A Case Study on Improving Maintainability and Evolvability using Architectural Constraints

Leonardo Humberto Guimarães Silva[1], **Ricardo Terra**[2], **Marco Túlio Valente**[2]

[1]Instituto Federal do Norte de Minas Gerais, Brazil

[2]Federal University of Minas Gerais, Brazil

leonardo.silva@ifnmg.edu.br, terra@dcc.ufmg.br, mtov@dcc.ufmg.br

Abstract. *Developers usually rely on patterns and best practices to increase the quality of their projects. However, as projects evolve, it is usual to observe deviations in the use of the patterns and best practices defined during the initial design of a system. This article aims to illustrate the application of a static, domain-specific, and declarative dependency constraint language, called DCL, to express architectural patterns and design principles that contribute to the maintainability and evolvability – and therefore to the internal quality – of a software system. We present in the paper several architectural constraints that demonstrate the benefits achieved by DCL in one motivating system and in five real-world, open-source object-oriented applications.*

1. Introduction

Software architecture is usually defined as a set of design decisions that are critical for the success and the quality of complex software systems. It includes how systems are structured in components and constraints on how components must interact [7, 5, 1]. The definition of an architecture embraces different patterns and best practices. However, as projects evolve, it is usual to observe deviations in the use of the patterns and best practices defined during the initial design of a system [3, 12, 8]. The overall result is a negative impact on quality factors normally achieved by well-planned architectural designs, mainly maintainability and evolvability [2, 15].

In a previous workshop paper we have provided a preliminary discussion on how a dependency constraint language could be used to express best practices and patterns that make software systems easier to evolve and maintain [20]. To illustrate the discussion we have relied on a motivating system whose architecture embraces several patterns and best practices normally found in architectural designs. In the current paper, we extend and complement this previous work by relying not only on a single illustrative system but also on architectural constraints extracted from five real-world, open-source object-oriented applications. In order to express such architectural constraints, we rely on a static, domain-specific, and declarative language called called DCL (Dependency Constraint Language) [18, 17, 19]. DCL is a language that allows developers and project leaders to restrict the spectrum of dependencies than can be established in object-oriented systems. More specifically, DCL is a language that allows developers, architects and project managers to define acceptable and unacceptable dependencies (constraints) according to the initial design of a system. Once the constraints are defined, they are automatically checked by an integrated tool (actually, an Eclipse plug-in).

Our goals with the paper arethreefold. First, we would like to shed light on the importance of respecting the architecture's definition as a system evolves. In fact, architecture erosion can have a crucial impact on many software quality factors, including maintainability and evolvability [8, 2]. Second, we would like to provide insights on the types of architectural constraints that must be controlled in order to avoid a negative impact on the internal quality of a software system. And third, we would like to demonstrate that by using a simple language, such as DCL, it is possible to express architectural constraints that have a clear importance in the maintainability and evolvability of real-world object-oriented applications.

The remainder of this paper is organized as follows. Section 2 introduces the DCL language. Section 3 describes a first case study that illustrates the application of DCL in a small software system. Section 4 extends the previous study with a presentation on how DCL can be used in five real-world software systems. Section 5 presents related work. Finally, Section 6 provides the conclusions and suggestions for future work.

2. The DCL Language

DCL is a declarative, statically checked domain-specific language that supports the definition of structural constraints between modules [18, 17, 19]. The language supports the definition of dependencies originated from accessing methods and fields, declaring variables, creating objects, extending classes, implementing interfaces, throwing exceptions, and using annotations. Essentially, DCL's main purpose is to indicate the presence of structural violations that clearly represent architectural anomalies that contribute to the architectural erosion of a system.

Figure 1 resumes the syntax used to express architectural constraints on DCL. These constraints and the main elements of the DCL language are described next.

