Towards a Dependency Constraint Language to Manage Software Architectures

Ricardo Terra and Marco Tulio de Oliveira Valente

Institute of Informatics, PUC Minas, Brazil
`rterrabh@gmail.com`,`mtov@pucminas.br`

Abstract. This paper presents a dependency constraint language that allows software architects to restrict the spectrum of dependencies that can be presented in a given software system. The ultimate goal is to provide designers with means to define acceptable and unacceptable dependencies according to the planned system architecture. Once defined, such restrictions will be automatically enforced by a tool, thus avoiding silent erosions in the architecture. The paper also presents first results of applying the language in a Web-based system.

1 Introduction

Software architecture is usually defined as the set of design decisions that have impact on each aspect of the construction and evolution of large software systems. This includes how systems are structured into components and constraints on how components should interact [3,2]. Despite its unquestionable importance, the documented architecture of a system – if available at all – usually does not reflect its actual implementation. In practice, deviations from the planned architecture are usually common, due to unawareness by the developers part, conflicting requirements, technical difficulties etc [6]. More important, such deviations are usually not captured and resolved, leading to the phenomena known as architecture erosion and architectural drift [10].

This paper is centered on the observation that improper inter-module dependencies are one of the principal sources of architectural violations. For instance, suppose a strictly layered system $M_p, M_{p-1}, \ldots, M_0$ (where M_0 represents the module in the lowest level of the hierarchy). Therefore, in this system, M_i can only use services provided by module M_{i-1}, $i > 0$. Any system change that violates this rule is, in fact, undermining its planned architecture. As another example, suppose a web-system that includes a controller module C and a module P that encapsulates persistence services. Clearly in this system, C is the only module that can handle HTTP requests and responses (using servlets or another similar technology). In the same way, P is the only module that can rely on the services provided by a persistence framework (such as Hibernate, for example).

Current mainstream programming languages support information hiding by the means of interfaces and visibility modifiers (such as `public`, `private`, and `protected`). However, they do not provide means to restrict inter-module dependencies. In practice, any public service provided by a module (or class) M

can be used by any other system module. In order to tackle this problem, we are working on a dependency constraint language that allows software architects to restrict the spectrum of dependencies that can be presented in a given software system. This language should provide designers with means to define acceptable and unacceptable dependencies according to the planned system architecture. Once defined, such restrictions can be automatically enforced by a tool integrated to common programming environments (such as Eclipse). Thus, our ultimate goal is to provide architectural conformance by construction, using a static, declarative dependency constraint language.

The remainder of this paper is organized as follows. Section 2 provides a preliminary description of the dependency language that we are designing. Section 3 illustrates the application of the proposed language in a simple case study. Section 4 discusses related work and Section 5 concludes.

2 Dependency Constraint Language

The main purpose of the proposed language is to support the definition of constraints between modules. In our notation, a module is just a set of classes. Suppose, for example, the following module definitions:

```
module A: org.foo.persistence.*
module B: org.foo.view.*, org.foo.model.Ticket, org.foo.model.Driver
```

Module `A` includes all public classes from the package `org.foo.persistence`. Module `B` includes all public classes from the package `org.foo.view` and classes `Ticket` and `Driver` from the package `org.foo.model`.

The language supports the definition of the following constraints:

– *Only classes from module A can depend on types defined in module B*, where the possible dependencies are as follows:

  - `only A can-access B`: only classes declared in module `A` can access non-private members of classes declared in module `B`. Access in this case means calling methods, reading or writing to fields.
  - `only A can-declare B`: only classes declared in module `A` can declare variables of types declared in module `B`.
  - `only A can-handle B`: only classes declared in module `A` can access and declare variables of types declared in module `B`. In other words, this is an abbreviation for `only A can-access, can-declare B`.
  - `only A can-create B`: only classes declared in module `A` can create objects of classes declared in module `B`.
  - `only A can-extend B`: only classes declared in module `A` can extend classes declared in module `B`.
  - `only A can-implement B`: only classes declared in module `A` can implement interfaces declared in module `B`.
  - `only A can-throw B`: only methods from classes declared in module `A` can throw exceptions declared in module `B`.

