Tool Support for Architecture Stratification

Thomas Kühne, Martin Girschick, Felix Klar

Fachgebiet Metamodellierung
Fachbereich Informatik
Technische Universität Darmstadt
Germany
{kuehne, girschick}@informatik.tu-darmstadt.de
felix@klarentwickelt.de

Abstract: Architecture Stratification is a technique for describing and developing complex software systems on multiple levels of abstractions. In this paper we present an approach and a corresponding implementation—in the form of a Fujaba plugin—for refining models including their behavior. Our plugin enables Fujaba models to be annotated with refinement directives and supports the definition of corresponding refinement transformations with a combination of "Story-Driven-Modeling" and Java code. In this paper we motivate architecture stratification, describe how to define and use transformations, and present a case study.

1 Introduction

Today's large software systems have reached such a level of complexity that a single architectural view is not sufficient anymore to appropriately capture their high-level architecture, detailed design, and low-level realization. If a system is described from a bird's eye view—using a very high-level architecture description—many important details regarding performance, local extensibility, etc. remain hidden. If, however, one chooses a view revealing much more detail so as to allow the above properties to be evaluated, the complexity will become unwieldy and it is then difficult to see the wood for the trees.

Architecture stratification is an approach that connects multiple views on a single system with refinement translations, so that each view completely describes the whole system on a particular level of abstraction. This way, single levels do not only present an optimal mix of overview and detail for various stakeholders, but they also separate and organize a system's extension points, patterns, and cross-cutting concerns [AK03].

We subsequently describe how to use our Fujaba [NNZ00] plugin SPin for architecture stratification (section 2) and present a corresponding case study (section 3). Finally, we address related and future work (sections 4 & 5) and conclude in section 6.

2 Architecture Stratification with SPin

Figure 1 illustrates the basic idea of Architecture Stratification: The linear arrangement of interconnected system descriptions at increasing levels of abstraction. While in principle, both downward- (development) and upward- (re-engineering) generation directions are possible, in this paper we are focusing on the downward direction only.

Note that the vertical dimension of Figure 1 does not represent a time line. All levels coexist and are available for inspection concurrently, at any time. Ultimately, our goals are full traceability and both downward and upward propagation of edits at any level, however, our current technology only supports downward propagation of changes.

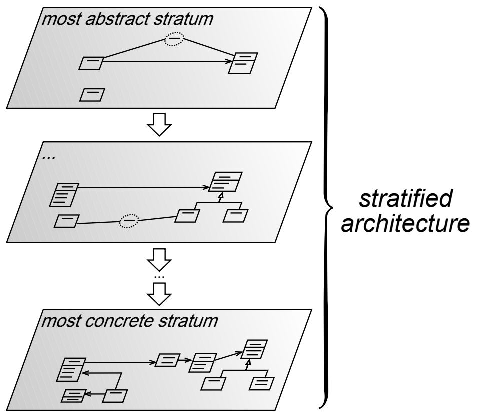

Figure 1: *Architecture Stratification*

Our plugin does not only transform models (e.g., class diagrams) but also any associated code (e.g., method implementations). We can therefore refine a very simple—but fully functional—system at the top level, into a complex one that supports more functionality and exhibits more non-functional properties (e.g., distribution). This most detailed system description can then be used to create an executable system by virtue of the Fujaba code generation engine.

2.1 Refinement Annotations

Which of the model elements in a stratum should be refined into a realization at the next level below, is governed by so-called refinement annotations. In a refinement step these then trigger corresponding transformation rules. As these rules typically need to consider multiple model elements at a time and sometimes even need information that is not present in the model, the annotations feature links to other model elements (e.g., enumerating the observers for a given subject in the context of the Observer pattern [GHJV94]) and can be parameterized using basic types (e.g., a string specifying the name of a class that will be generated).

We chose a notation for refinement annotations similar to UML collaborations occurring in UML class diagrams. Both notations share the need to specify which elements form a structure and what role the referenced elements play. In our case, we need to designate which element(s) should be involved in a single refinement transformation, which element(s) are to be used as a parameter to the transformation, and what their corresponding role is. Compared to stereotypes which are commonly used to guide transformations, our refinement annotations enable much more explicit control and a visual approach to specifying transformation parameters. Our notation is obviously less space efficient than stereotypes, but we support a “collapsed” presentation mode that is visually as non-intrusive as stereotypes.

