Automatic architectural enforcement

Anders Mattsson

Combitech AB, P.O. Box 1017, SE-551 11 JÖNKÖPING, Sweden,
anders.mattsson@combitech.se

Abstract. Automatic architectural enforcement would be very beneficial especially in product line development using open source practices where there is very limited or no access to the architects and the architecture is of paramount importance. However, current techniques for modelling software architecture do not support the modelling of architectural design rules which means that architectural enforcement is achieved by manual reviews. This paper addresses this problem by proposing how architectural design rules could be expressed in UML in a meta-model for the system model.

1 Introduction

Maintaining a clear and consistent architecture is of paramount importance to achieve a successful product line from which new products can be spawned during a long time. The current state of practice is to document the architecture informally and to use manual reviews to enforce it, this is hard and error prone work in closed source single site development but is even harder in distributed open source-like development with limited or no access to the architects. It would therefore be very beneficial to be able to automatically enforce the architecture on the detailed design, especially in product line development using open source practices. One approach to achieve this could be to use Model-Driven Development (MDD) [1]. In MDD, design artefacts are represented as formal or semi-formal models to allow tool-supported automation of time consuming and error prone manual tasks. However, one class of design artefact is excluded from current MDD approaches, in spite of being recognised in current research as being very important [2-7]: architectural design rules. In this paper we propose an approach to removing this anomaly and thus allowing automatic enforcement of the architecture.

This paper is organized as follows. In section two we clarify the role of architectural design rules. In section three we present MDD in relation to architectural design rules. In section four we present our approach to model architectural design rules and relate it to the body of literature. To demonstrate the approach an example is given in section five. Finally, we present a summary and future research direction in section six.

Dagstuhl Seminar Proceedings 08142
Combining the Advantages of Product Lines and Open Source
http://drops.dagstuhl.de/opus/volltexte/2008/1545

2 Architectural design rules

IEEE has established a set of recommended practices for the architectural description of software-intensive systems [8] which are followed by several architectural design methods [9-12]. A common understanding in architectural methods is that the architecture is represented as a set of components related to each other [13, 14]. The components can be organized into different views focusing on different aspects of the system. Different methods propose different views; typical views are a view showing the development structure (e.g. packages and classes), a view showing the runtime structure (processes and objects) and a view showing the resource usage (processors and devices). In any view each component is specified with the following:

- An interface that documents how the component interacts with its environment.
- Constraints and rules that have to be fulfilled in the design of the component.
- Allocated functionality.
- Allocated requirements on quality attributes.

A typical method of decomposition (see for instance [9]and [11]) is to select and combine a number of patterns that address the quality requirements of the system and use them to divide the functionality in the system into a number of elements. Child elements are recursively decomposed in the same way down to a level where no more decomposition is needed, as judged by the architect. The elements are then handed over to the designers who detail them to a level where they can be implemented. For common architectural patters such as Model-View-Controller, Blackboard or Layers [15] this typically means that you decompose your system into subsystems containing different kinds of classes (such as models, views and controllers). However the instantiation into actual classes is often left to the detailed design, for two main reasons:

1. Functionality will be added later, either because it was missed or because a new version of the system is developed, so more elements will be added later that also have to follow the design patterns decided by the architect.
2. It is not of architectural concern. The concern of the architect is that the design follows the selected architectural patterns, not to do the detailed design.

This means that a substantial part of the architecture consists of design rules on what kinds of elements, with behavioural and structural rules and constraints, there should be in a certain subsystem.

The importance of architectural design rules is also highlighted in current research in software architecture which is focused on the treatment of architectural design decisions as first class entities [2, 4-7], where architectural design decisions impose rules and constraints on the design together with rationale. However, there is not yet any suggestion on how to formally model these design rules. The current suggestion is to capture them in text and to link them to the resulting design. This may be sufficient for rules stating the existence of elements (“ontocrisis” in [5]) in the design, such as a subsystem or an interface, since the architect can put the actual element (i.e.

a certain subsystem) into the system model at the time of the decision. It is however not sufficient for rules on potentially existing elements ("diacrisis" in [5]) such as rules on what kinds of elements, with behavioural and structural rules and constraints, there should be in a certain subsystem, since the actual elements are not known at the time when the design decision is made. Instead, the rule-based design occurs later in the detailed design phase, and involves other persons, potentially even in a different version of the system.

3 MDD and Architectural Design Rules

The basic idea of MDD is to capture all important design information in a set of formal or semi formal models that are automatically kept consistent by tools. The purpose is to raise the level of abstraction at which the developers work and to eliminate time consuming and error prone manual work in keeping different design artefacts consistent [1].