– *Classes declared in module A cannot depend on types defined in module B*, where the dependencies that can be forbidden are as follows:
  - `A cannot-access B`: no classes declared in module `A` can access non-private methods or fields of classes declared in module `B`.
  - `A cannot-declare B`: no classes declared in module `A` can declare variables of types declared in module `B`.
  - `A cannot-handle B`: no classes declared in module `A` can access or declare variables of types declared in module `B`.
  - `A cannot-create B`: no classes declared in module `A` can create objects of classes declared in module `B`.
  - `A cannot-extend B`: no classes declared in module `A` can extend classes declared in module `B`.
  - `A cannot-implement B`: no classes declared in module `A` can implement interfaces declared in module `B`.
  - `A cannot-throw B`: no methods from classes declared in module `A` can throw exceptions declared in module `B`.

– *Classes declared in module A must depend on types defined in module B*, where the dependencies that can be required are as follows:
  - `A must-extend B`: all classes declared in module `A` must extend a class declared in module `B`.
  - `A must-implement B`: all classes declared in module `A` must implement at least an interface declared in module `B`.

3 Case Study

In order to illustrate and motivate the need of a dependency language we have devised and implemented the main modules of an electronic government information system used by state's department of motor vehicles to handle traffic law violations, such as exceeding the speed limit, parking in an unauthorized area, driving without license etc. The devised system, called Traffic Ticket Online, has a web-based user interface that drivers can use to search for detailed information about their tickets and also paying tickets online. On the other hand, traffic authorities use the system to register tickets and perform associated operations.

Architecture: As described in Figure 1, the architecture of the system follows the Model-View-Controller (MVC) architectural pattern [2]. The Model layer contains Business Objects (BO), Data Transfer Objects (DTO), and Data Access Objects (DAO). Business Objects represent objects that encapsulate business rules and behavior. Data Transfer Objects represent domain entities such as drivers, tickets, law violations etc. Data Access Objects provide an abstract interface to the underlying persistence framework. Particularly, in the current system implementation we are using Hibernate for object/relational persistence. The Controller layer contains components that monitor user inputs, manipulate the Model, and update the View accordingly. The Traffic Ticket Online architecture prescribes that the Struts framework should be used by the Controller to handle HTTP requests. Such requests are then forwarded to a facade component, which provides a unique point

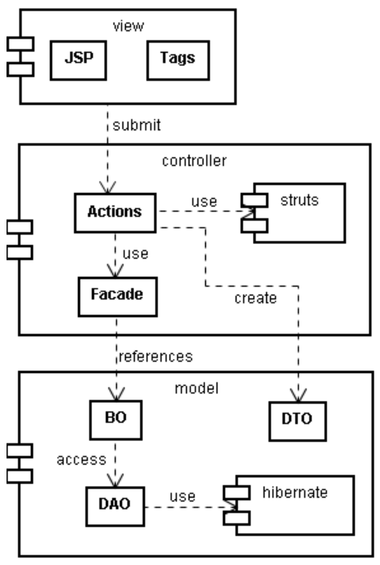

Fig. 1. Traffic Ticket Online Architecture

of access to the model. Finally, the View layer is composed by Java Server Pages (JSP). In summary, the architecture of the system relies on patterns (MVC, Factory, Facade, Business and Data-Access Objects etc) and frameworks and technologies (Hibernate, Struts, JSP etc) that are widely used nowadays when architecting web-based systems.

Constraints: Figure 2 illustrates how the proposed language can be used to regulate acceptable and unacceptable dependencies in the Traffic Ticket Online system. Initially, a sequence of `modules` definitions are used to group related classes (lines 1-12). It can be observed that the defined modules closely resemble the modules presented in the architectural view of the system depicted in Figure 1. This provides evidence that the proposed language can regulate dependencies between entities normally used by software architectures to describe their systems.