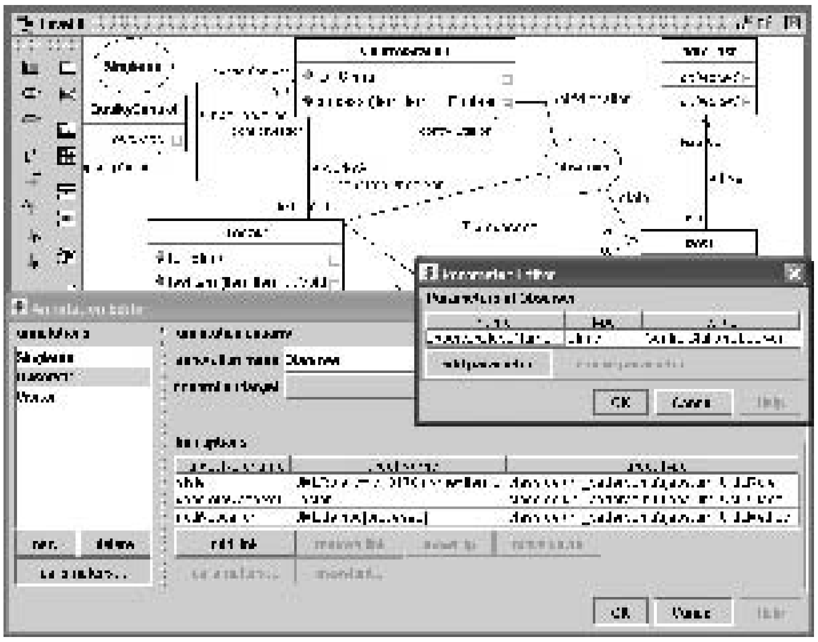

Figure 2: *Editing Refinement Annotations*

SPin provides a dedicated *annotation editor* to support the introduction and parameterization of annotations. Figure 2 shows a screenshot of the annotation editor displaying the parameters of an “Observer” annotation. The annotation is parameterized with one basic type “observerClassName” (see window “Parameter Editor”), designating the name of the to be generated Observer interface, and three links (see window “Annotation Editor”). The “concreteObserver” link picks the element that should play the role of the observer, whereas “state” picks the subject’s attribute, that will provide the “to be observed”-information. Finally “notifyLocation” specifies a method that should notify all observers after a subject’s state changes.

Once a model is completely annotated, the user may use the context menu of an annotation to initiate the corresponding transformation process. SPin, however, also supports the automatic transformation of all annotations of the same kind within a stratum. Further automation, such as a persistent selection of a set of annotations for a stratum and/or the recursive unfolding of strata until the most detailed level has been reached, is planned for future versions of SPin.

2.2 Refinement Rules

Refinement rules, triggered by an unfolding of an annotation, are completely user defined. SPin only provides the machinery for rule authors to create rules and stratum designers to use the rules. The rules themselves are part of a rule library, which can be extended dynamically while the Fujaba environment is being used.

Rules are specified using either Java code or *Story Driven Modeling* (SDM) [NNZ00] diagrams, which results in a maximum of flexibility but also limits the automatic support (e.g., regarding tracebility) the system may provide (see also section 5).

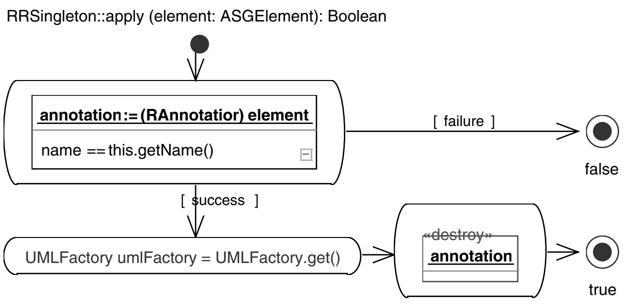
RRSingleton::apply (element: ASGElement): Boolean
annotation := (RAnnotatior) element
name == this.getName()
[failure]
false
[success]
UMLFactory umlFactory = UMLFactory.get()
«destroy»
annotation
true

Figure 3: *Generated `apply`-method*

SPin equips the Fujaba UML class diagram editor with a "create rule"-action, which invokes a "new rule" dialog. After entering the rule name and further data, SPin automatically generates the body of an `apply` method. This method's pattern matching part—the one that checks whether the rule is applicable or not—is specified using Fujaba's SDM capabilities, resulting in a semi-graphical implementation which is more self-explanatory and easier to maintain than Java code. In Figure 3, the first check makes sure that the model element to be transformed indeed has the correct annotation ("Singleton" in this case). If so, a reference to a *UMLFactory* is created, so that the rule author may easily create new UML elements within the core transformation part (not yet specified in Figure 3). Finally the annotation is removed from the diagram, since at this point in time the annotation has served its purpose. The rule author may still change any part of the above matching pattern, but it is provided automatically as a default, since this is how the basic structure of most refinement rules looks like.