MDD requires that the work products produced and used during development is captured in models to allow automation of non-creative tasks such as transformation of models into code or conformance checks between different design artefacts. There exist several approaches to Model-Driven Development (MDD) such as OMG's MDA [16], Domain Specific Modelling (DSM) [17, 18], and Software Factories [19] from Microsoft. Since neither these nor any architectural design methods address the problem on how to model architectural design rules, the state of practice is to describe architectural design rules in informal text. This means that we have to rely on manual routines to make sure that they are followed.

4 Modelling architectural design rules

There are a large number of Architectural Description Languages (ADL) [21-23], including UML, specified for describing the architecture of software systems. These typically allow one to specify components with relations and interfaces together with functional and structural constraints. They do not however provide any means to specify constraints or rules on groups of conceptual components only partly specified by the architect that are intended to be instantiated and detailed by designers. For instance, in the project we reported on in [20], the architects needed to specify a set of rules on behaviour and relations on a conceptual component called arcComponent without knowing which specific arcComponents would be relevant. Rather, they were to be identified and designed by the designers according to the rules stated by the architects.

The problem of modelling design rules is essentially the same problem as modelling the solution part of a design pattern since the solution specifies rules to follow in the design. There are a number of suggestions on how to formally model design pattern specifications [24-29]. They are however all limited in what kind of rules they can formalize, typically only structural rules. In addition all approaches except [28] require the architect to use mathematical formalisms such as predicate logic and set theory that may be unfamiliar or hard to understand both for architects and developers.

Since UML is a modelling language familiar both to architects and designers we propose an approach where we use UML to specify constraints, the architectural design rules, on a system model also in UML. Similar to [29] we propose to use a UML profile model to constrain the system model but instead of defining constraints of stereotypes in OCL we propose to model these in a meta-model in UML. A meta-model defines the modelling concepts to be used when building a model in the same way that a system-model defines the elements that exist in a system [30]. So, if one uses UML in a meta-model one can model rules and constraints on a system model in the same way one can model rules and constraints on a system in a system model. To use UML at the meta-model level one simply lifts all the concepts in UML up one meta-level. These meta-model elements are then transformed into stereotypes to be used in the system model, carrying the constraints given by the meta-model. In Table 1 interpretations at the meta-model level for the most basic UML concepts are given. To highlight the regularity in the interpretation the normal model level interpretations are also given.

Table 1. Meta-model interpretation of UML concepts

UML Concept	Normal interpretation	Meta-model interpretation
Class	Represents a type of object either in the system or in the problem domain. All objects of a class share the properties of the class	Metaclass, represents a type class in the system model. All the classes share the properties of the metaclass. A metaclass represents a stereotype applicable to classes in the system-model
Association between class A and class B	Represents a relation between objects of class A and class B. For example that a person may own a number of cars or that a controller controls two pumps.	MetaAssociation, represents a relation between classes of metaclass A and metaclass B. The multiplicity on one side specifies how many classes a class of the metaclass of the other side may be associated with. A meta-association represents a stereotype applicable to associations in the system model.
Composition where class A contains Class B	Means that an object of class A contains a number of objects of Class B.	MetaComposition, means that a class of MetaClass A contains a number of classes of MetaClass B. A meta-composition represents a stereotype applicable to compositions in the system model.
Inheritance where class B inherits class A	Means that Class B is a subtype of Class A in such a way that each object of Class B has all the properties of Class A as well as the properties of Class B.	MetaInheritance, means that MetaClasss B is a subtype of MetaClass A in such a way that each Class of MetaClass B has all the properties of MetaClass A as well as the properties of MetaClass B. This may be interpreted in the way that a class of MetaClass B shall inherit a class of MetaClass A since all classes of MetaClass A has all the properties of MetaClass A.

5 An example

To demonstrate the approach we use an example. A common method to as far as possible model architectural design rules in the system model is to use a combination of abstract classes, accompanied by design rules in natural language. This is illustrated in the example in Fig. 1 where Observer and Subject are abstract classes implementing part of the Observer pattern [31] and the comments contain the textual part of the design rules that apply to the elements represented by the packages Distribution and Data_Store.

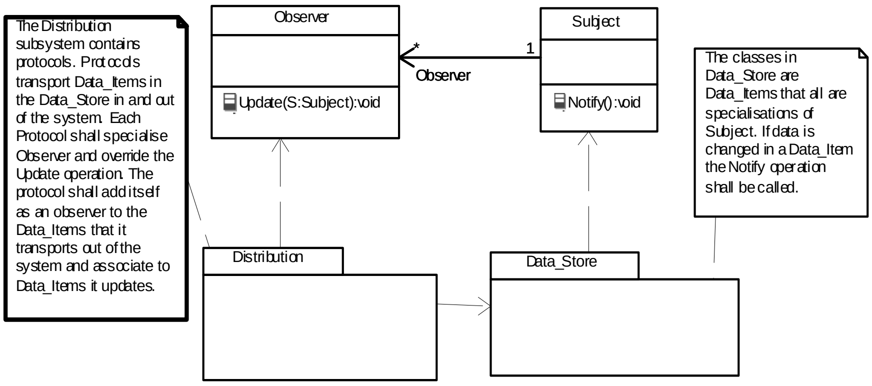

Fig. 1. A traditional way of modelling architectural design rules

If we instead model these rules in a metamodel rather than in the system model, using UML we get a model such as that in Fig. 2. The circles R1 to R6 point out how the corresponding rules below, directly fetched from the comments in Fig. 1, are represented in the model.