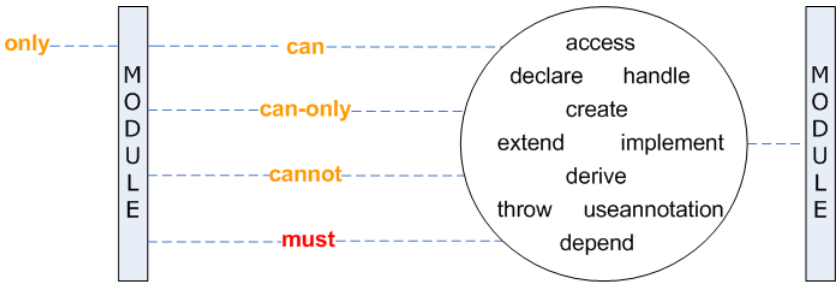

Figure 1. Architectural constraints provided by DCL

Modules: A module is basically a set of classes. Suppose, for example, the following modules of a system:

```
module View: org.foo.view.*
module DataStructure: org.foo.util.**, org.foo.view.Tree
module Remote: java.rmi.UnicastRemoteObject+
```

According to this syntax, the `View` module contains all classes from package `org.foo.view`. The `DataStructure` module contains all classes from package `org.foo.util`, its subpackages and also the class `org.foo.view.Tree`. Finally,

the `Remote` module has all subclasses from `java.rmi.UnicastRemoteObject`.

Divergences: A divergence happens when a dependency that exists in the source code violates an initially defined architectural constraint. In order to capture divergences, DCL supports the definition of the following kinds of constraints between modules:

- `only A can-`*x*` B`[1]: Only classes of module `A` can depend on types defined in module `B`. For example, the constraint `only DAOFactory can-create DAO` defines that only a factory class can create data access objects.
- `A can-only-`*x*` B`: Classes of module `A` can only depend on types defined in module `B`. For example, the constraint `Util can-only-depend Util, $java` defines that utility classes can only depend on their own classes or Java API classes.
- `A cannot-`*x*` B`: Classes of module `A` cannot depend on types defined in module `B`. For example, the constraint `Facade cannot-handle DTO` defines that facade classes cannot manipulate entity classes.

These constraints cover all the usual dependencies of object-oriented languages, including *access*, *declare*, *create*, *extend*, *implement*, *throw*, and *useannotation*. Since the purpose of the constraints is to capture divergences, they alert dependencies that cannot appear in the source code. Basically, constraints *only can* forbid dependencies originated from classes not specified in the source modules of the rules. Constraints *can-only* forbid dependencies to classes not specified in the target modules of the rules.

Absences: An absence denotes the situation when the source code does not establish a dependency that is prescribed by the planned architecture. In order to capture absences, DCL supports the definition of the following constraints:

- `A must-`*x*` B`: Classes of module `A` must depend on types defined in module `B`. For example, the constraint `DTO must-implement java.io.Serializable` defines that entity classes must implement Java's serializable interface.

A complete description of all constraints in DCL can be found at [19]. The DCL language and the `dclcheck` tool – an Eclipse plug-in that implements the proposed language – are publicly available at: `http://www.dcc.ufmg.br/~mtov/dcl`.

3. Initial Case Study: TerraMarket

We will first rely on a motivating system called TerraMarket to illustrate some architectural patterns and best practices commonly applied in modern object-oriented systems [20]. The TerraMarket system handles common activities in a grocery store, such as sales, customer management, ordering etc. The system architecture follows the *Model-View-Presenter* architectural pattern [5], as presented on Figure 2. In this figure, the

[1]The literal *x* refers to dependency type, which can be either more generic (`depend`) or more specific (`access`, `declare`, `create`, `extend`, `implement`, `throw` and `useannotation`).

Model layer contains Business Objects (BOs), Data Transfer Objects (DTOs), and Data Access Objects (DAOs). BOs encapsulate business rules and behavior. DTOs represent domain entities, such as client, product etc. DAOs provide an interface to access an underlying persistence framework. Particularly, TerraMarket uses Hibernate[2] for persistence.