In lines 13-23, sequences of *only* constraints are defined. Essentially, such constraints are fundamental to guarantee that the original MVC architecture is preserved during the evolution of the system. For example, some of the constraints define that only classes from the Controller layer can handle (i.e. access and declare) types from the `Facade` module and from the Struts framework (line 16). This avoid for example the View layer to bypass the Controller and access directly the Model. Moreover, a specific constraint specifies that the `Facade` is the only module in the Controller layer that can handle types associated to business objects (line 18). In summary, the constraints express a key property about the dependencies directions in the MVC pattern: the Controller should depend on the Model, but the Model does not depend on the Controller. Instead, the Model only depends on the Hibernate persistence framework (line 20).

```
 1: %Modules
 2: module Tags:            com.tto.view.taglib.*
 3: module Controller:      com.tto.controller.action.*
 4: module ControllerExcp:  com.tto.controller.exception.*
 5: module Facade:          com.tto.controller.facade.*
 6: module BO:              com.tto.model.bo.*
 7: module DAO:             com.tto.model.dao.*
 8: module HibernateDAO:    com.tto.model.dao.hibernate.*
 9: module DTO:             com.tto.model.dto.*
10: module ModelExcp:       com.tto.model.exception.*
11: module Hibernate:       org.hibernate.*
12: module Struts:          com.opensymphony.xwork2.*

13: %Can constraints
14: only Controller can-create, can-declare DTO
15: only Controller can-access com.tto.service.FacadeService
16: only Controller can-handle Facade, Struts
17: only Controller, Facade can-throw ControllerExcp
18: only Facade, BO can-handle BO
19: only BO can-handle DAO
20: only HibernateDAO can-handle Hibernate
21: only com.tto.model.BOFactory can-create BO
22: only com.tto.model.DAOFactory can-create DAO
23: only BO, DAO can-throw ModelExcp

24: %Cannot constraints
25: Facade cannot-access DTO

26: %Must constraints
27: BO must-extend com.tto.model.bo.DefaultBO
28: Controller must-extend com.opensymphony.xwork2.ActionSupport
29: DAO must-implement com.tto.model.IDefaultDAO
30: DTO must-extend com.tto.dto.Persistent
31: com.tto.dto.Persistent must-implement java.io.Serializable
32: Facade must-implement com.tto.facade.IFacade
33: HibernateDAO must-extend
       com.tto.model.dao.hibernate.DefaultHibernateDAO
34: HibernateDAO must-implement DAO
35: Tags must-implement javax.servlet.jsp.tagext.JspTag
```

Fig. 2. Dependency Constraints Rules for the Traffic Ticket Online system

It is also important to mention the role of the different *can* relations types in the constraints of lines 13-25. For example, using the proposed constraint language, it was also possible to make explicit the difference between factories and clients of a given type. For example, there is a constraint that requires that BOs can only be created in the `BOFactory` class (line 21). Moreover, another constraint expresses that only the `Facade` can rely on `BO`'s services, (but it cannot create such objects, as described). As another example, exceptions defined in the module `ControllerExcp` can only be throwed by methods in the `Controller` and `Facade modules` (line 17).

In lines 26-35, sequences of *must* constraints are defined. Such constraints are used to guarantee that all classes that integrate a given module implement or extend a given type. Usually, this type can be defined in another system module or can be provided by an external framework. As an example of the first case, each `BO` must extend an internal class named `DefaultBO` (line 27). As an example of the second case, each class in the `Tags` module must implement the `javax.servlet.jsp.tagext.JspTag` interface (line 35). Such constraints are important to guarantee that the system correctly reuses services provided by other classes and frameworks. In some way, they contribute to guide developers to use external frameworks correctly, as prescribed by the system architecture.

4 Related Work

Over the past decade, at least the following techniques have been proposed to deal with the architecture erosion and drift problems.

Constraint Languages: Sangal et al. have proposed the use of Dependency Structure Matrixes (DSM) to reveal existing dependencies and the underlying architectural pattern of complex software systems [11]. They also propose the use of design rules in order to highlight DSMs entries that violate the planned architecture. The dependency constraint language proposed in this paper is inspired in the design rules language. However, Sangal's language supports the definition of only two forms of relations between modules: `can-use` and `cannot-use`. On the other hand, our language allows the definition of a richer set of relations.

