., 2008] Deissenboeck, F., Juergens, E., Hummel, B., Wagner, S., y Parareda, B. M., and Pizka, M. (2008). Tool support for continuous quality control. *IEEE Software*, 25(5):60–67.

[Duvall et al., 2007] Duvall, P., Matyas, S., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education.

[Fowler, 1999] Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley, Boston.

[Fowler, 2002] Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston.

[Fowler and Foemmel, 2006] Fowler, M. and Foemmel, M. (2006). Continuous integration. Technical report, Thought-Works.

[Frenzel et al., 2007] Frenzel, P., Koschke, R., Breu, A., and Angstmann, K. (2007). Extending the reflexion method for consolidating software variants into product lines. In *14th Working Conference on Reverse Engineering (WCRE)*, pages 160–169.

[Garlan and Shaw, 1996] Garlan, D. and Shaw, M. (1996). Software architecture: Perspectives on an emerging discipline. Prentice Hall.

[Goldstein and Segall, 2015] Goldstein, M. and Segall, I. (2015). Automatic and continuous software architecture validation. In *37th International Conference on Software Engineering (ICSE)*, pages 59–68.

[Guerra, 2014] Guerra, E. (2014). Designing a framework with test-driven development: A journey. *IEEE Software*, 31(1):9–14.

[Guerra et al., 2013] Guerra, E., Alves, F., Kulesza, U., and Fernandes, C. (2013). A reference architecture for organizing the internal structure of metadata-based frameworks. *Journal of Systems and Software*, 86(5):1239–1256.

[Guerra et al., 2015] Guerra, E., Wirfs-Brock, R., and Yoder, J. (2015). Patterns for initial architectural design on agile projects. In *4th Asian Conference on Pattern Languages of Programs (AsianPLoP)*.

[Hello2morrow, 2017] Hello2morrow (2017). Sonargraph-architect. https://www.hello2morrow.com/products/sonargraph/architect9.

[Hinsen et al., 2009] Hinsen, K., Läufer, K., and Thiruvathukal, G. (2009). Essential tools: Version control systems. *Computing in Science & Engineering*, 11(6):84–91.

[Hochstein and Lindvall, 2005] Hochstein, L. and Lindvall, M. (2005). Combating architectural degeneration: a survey. *Information and Software Technology*, 47(10):643–656.

[Kellner et al., 2017] Kellner, I., Angee, E., and Hoyer, A. (2017). Sonargraph plugin. https://wiki.jenkins-ci.org/display/JENKINS/Sonargraph+Plugin.

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

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

[Knodel et al., 2008b] Knodel, J., Muthig, D., and Rost, D. (2008b). Constructive architecture compliance checking - an experiment on support by live feedback. In *24th International Conference on Software Maintenance (ICSM)*, pages 287–296.

[Koschke et al., 2009] Koschke, R., Frenzel, P., Breu, A., and Angstmann, K. (2009). Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17:331–366.

[Koschke and Simon, 2003] Koschke, R. and Simon, D. (2003). Hierarchical reflexion models. In *10th Working Conference on Reverse Engineering (WCRE)*, pages 36–47.

[Lindvall and Muthig, 2008] Lindvall, M. and Muthig, D. (2008). Bridging the software architecture gap. *Computer*, 41(6):98–101.

[Maffort et al., 2013a] Maffort, C., Valente, M. T., Anquetil, N., Hora, A., and Bigonha, M. (2013a). Heuristics for discovering architectural violations. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 1–10.

[Maffort et al., 2013b] Maffort, C., Valente, M. T., Bigonha, M., Hora, A., Anquetil, N., and Menezes, J. (2013b). Mining architectural patterns using association rules. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 375–380.

[Maffort et al., 2016] Maffort, C., Valente, M. T., Terra, R., Bigonha, M., Anquetil, N., and Hora, A. (2016). Mining architectural violations from version history. *Empirical Software Engineering*, 21(3):854–895.

[Merson, 2013] Merson, P. (2013). Ultimate architecture enforcement: custom checks enforced at code-commit time. In *2013 Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, pages 153–160.

[Merson et al., 2014] Merson, P., Yoder, J., Guerra, E., and Aguiar, A. (2014). Continuous inspection. In *2nd Asian Conference on Pattern Languages of Programs (AsianPLoP)*, pages 1–13.

[Mitchell and Mancoridis, 2006] Mitchell, B. S. and Mancoridis, S. (2006). On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208.

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

[Murphy et al., 2001] Murphy, G., Notkin, D., and Sullivan, K. (2001). Software reflexion models. *IEEE Transactions on Software Engineering*, 27(4):364–380.