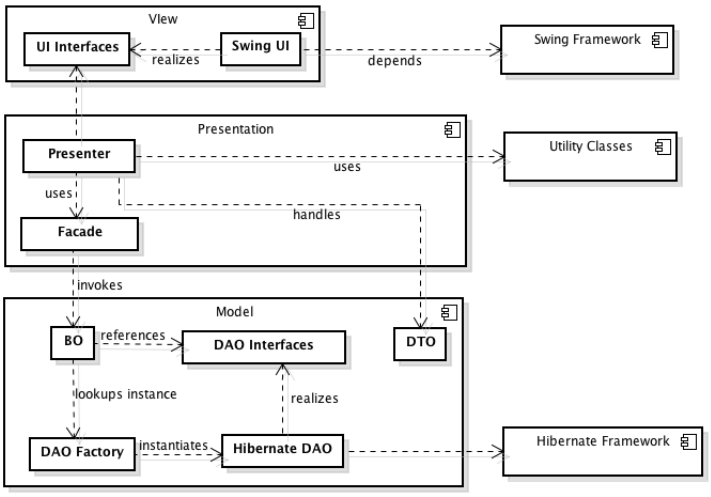

Figure 2. TerraMarket's architecture

Finally, the Presenter layer contains classes that intercept events triggered by the View. It monitors and adapts users entries, manipulating the Model and updating the View. The View layer uses the Swing Framework to create the User Interface. The requests activated by the View are sent to the Facade component, which provides a unique access point to the Model layer. In summary, TerraMarket's architecture is based on patterns (*MVP*, *Factory*, *Facade*, *Data Access Objects* etc) and technologies (Swing, Hibernate, etc) widely used in the development process of modern object-oriented systems.

We will use TerraMarket to illustrate several architectural patterns and best programming and design practices. Initially, we show how patterns could be defined in DCL. After this, we relate the main benefits of the preservation of these patterns to the internal quality of the system. To make the understanding of DCL constraints easier and cleaner the following module's definition will be used in the rest of this section:

```
1  module View: com.tm.view.*
2  module Model: com.tm.model.*
3  module Presenter: com.tm.presenter.*
4  module Facade: com.tm.presenter.Facade
5  module SwingUI: com.tm.view.SwingUI.*
6  module Swing: javax.swing.*
7  module BO: com.tm.model.bo.*
8  module HibernateDAO: com.tm.model.dao.HibernateDAO
9  module DAOFactory: com.tm.model.dao.DAOFactory
10 module DTO: com.tm.model.dto.*
11 module Util: com.tm.Util.*
12 module ApacheUtils: org.apache.*
```

[2]http://www.hibernate.org

3.1. Layered Architecture

The Layered Architecture Pattern provides a development model in which components are organized in hierarchical layers [5]. For example, suppose a strictly layered system $M_i, M_{i-1}, \ldots, M_0$ (where M_0 represents the module in the lowest level of the hierarchy). Therefore, M_i can only use services provided by module M_{i-1}, $i > 0$. Any change in the system that violates this rule is, in fact, undermining its planned architecture.

The following DCL constraints can be used to express that TerraMarket follows an layered architecture:

```
1 view cannot-depend Model
2 only Presenter can-handle Facade
```

In line 1, a constraint prevents the View from establishing any dependency with the Model layer. In line 2, a constraint guarantees the access to the system Facade only to classes from Presenter layer.

Benefits: The previous constraints guarantee that no changes that violate the layered architecture can be applied to TerraMarket. In other words, they make the system easier to maintain and reuse. For example, TerraMarket architecture prescribes that only components in the Presenter layer can have access to the Facade. Moreover, it prevents the View from establishing any kind of dependency with the Model, thus preserving the MVP pattern.

The strict adherence to these constraints also helps developers to detect defective program components. For example, if some data access problem is reported by TerraMarket's users or developers, the failure probably will be located in classes from the Model layer. Likewise, if developers need to change DAO or BO components, it is guaranteed that only the Presenter layer might be affected. Finally, suppose we need to change the Model layer, the View layer will remain intact. In this particular case, only the Facade needs to be updated.