In our example, the rule's precondition has to be enhanced to check whether the annotation is bound to a UML class. The transformation code itself then adds an attribute holding the singleton-instance, a private constructor and a get-method that returns the singleton-instance. Once finished, the rule can be exported to the rule library, so that it may be used to transform a UML class into a "singleton".

Note that while our discussion and examples discuss classes and class diagrams only, SPin may be used to transform any element of a model based on Fujabas ASG-metamodel.

3 Case Study

We now demonstrate the utility of SPin by considering an example system that simulates a quality control assembly line. In order to keep the example small, it only involves the application of three design-patterns: “Singleton”, “Observer”, and “Visitor” [GHJV94]. In general, Architecture Stratification is about more than mere “pattern application” [AK03].

3.1 System Description

Figure 4 shows a high-level view on the system’s structure. The system has a main quality control unit (*QualityControl*) that must be accessible as a singleton instance (hence the corresponding annotation). It controls an assembly line that consists of a variable number of control stations (*ControlStation*). These stations check items (abstract class *Item*) passed to them by the assembly line. Our example features one concrete item type only (*Screw*).

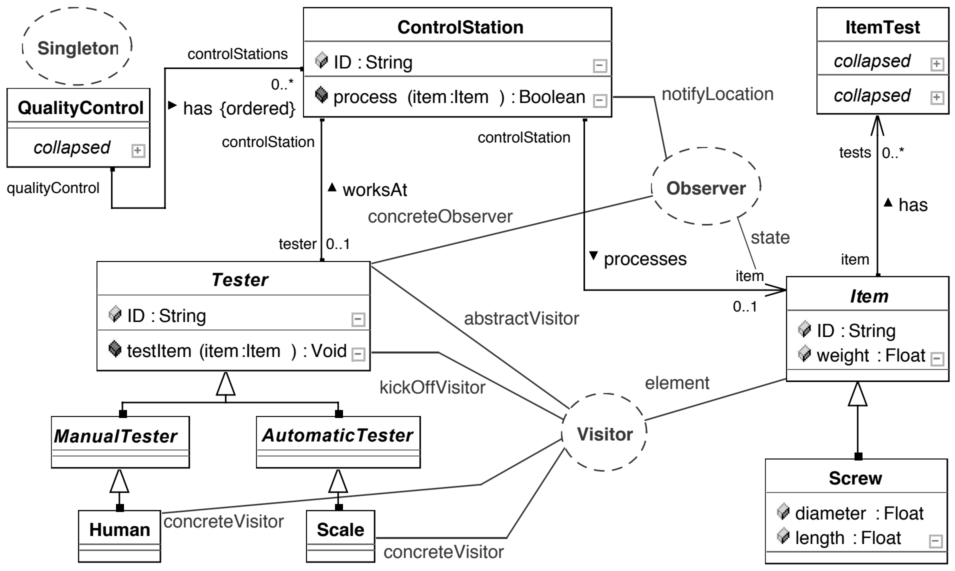

Figure 4: *Case Study: Most Abstract Stratum*

Control stations feature a tester which checks the current item. For each observed item a test report (*ItemTest*) is created. Testers come in two kinds: manual testers (here, *Human*) that are able to perform very complex tests, and automatic testers, e.g., industry-robots that are specialized for testing a single property of an item (here, *Scale*).

Let’s have a closer look at the Visitor-annotation: It features link “element”, specifying which class(es) is/are the element(s) to be visited, and link “abstractVisitor”, specifying, which class is the superclass of all concrete visitors. The latter are referred to by the two "concreteVisitor' links and designate methods that need to receive an implementation in a lower stratum. Additionally link "kickOffVisitor" defines the method that shall invoke the visitor.

3.2 Refining the System

Unfolding the "Singleton" annotation requires no further work—any required additional artifacts, including code snippets, are automatically generated. In contrast, the generated Observer and Visitor realizations are, of course, not complete on their own. For instance, subjects of the Observer pattern (here, *ControlStation*) need to send out notification messages to all their observers. In our example, method `process(item:Item)` of class *ControlStation* is extended with a corresponding `notifyObservers`-call (see link "notifyLocation"). In previous versions of SPin this had to be done manually and the extra code did not survive re-generation steps. A better solution is to provide the extra code as a parameter to the "Observer"-annotation. Due to space constraints we cannot further discuss this alternative and other options.