[Nierstrasz and Lungu, 2012] Nierstrasz, O. and Lungu, M. (2012). Agile software assessment. In *20th International Conference on Program Comprehension (ICPC)*, pages 3–10.

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

[O'Sullivan, 2009] O'Sullivan, B. (2009). Making sense of revision-control systems. *Queue*, 7(7):30–40.

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

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52.

[Pinto and Terra, 2015] Pinto, A. F. and Terra, R. (2015). Processo de conformidade arquitetural em integração contínua. In *2nd Latin-American School on Software Engineering (ELA-ES)*, pages 42–53. (in Portuguese).

[Rama and Patel, 2010] Rama, G. M. and Patel, N. (2010). Software modularization operators. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10.

[Reenskaug, 2003] Reenskaug, T. (2003). The model-view-controller (MVC) its past and present.

[RhodeCode, 2017] RhodeCode (2017). Version control systems popularity in 2016. https://rhodecode.com/insights/version-control-systems-2016.

[Sangal et al., 2005] Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex software architecture. In *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176.

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

[Spinellis, 2005a] Spinellis, D. (2005a). Version control, part 1. *IEEE Software*, 22(5):107–107.

[Spinellis, 2005b] Spinellis, D. (2005b). Version control, part 2. *IEEE Software*, 22(6):c3–c3.

[Sullivan et al., 2001] Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. (2001). The structure and value of modularity in software design. In *9th International Symposium on Foundations of Software Engineering (FSE)*, pages 99–108.

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

[Terra et al., 2015] Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2015). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45(3):315–342.

[Unphon and Dittrich, 2010] Unphon, H. and Dittrich, Y. (2010). Software architecture awareness in long-term software product evolution. *Journal of Systems and Software*, 83(11):2211–2226.

[Verbaere et al., 2008] Verbaere, M., Godfrey, M. W., and Gîrba, T. (2008). Query technologies and applications for program comprehension. In *16th International Conference on Program Comprehension (ICPC)*, pages 285–288.

[White, 2014] White, O. (2014). Java tools and technologies landscape for 2014. https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/12/.

[Wirfs-Brock et al., 2015] Wirfs-Brock, R., Guerra, E., and Yoder, J. (2015). Patterns to develop and evolve architecture during a agile software project. In *22nd Conference on Pattern Languages of Programs (PLoP)*.

A Complete Description of Architectural Constraints

Table 5 reports a complete description of the architectural constraints of Leona. It also reports the release we added or removed (optional) the constraint.

Table 5: Constraints Evolution Rastreability

Ref	Constraint	Release Added	Release Removed
AC1	**only** Controller **can-depend** Controller	r0	-
AC2	**only** Controller **can-declare** VRaptor	r0	-
AC3	Service **cannot-depend** Controller, Servlet	r0	-
AC4	**only** Model **can-declare** Persistence	r0	r5
AC5	**only** DAO **can-depend** Esfinge	r0	r5
AC6	View **cannot-access** Model	r0	r5
AC7	Model **must-declare** Serializable	r0	-
AC8	**only** Controller **can-declare** JavaxSwing	r0	-
AC9	DAO **cannot-declare** FileNotFound	r1	-
AC10	Model **cannot-declare** IOException	r1	-
AC11	Model **cannot-declare** SQL	r1	r2
AC12	**only** Service **can-declare** File	r1	-
AC13	**only** Service **can-declare** FileIS	r1	-
AC14	**only** Service **can-declare** InetAddress	r1	-
AC15	**only** Service **can-declare** ImageIO	r1	-
AC16	Model **cannot-declare** ParseException	r2	-
AC17	**only** Controller **can-declare** HttpSession	r2	-
AC18	**only** Service **can-declare** Quartz	r2	-
AC19	Service **can-depend-only** Controller	r2	r2'
AC20	**only** Servlet **can-declare** PrintWriter	r3	-
AC21	**only** Servlet **can-declare** ServletEx	r3	-
AC22	**only** Servlet **can-declare** HttpServlet	r3	-
AC23	**only** Servlet **can-declare** HttpServletRes	r3	-
AC24	Service **cannot-declare** HttpServletReq	r3	-
AC25	**only** Service **can-declare** FTP	r4	-
AC26	Controller **cannot-declare** Logging	r4	-
AC27	**only** Service **can-declare** ArrayList	r4	-
AC28	**only** Service **can-declare** Security	r4	-
AC29	**only** Service **can-declare** NetURL	r4	-
AC30	**only** DAO **can-declare** Persistence	r5	-
AC31	**only** Service **can-declare** BufferedImage	r5	-
AC32	**only** Service **can-declare** FileOS	r5	-
AC33	**only** Model **can-declare** Entity	r5	-
AC34	**only** Model **can-declare** PersistenceGV	r5	-
AC35	**only** Model **can-declare** PersistenceGT	r5	-
AC36	**only** Model **can-declare** PersistenceId	r5	-
AC37	**only** DAO **can-declare** EntityManager	r5	-
AC38	**only** DAO **can-declare** EMF	r5	-
AC39	**only** DAO **can-declare** EsfingeEMP	r5	-
AC40	**only** DAO **can-declare** EsfingeRepo	r5	-
AC41	Controller **cannot-declare** Serializable	r5	-
AC42	Servlet **cannot-declare** SQL	r5	-
AC43	Model **cannot-declare** EsfingeQB	r5	-
AC44	Servlet **cannot-declare** List	r5	-