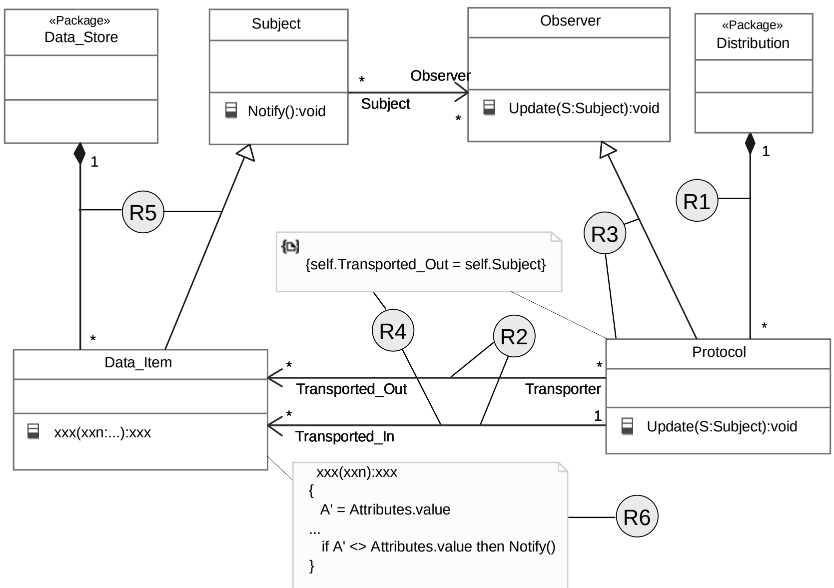

Fig. 2. Observer pattern in a meta-model

R1. “The Distribution subsystem contains protocols”

R2. “Protocols transport Data_Items in the Data_Store in and out of the system.”

R3. “Each Protocol shall specialise Observer and override the Update operation.”
R4. “The protocol shall add itself as an observer to the Data_Items that it transports out of the system and associate to Data_Items it updates”
R5. “The classes in Data_Store are Data_Items that all are specialisations of Subject.”
R6. “If data is changed in a Data_Item the Notify operation shall be called.”

A system model conforming to this model is for instance the one in Fig. 3. This figure also shows how the classes in the metamodel have been transformed into stereotypes. A non conforming model would be one that had more than one protocol that “transported_in” any of the data items or one that had a protocol associated with another protocol. This simple example shows that it is possible to model architectural rules at the meta-model level that is not possible to model at the system-model level, in a straight forward way in standard UML.

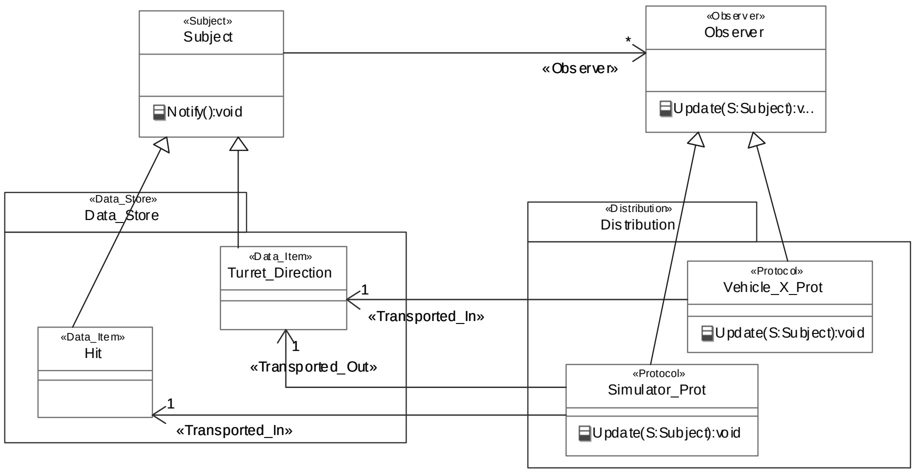

Fig. 3. System-model conforming to the meta-model

6 Summary and future research

Enforcement of architectural deign rules are important in any development project but especially in product line development where the architecture must hold for a long time of development of new products without eroding. In the current body of literature there are no complete solutions on how to model architectural design rules. This means that we have to rely on laborious and error prone manual work to enforce the architectural rules on the system design. In the context of distributed open source-like development of product lines this poses an especially big problem since there are limited or no access to the architects at the same time as the architecture is of paramount importance. This paper presents an idea on how to solve this problem based entirely on standard UML in a way familiar to both architects and designers that at the same time are amendable to automation.