3.2. Programming to Interfaces

Programming to interfaces is a design principle that advocates that developers should separate the interface of a component from its implementation. Essentially, it creates an abstract layer between client and server components [5]. For TerraMarket, the following DCL constraints can be used to express this principle:

```
1 Presenter cannot-handle SwingUI
2 BO cannot-handle HibernateDAO
```

In line 1, a constraint prevents the Presenter layer from manipulating UI implementations directly. Similarly, in line 2, another constraint prevents business objects (BOs) from manipulating data access objects (DAOs) directly.

Benefits: The preservation of the constraints ensures the separation between clients and servers, avoiding the users of a service to know its internal implementation details. For

example, in cases developers need to change or replace the whole UI or DAO components, the other system components will not suffer any impact. As second and third examples, suppose TerraMarket's GUI needs to be migrated from Swing to SWT (Standard Widget Toolkit) or that the persistent layer needs to be migrated from Hibernate to pure JDBC (Java Database Connectivity). In both examples, the proposed changes will have any impact in the Presenter layer or in BO components.

3.3. Creational Patterns

The runtime behavior of object-oriented systems is based on the interaction among different objects. However, in some cases, a class may be instantiated many times unnecessarily, since its objects might had been created only once and shared by their multiple clients [5]. There are various patterns that provide a better way to create objects uniquely [6]. For example, the Singleton pattern restricts the instantiation of a class to just one object. Thus, when an instance of a Singleton is needed, a static method is invoked to return the unique instance.

Following the same principle, the Abstract Factory pattern provides a way to encapsulate a group of individual factories that have a common goal. These individual factories usually have a method with an interface as its return type and that returns the instance (unique in most cases) of the object. Thus, every time a class needs an object, it just need to request the object's specific factory. For TerraMarket, the following DCL constraints can be used to express creational patterns:

```
1 only Facade can-create Facade
2 only DAOFactory can-create HibernateDAO
```

In line 1, the constraint ensures that only the Facade can create its own type, therefore preserving its Singleton behavior. On the other hand, in line 2, a second constraint prevents any class of the system, besides the DAOFactory, to create HibernateDAO objects.

Benefits: The preservation of the mentioned creational constraints ensures that new objects will not be created unresctrictely. For example, it is possible to express a centralized place to create DAOs. Therefore, maintenance related to the creation of DAOs will be located in this centralized program component. Moreover, changes related to the way objects are created, the number of instances allowed, the presence of caches are also centralized in just one class. Likewise, abstract factories provide the advantage of abstraction as we have mentioned before. BO classes could be mentioned as an example, since they cannot establish any kind of dependency with DAO implementations. Therefore, when a DAO implementation is requested to a factory, the corresponding interface will be returned. Furthermore, BO classes do not know what particular implementation they use, which ensures a complete decoupling from DAO implementations.

3.4. Maximize Reuse

Reuse plays a central role in any system development process. Basically, reuse techniques must promote the creation of components that can be shared among multiple systems. Therefore, they must not dependent on specific project classes. For `TerraMarket`, the

following DCL constraints can be used to enforce reusable components:

```
1 Util can-only-depend Util, ApacheUtils, $java
2 DTO can-only-depend DTO, $java
```

In line 1, a constraint ensures that utility classes can only establish dependencies with themselves, with Java API classes and with Apache framework utility classes. Similarly, in line 2, a constraint ensures that data transfer objects can only depend on themselves and on Java API classes.

Benefits: The preservation of the mentioned constraints ensures that classes can be reused by multiple projects. In the long-term, these constraints tend to benefit maintainability and evolvability because the reusable components have already been verified in many scenarios and, therefore, they are less error-prone.

3.5. External Dependencies Control

Real systems tend to be coupled to various external systems, especially frameworks. However, this form of coupling must be controlled in a way that the external framework is just used be the appropriate classes. For example, a persistence framework must be used only by data access classes, no other dependencies must be established. For TerraMarket, the following DCL constraints can be used to minimize coupling to external components and frameworks:

```
1 only SwingUI can-depend Swing
2 only HibernateDAO can-depend Hibernate
```

These constraints only allow UI and DAO implementations to establish dependencies with the Swing and Hibernate frameworks, respectively.