B Final DCL Specification

Listing 1 reports the complete specification of modules and constraints for the latest release of Leona.

```
%Modules
module Controller:     br.leona.servidor.controller.*
module Model:          br.leona.server.model.*
module Service:        br.leona.servidor.service.*
module DAO:            br.leona.server.dao.*
module Servlet:        br.leona.servidor.servlet.*
module VRaptor:        br.com.caelum.vraptor.*
module BufferedImage:  java.awt.image.BufferedImage.*
module File:           java.io.File.*
module FileIS:         java.io.FileInputStream.*
module FileOS:         java.io.FileOutputStream.*
module InetAddress:    java.net.InetAddress.*
module NetURL:         java.net.URL.*
module Security:       java.security.*
module ImageIO:        javax.imageio.ImageIO.*
module Entity:         javax.persistence.Entity.*
module EntityManager:  javax.persistence.EntityManager.*
module EMF:            javax.persistence.EntityManagerFactory.*
module PersistenceGV:  javax.persistence.GeneratedValue.*
module PersistenceGT:  javax.persistence.GenerationType.*
module PersistenceId:  javax.persistence.Id.*
module Persistence:    javax.persistence.Persistence.*
module HttpServlet:    javax.servlet.http.HttpServlet.*
module HttpServletRes: javax.servlet.http.HttpServletResponse.*
module HttpSession:    javax.servlet.http.HttpSession.*
module ServletEx:      javax.servlet.ServletException.*
module JavaxSwing:     javax.swing.*
module FTP:            org.apache.commons.net.ftp.*
module EsfingeEMP:     org.esfinge.querybuilder.jpa1.EntityManagerProvider.*
module EsfingeRepo:    org.esfinge.querybuilder.Repository.*
module Quartz:         org.quartz.*
module FileNotFound:   java.io.FileNotFoundException
module IOException:    java.io.IOException
module Serializable:   java.io.Serializable
module SQL:            java.sql.*
module ParseException: java.text.ParseException
module HttpServletReq: javax.servlet.http.HttpServletRequest
module EsfingeQB:      org.esfinge.querybuilder.QueryBuilder

%Constraints After Refactoring:
%RAC1 (AC1,AC2*,AC8,AC17)
only Controller can-depend Controller,JavaxSwing,HttpSession,VRaptor
%RAC2 (AC9,AC16)
only Controller,Service can-throw FileNotFound,ParseException
%RAC3 (AC10)
only Controller,Service,Servlet can-throw IOException
%RAC4 (AC24)
only Controller,Servlet can-depend HttpServletReq
%RAC5 (AC12,AC13,AC14,AC15,AC18,AC31,AC32,AC25,AC29)
only Service can-depend File,FileIS,InetAddress,ImageIO,Quartz,
                        BufferedImage,FileOS,FTP,NetURL
%RAC6 (AC21,AC22,AC23)
only Servlet can-depend ServletEx,HttpServlet,HttpServletRes
%RAC7 (AC28)
only Service can-declare Security
%RAC8 (AC30,AC37,AC38,AC39,AC40.AC43)
only DAO can-depend Persistence,EntityManager,EMF,EsfingeEMP,
                    EsfingeRepo,EsfingeQB
%RAC9 (AC33,AC34,AC35,AC36)
only Model can-depend Entity,PersistenceGT,PersistenceId,PersistenceGV
%RAC10 (AC2*)
Controller must-useannotation VRaptor
%RAC11 (AC3)
Service cannot-depend Controller, Servlet
%RAC12 (AC7)
Model must-implement Serializable
%RAC13 (AC41)
only Model can-implement Serializable
%RAC14 (AC42)
Servlet,DAO cannot-depend SQL

%Constraints that have been removed:
%(AC20) only Servlet can-declare PrintWriter
%(AC26) Controller cannot-declare Logging
%(AC27) only Service can-declare ArrayList
%(AC44) Servlet cannot-declare List
```

Listing 1: DCL Specification of the latest release of Leona