We are now extending this work in the following directions:

- The architectural rules of full industrial systems will be documented using this technique.
- The transformation from meta-model UML constructs to UML profile stereotypes will be formalized.
- Tooling for checking a system model against architectural rules in a meta-model will be developed.
- The approach will be tested in a running project to get feedback on the usability in practice.

Acknowledgements. This research has been financially supported by the ITEA project COSI (Co-development using inner & Open source in Software Intensive products) (http://itea-cosi.org) through Vinnova (http://www.vinnova.se/).

References

1. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal 45 (2006) 451-461
2. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. Proceedings of the Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 05) (2005) 109-120
3. Jansen, A., Bosch, J.: Evaluation of tool support for architectural evolution. (2004) 375-378
4. Jansen, A., van der Ven, J., Avgeriou, P., Hammer, D.K.: Tool Support for Architectural Decisions. Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 07), Mumbay, India (2007) 44-53
5. Kruchten, P.: An ontology of architectural design decisions in software intensive systems. 2nd Groningen Workshop on Software Variability (2004) 54-61
6. Kruchten, P., Lago, P., van Vliet, H.: Building Up and Reasoning About Architectural Knowledge. Quality of Software Architectures, Vol. 4214. Springer Berlin / Heidelberg (2006) 43-58
7. Tyree, J., Akerman, A.: Architecture decisions: demystifying architecture. IEEE Software 22 (2005) 19-27
8. IEEE: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE (2000)
9. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley, Boston (2003)
10. Kruchten, P.B.: The 4+1 View Model of architecture. IEEE Software 12 (1995) 42-50
11. Bosch, J.: Design and use of software architectures : adopting and evolving a product-line approach. Addison-Wesley, Reading, MA (2000)
12. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P.: A general model of software architecture design derived from five industrial approaches. Journal of Systems and Software In Press, Corrected Proof (2006)
13. Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering 21 (1995) 314-335

14. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Software Engineering Notes 17 (1992) 40-52
15. Buschmann, F.: Pattern-oriented software architecture: a system of patterns. Wiley, Chichester ; New York (1996)
16. OMG: MDA Guide version 1.0.1. OMG (2003)
17. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE 91 (2003) 145-164
18. Tolvanen, J.P., Kelly, S.: Defining domain-specific modeling languages to automate product derivation: collected experiences. Software Product Lines 9th International Conference, SPLC 2005 Proceedings Lecture Notes in Computer Science, Vol. 3714. Springer (2005) 198-209
19. Greenfield, J., Short, K.: Software factories : assembling applications with patterns, models, frameworks, and tools. Wiley Pub., Indianapolis, IN, USA (2004)
20. Mattsson, A., Lundell, B., Lings, B., Fitzgerald, B.: Experiences from Representing Software Architecture in a Large Industrial Project Using Model Driven Development. Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent. IEEE Computer Society, Minneapolis, USA (2007)
21. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (2000) 70-93
22. Medvidovic, N., Dashofy, E.M., Taylor, R.N.: Moving architectural description from under the technology lamppost. Information and Software Technology 49 (2007) 12-31
23. Medvidovic, N., Rosenblum, D., S., Redmiles, D., F., Robbins Jason, E.: Modeling software architectures in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodologies. 11 (2002) 2-57
24. Mikkonen, T.: Formalizing design patterns. Software Engineering, 1998. Proceedings of the 1998 (20th) International Conference on (1998) 115-124
25. Lauder, A., Kent, S.: Precise Visual Specification of Design Patterns. Proceedings of the 12th European Conference on Object-Oriented Programming. Springer-Verlag (1998)
26. Eden, A.H.: A Theory of Object-Oriented Design. Information Systems Frontiers 4 (2002) 379-391
27. Bayley, I.: Formalising Design Patterns in Predicate Logic. Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on (2007) 25-36
28. Mak, J.K.H., Choy, C.S.T., Lun, D.P.K.: Precise modeling of design patterns in UML. Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on (2004) 252-261
29. Zdun, U., Avgeriou, P.: Modeling architectural patterns using architectural primitives. Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications. ACM, San Diego, CA, USA (2005)
30. Atkinson, C., Kuhne, T.: Model-driven development: A metamodeling foundation. IEEE Software 20 (2003) 36-41
31. Gamma, E.: Design patterns : elements of reusable object-oriented software. Addison-Wesley, Reading, Mass. (1995)