Benefits: The definition of these constraints ensures that no unintended coupling will be established with external frameworks. This form of dependency control avoids that modifications and updates in frameworks, or even a complete change of a framework, affect unrelated classes.

3.6. Promoting Inheritance

It is normal to have groups of classes that share properties and common behavior in software systems. A good practice, usually adopted in these circumstances, relies on the creation of a base class in which the common members are implemented. The following DCL constraints illustrate this practice in the TerraMarket system:

```
1 Presenter must-extend com.tm.controller.presenter.BasePresenter
2 DTO must-derive com.tm.model.dto.BaseDTO, java.io.Serializable
3 IDAO must-extend com.tm.model.dao.IBaseDAO
4 HibernateDAO must-derive BaseHibernateDAO, IDAO
```

The previous constraints share the same purpose: to force a group of classes to inherit from a base class. For example, classes from Presenter layer share properties and common behavior. Thus, in line 1, a constraint ensures that Presenter classes must extend a base class. The same occurs with data transfer objects, since they have the fields `code`, `version` and `ChangingDate`, besides their respective access methods and generic JPA (Java Persistence API) annotations. For this purpose, the constraint in line 2 ensures that all DTOs are serializable (i.e. they can be transmitted in a network) and extend a base class. Finally, in line 3, a constraint ensures that DAO interfaces must extend a base interface and, in line 4, a constraint ensures that Hibernate DAO implementations must extend a base class and implement their respective interfaces.

Benefits: The preservation of these constraints avoids code duplicity and promotes reuse in the development process. For example, by enforcing that every Presenter class extends a base class developers can not implement operations that have already been implemented in the superclass. Moreover, a change that affects all Presenter classes can be easily implemented using their base class. Finally, the explicit definition of these constraints avoids usual errors, like a non-serializable DTO or a DAO implementation without its respective interface.

4. Extended Case Study: Five Real-World Object-Oriented Systems

In this section, we reproduced the experiments of the previous section using five real object-oriented systems. The systems have been evaluated considering the same architectural constraints: layered architectures, programming to interfaces, creational patterns, reuse, external dependencies control and inheritance. Since all systems are open-source, we have just relied on the available documentation to formulate and to validate the proposed DCL constraints. As further work, we have plans to validate such constraints with the developers of each system (possibly using the discussion forum that open-source projects usually maintain on the Internet).

4.1. JabRef

JabRef[3] is a free, Java-based graphical application for managing bibliographical databases. The native file format used by JabRef is BibTeX, the standard LaTeX bibliography format. Among the features provided by JabRef, we can highlight: advanced BibTeX editor, search functions, classification of entries, various import and export formats and customization of BibTeX fields.

External Dependencies Control: The following DCL constraints can provide external dependency control in JabRef:

```
1 only tests.net.sf.jabref.* can-depend junit.framework.TestCase
```

This constraint ensures that only the packages properly designed for testing can depend on the external framework JUnit. Therefore, this constraint helps developers to

[3]http://jabref.sourceforge.net/ - Version 2.6

organize and implement their test cases.

4.2. Jetty

Jetty[4] is a free, Java-based application that provides a Web server and `javax.servlet` container, plus support for Web Sockets, OSGi, JMX, JNDI, JASPI, AJP and many other communication and integration technologies. Jetty can be embedded in devices, tools, frameworks, application servers, and clusters. Figure 3 represents a high level view for Jetty's architecture. A `Connector` module accepts `Http` connections and passes them to a `Jetty Server`. `Handlers` are responsible for receiving requests, producing responses and sending them back to the server. The work is done by threads taken from a thread pool.