Architectural Recovery and Conformance Tools: Architectural recovering frameworks rely on reengineering technologies to extract high-level architectural models from existing systems [13,4,8]. The main challenge of such frameworks is recovering models that are similar to the ones sketched by developers, in terms of conciseness, abstraction level and architectural elements. Reflexion models (RM) aim to handle such problem by requiring developers to provide a high-level model of the planned system architecture and a declarative mapping between such model and the source code [9]. A RM-based tool (such as the SAVE Eclipse plug-in [5,6]) highlights convergence, divergence and absence relations between the high-level model and the source code. However, we believe that our approach 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 soon after they are implemented in the source code.

Architectural Description Languages (ADL): ADLs represent another alternative to enforce architectural conformance by construction [7]. Such languages allow developers to express the architectural behavior and software systems structure in an abstract, declarative language. Code generation tools can then be used to map architectural descriptions to source code in a given programming language. However, such approaches normally require the use of specific architecture-based development tools and compilers, in order to keep the generated code synchronized with the architectural specification. A variant of this approach advocates the extension of current programming languages with architectural modeling constructs, which in practice demand developers to dominate a completely new programming language [1]. Our approach tackles this problem proposing a simple and declarative language to define dependencies constraints between modules. Stafford and Wolf have proposed a dependence analysis technique for use with ADLs [12]. Therefore, their main objective is to support architectural conformance at the ADL-level, i.e. the proposed technique does not require the availability of the system source code.

5 Conclusions and Future Work

Our research is centered on three hypotheses: (i) that improper inter-module dependencies are an important source of architectural violations; (ii) that a small, declarative dependency constraint language as the one presented in the paper can be employed to detect many of such violations; (iii) that such language can be integrated with a small overhead to the common edit/compile/run cycle performed by developers when building software systems with modern IDEs, thus enforcing architecture conformance by construction.

The Traffic Ticket Online system presented in Section 3 has provided us with encouraging feedback about the application of our dependency language. However, in order to provide more robust arguments that can support the first two hypotheses mentioned above we are starting to apply the proposed language to a human resource management system, used by the Brazilian Federal Government to handle information about public employees. Previous versions of this system are accessible from a CVS repository, which will allow us to apply our dependency language to several of such versions. The ultimate goal is to demonstrate that the proposed language could have been used to prevent important violations perpetrated to the original system architecture.

Our initial plan is starting the second case study by first defining the dependency constraints of the evaluated system. Next, we will check such constraints manually, i.e. our plan is starting the implementation of a tool that can check the proposed constraint language only after finishing this second case study. The reason is that we believe that the study will help us to improve the language, possibly suggesting new kinds of dependencies not supported by its initial version.

References

1. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. In: 22nd International Conference on Software Engineering, pp. 187–197 (2002)
2. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley, Reading (2002)

3. Garlan, D., Shaw, M.: Software Architecture Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
4. Kazman, R., Carrière, S.J.: Playing detective: Reconstructing software architecture from available evidence. Automated Software Engineering 6(2), 107–138 (1999)
5. Knodel, J., Muthig, D., Naab, M., Lindvall, M.: Static evaluation of software architectures. In: 10th European Conference on Software Maintenance and Reengineering, pp. 279–294 (2006)
6. Knodel, J., Popescu, D.: A comparison of static architecture compliance checking approaches. In: IEEE/IFIP Working Conference on Software Architecture, p. 12 (2007)
7. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
8. Muller, H.A., Klashinsky, K.: Rigi a system for programming-in-the-large. In: International Conference on Software Engineering, pp. 80–87 (1988)
9. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software reflexion models: Bridging the gap between source and high-level models. In: SIGSOFT Symposium on Foundations of Software Engineering, pp. 18–28 (1995)
10. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. Software Engineering Notes 17(4), 40–52 (1992)
11. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software architecture. In: 20th Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 167–176 (2005)
12. Stafford, J.A., Wolf, A.L.: Architecture-level dependence analysis for software systems. International Journal of Software Engineering and Knowledge Engineering 11(4), 431–451 (2001)
13. Yan, H., Garlan, D., Schmerl, B.R., Aldrich, J., Kazman, R.: DiscoTect: A system for discovering architectures from running systems. In: 26th International Conference on Software Engineering, pp. 470–479 (2004)