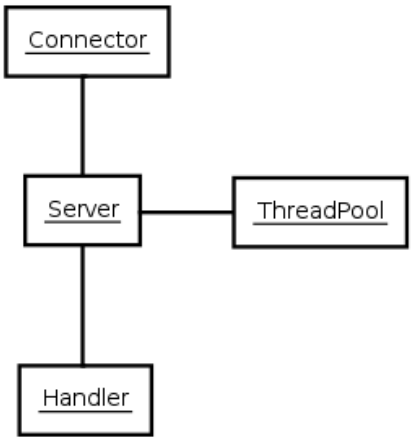

Figure 3. Jetty's architecture

Layered Architecture: The following DCL constraint can be defined to enforce Jetty's architecture:

```
1 module Server: org.mortbay.http*.*
2 module ThreadPool: org.mortbay.util.ThreadPool
3
4 only Server can-handle ThreadPool
```

In line 3, a constraint guarantees the access to the thread pool only to classes from the `Server` layer. In this way, `Connectors` and `Handlers` cannot manipulate or select threads from the thread pool.

Creational Patterns: A factory method is used in Jetty in order to create a log mechanism. Therefore, the following DCL constraint can be defined to express that only the

[4]http://sourceforge.net/projects/jetty - Version 5.1.15

factory can create a log object for the Jetty system:

```
1 module LogFactory: org.mortbay.log
2 module Log: org.apache.commons.logging.Log
3
4 only LogFactory can-create Log
```

Programming to Interfaces: The following DCL constraints can be defined in Jetty to force the use of interfaces:

```
1 module AJP: org.mortbay.http.ajp.*
2 module Handlers: org.mortbay.http.handler.*
3
4 AJP cannot-handle Handlers
```

In line 3, a constraint prevents the package that implements the integration with the web socket AJP from manipulating directly `Handlers` that receive requests from the `Http` server. Instead, AJP sockets can handle the interface `org.mortbay.http.HttpHandler` but not its implementations, found in the `Handlers` module.

4.3. JHotDraw

JHotDraw[5] is a free, Java-based framework to the instantiation of graphic editors. It is a Java version of the original Smalltalk HotDraw framework developed by Kent Beck and Ward Cunningham.

Creational Patterns: The Factory method is used in JHotDraw to create the user interface components at a particular location. This pattern defines an interface for creating an object, delegating to subclasses the decision about which class to instantiate [5]. Menus and toolbars in JHotDraw are created by a class that implements the `ApplicationModel` interface. This interface represents the factory. Therefore, the following DCL constraints can be defined to specify that only the factory can create figures:

```
1 module menus: javax.swing.JMenu+
2 module toolbars: javax.swing.JToolBar+
3 module appModels: org.jhotdraw.app.* ApplicationModel
4
5 only appModels can-create menus and toolbars
```

In the first two lines we specify the packages that constitute the menus and toolbars in JHotDraw. In line 3, we define the classes that constitute a module called `appModels`. In line 5, a constraint guarantees that every menu and toolbar must be created through the factory in `appModels`.

[5]http://www.jhotdraw.org/ - Version 7.5.1

Reuse: The following DCL constraint can be defined to increase the reuse of a `Util` package in the `JHotDraw` system:

```
1 Util can-only-depend Util, java
```

The previous constraint ensures that utility classes can only establish dependencies among themselves and with Java API classes. A similar constraint has been proposed in Subsection 3, for the TerraMarket system.

External Dependencies Control: The following DCL constraint illustrates a dependency control in JHotDraw:

```
1 module dom: org.w3c.dom.**
2 module xmlsax: org.xml.sax.**
3
4 only org.jhotdraw.xml.** can-depend dom, xmlsax
```

In the first two lines we identify and name external modules used to XML manipulation. In line 4 the constraint ensures that only the package `org.jhotdraw.xml.**` can depend on the external modules. Therefore, it makes this dependency easier to understand and update.

Inheritance: The following DCL constraints can be defined to promote the use of inheritance in JHotDraw system:

```
1 module figures: org.jhotdraw.*Figure
2 figures must-extend org.jhotdraw.draw.AbstractFigure
3
4 module views: org.jhotdraw.draw.*DrawingView
5 views must-implement org.jhotdraw.draw.DrawingView
```

The first constraint ensures that every figure must extend an abstract class called `AbstractFigure`. It guarantees that every figure will inherit the same features provided by this abstract class. The second constraint ensures that all views must implement an interface called `DrawingView`. This interface specify the common behavior for every view in JHotDraw.

4.4. JUnit

JUnit[6] is a framework for writing and running automated tests. The framework is integrated with Ant[7] and is used by developers to implement unit tests in Java. The goal is to accelerate programming and increase the quality of the code. Basically, JUnit provides comprehensive assertion facilities to verify expected versus actual results.

[6] http://www.junit.org/ - Version 4.9
[7] http://ant.apache.org/

Reuse: Reuse is a key property of every framework. The following DCL constraint can be defined to increase the reuse of the framework package in the `JUnit` system:

```
junit.framework.* can only depend junit.framework.*, java
```

Basically, this constraint guarantees that the framework package will not depend on packages that control user interfaces, extensions, or the way test classes are invoked.

Inheritance The following DCL constraint enforces the use of inheritance in the `JUnit` system:

```
junit.extensions.* must-extend junit.framework.*
```

This constraint ensures that every extension created for the framework must extend the framework itself. Therefore, it prevents the extension package from being misused.

4.5. Jung

Jung[8] is a free, Java-based framework that provides a common and extendible language for modeling, analysing, and visualizing data that can be represented as a graph or network. The framework allows Jung-based applications to make use of the extensive built-in capabilities of the Java API, as well as those of other existing third-party Java libraries.

Programming to Interfaces The following DCL constraints can be defined to illustrate the use of interfaces in the JUng system:

```
module Visualization: edu.uci.ics.jung.visualization.*
module Implementation: edu.uci.ics.jung.graph.impl.*

Visualization cannot-handle Implementation
```

The previous constraint ensures that the classes responsible for the visualization of a graph cannot handle the implementation of any graph element (vertices and edges). Instead, visualization classes must use the interfaces in `edu.uci.ics.jung.graph` to manipulate graph's elements.

5. Related Work

Related work on defining constraints to enforce architectural patterns can be organized in three main groups: Dependency Structure Matrix (DSM), Structural Constraint Language (SCL), and Reflexion Model Tools (RMT):

Dependency Structure Matrix (DSM): Sangal et al. have proposed the use of Dependency Structure Matrixes (DSM) to reveal existing dependencies and the underlying

[8] http://jung.sourceforge.net/ - Version 2.0.1

architectural pattern of complex software systems [16]. DSM are simple adjacency matrixes used to represent dependencies between modules of a software system. Sangal et al. propose the use of design rules in order to highlight DSM entries that violate the planned architecture of a system. However, DSM's design rules support the definition of only two forms of dependencies: `can-use` and `cannot-use`. Lattix Inc's Dependency Manager (LDM) is an architecture visualization and conformance tool based on dependency matrixes and design rules[9]. Using the GUI of this tool, architects can filter the dependencies considered in design rules. For example, they can specify that a particular `cannot-use` rule only disallows the creation of objects. On the other hand, DCL provides architects with concrete syntax to filter dependency relations (including `access`, `declare`, `create` etc) in a way that is independent from any particular GUI feature. Moreover, DCL supports other rules, including `only can` and `must`.

Structural Constraint Language (SCL): SCL (Structural Constraint Language) [9] – and its predecessor FCL (Framework Constraint Language) [10] – are logic-based languages for specifying a wide range of structural design constraints. The central goal of these languages is to check whether the source code respects its intended design, which usually requires more detailed constraints than those needed to express only architectural intent. To support design level constraints, SCL relies on an unrestricted first-order logic language with a rich set of functions to obtain information about the abstract syntax of object-oriented systems. However, SCL's expressiveness comes at the expense of a rather heavyweight notation – as admitted by the language authors – and poor performance. Furthermore, due to its focus on class and method level constraints, SCL lacks abstractions to define modules (i.e. logical collection of classes and packages), which are exactly the key abstractions needed when expressing architectural constraints.

An alternative for tackling the complexity inherent to full Prolog-like languages consists in defining a small, domain-specific language on top of such languages. For example, LogEn is an attempt in this direction [4]. In order to reduce complexity and increase performance, LogEn relies on Datalog, a restricted and optimized subset of Prolog. In order to express architectural intent, the language provides means for restricting dependencies between logical groups of code elements, called ensembles. However, since logEn's core is still a logic language, it preserves much of the expressive power and complexity typical from such languages. Moreover, its syntax is not simple and self-explaining enough for defining high-level architectural constraints, as recognized by the language authors. In order to provide a more comprehensive notation for expressing architectural constraints, LogEn authors have proposed a visual language, called VisEn, from which LogEn constraints can be automatically generated. In VisEn, boxes denote ensembles and arrows are used to denote allowed dependencies of any kind (including `access`, `declare`, `derive` etc). Therefore, VisEn does not allow architects to filter the various forms of dependency relations that are possible in object-oriented systems.

Reflexion Model Tools (RMT): Reflexion models require developers to provide a high-level model of the planned system architecture and a declarative mapping between such model and the source code model [13, 14]. A RM-based tool – such as the SAVE

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

Eclipse-based plug-in [11, 12] – highlights convergent, divergent, and absent relations between the high-level model and the source code model. However, we believe that DCL supports a richer set of relations between modules than the language used in RMs. Moreover, our language is designed to foster architecture conformance by construction, i.e. using our language modifications that violate the planned architecture are detected as soon as they are implemented in the source code.

6. Conclusions

The use of patterns and best practices is always recommended in any software development process. However, as projects evolve, it is usual to observe deviations in the use of the patterns and best practices defined during the initial design of a system. Thereby, software maintenance becomes a more difficult task. For that reason, using first a motivating system, we have demonstrated how to use the DCL language to check design patterns and best architectural practices that contributes directly to the internal quality of a system. In addition, we conducted an extended case studies using five real object-oriented systems to demonstrate how DCL can be used to increase classical software quality factors, mainly maintainability and evolvability.

As a future work, we aim to go deeper in the case studies mentioned to gather quantitative data to corroborate the benefits of the use of DCL during the evolution of the systems. We also intend to integrate DCL with a UML modeling tool in order to be able to represent DCL constraints while defining UML models. Furthermore, we intend to propose constraints to be used during the development process of a system.

Acknowledgments: This research has been supported by grants from FAPEMIG, CAPES, and CNPq.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd, edition, 2003.

[2] P. Clements and M. Shaw. The golden age of software architecture revisited. *IEEE Software*, 26(4):70–72, 2009.

[3] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

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

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

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[7] D. Garlan and M. Shaw. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[8] J. Gurp and J. Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

[9] D. Hou and H. J. Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[10] D. Hou, H. J. Hoover, and P. Rudnicki. Specifying framework constraints with FCL. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 96–110, 2004.

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

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

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

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

[15] L. Passos, R. Terra, R. Diniz, M. T. Valente, and N. das Chagas Mendonca. Static architecture conformance checking – an illustrative overview. *IEEE Software*, 27(5):82–89, 2010.

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

[17] R. Terra and M. T. Valente. Towards a dependency constraint language to manage software architectures. In *Second European Conference on Software Architecture (ECSA)*, volume 5292 of *Lecture Notes in Computer Science*, pages 256–263. Springer, 2008.

[18] R. Terra and M. T. Valente. Verificação estática de arquiteturas de software utilizando restrições de dependência. In *II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*, pages 1–14, 2008.

[19] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12):1073–1094, 2009.

[20] R. Terra and M. T. Valente. Definição de padrões arquiteturais e seu impacto em atividades de manutenção de software. In *VII Workshop de Manutenção de Software Moderna (WMSWM)*, pages 1–8, 2010.