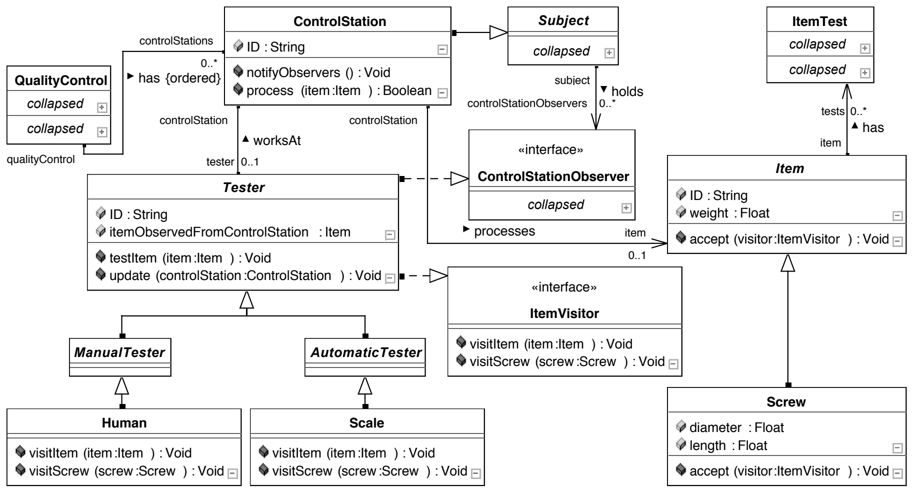

Figure 5: *Case Study: Most Detailed Stratum*

3.3 Generating the System

Now that the most detailed stratum has been generated, Fujaba's codegenerator may be used to generate executable code from it, i.e., convert all model-related constructs, such as associations, etc. into plain Java code and combine it with the code that has been accumulated by all strata refinements.

Note that a complete code generation is possible only because of three properties of our approach: First, the most abstract stratum already contains code that defines the behavior for a simple system at this high level of abstraction. Second, subsequent transformations, affect both model elements and associated code. Third, any code that cannot be automatically created by transformations rules is supplied by the stratum designer at the appropriate stratum, i.e., appropriate level of abstraction. In other words, both the structure and the behavior of the system are modified and extended in a stepwise and synchronized manner.

4 Related Work

Together Architect[1] provides an extendable template-based mechanism for defining patterns, which can then be applied to class diagram elements. In contrast to SPin, however, these transformations are executed in a step by step fashion, whereas SPin automates the transformation of all annotations of one kind and will eventually support a fully automated application of all applicable transformations from top to bottom.

OptimalJ[2] from Compuware is a Java-oriented model driven development environment specialized to generate J2EE applications. Although model-to-model transformations are supported, true multi-level modeling in the style of Architecture Stratification is not available. OptimalJ imposes a rather guided development process on its users, which first have to select a type of application and then have to complete the model templates created by OptimalJ. The transformation process then generates the code and other needed artifacts. This approach is useful, if the needed application types are supported by OptimalJ, but fails if requirements dictate alternative solutions.

The model transformation framework *Mercator* [WJ04], which, similar to SPin, also uses UML class diagrams and corresponding model annotations to control transformations, follows the UML standard for profiles, and hence uses UML stereotypes for annotations. Our notation, similar to UML collaborations, is more expressive, directly indicating all involved elements in a visual fashion.

The "Bidirectional Object-Oriented Transformation Language" (BOTL) [MB03] also uses stereotypes as annotations. The pattern matching process in the source model is similar to ours whereas the generation of elements in the target model is always specified visually. Although this is also possible with SPin using Fujaba's SDM graph transformation scheme, our practice has shown that Java code often enables a more direct and concise definition of transformations. Automatic code generation within BOTL is planned but not implemented yet.

The MDA tool ArcStyler[3] follows the MDA approach where a platform independent model (PIM) is completely parameterized and then transformed to a new platform specific model (PSM). If this approach is used in a staged, incremental manner, it very much resembles the abstraction level stratification approach of SPin. ArcStyler defines transformations us-

[1] `http://www.borland.com/us/products/together/`
[2] `http://www.compuware.com/products/optimalj/`
[3] `http://www.interactive-objects.com/`

ing "cartridges" and UML stereotypes may be used to guide the transformation process. In addition so called *marks* are used to allow further parameterization of the model. Transformations are defined using the script language JPython, which is similar to our Java code definitions, however, less than the SDM capabilities that are available in Fujaba and SPin.

Microsoft's vision for model driven development is based on domain specific languages (DSLs) instead of UML. So called *Software Factories*, detailed in [GS03], are presented as an extension to integrated development environments and add support for DSLs and model transformations. Both techniques share the ability to define customization points. However, Software Factory's so called "variability points" only add to the domain specific behavior of frameworks, which need to (pre-)exist. Software Factories are supposed to support advanced ways of performing multi-level modeling with a grid of models in the future, but many of the details and the implementation status remain unclear.

Czarnecki et al. propose the novel concept of "staged configuration" for feature modeling [CHE04]. This multi-layered modeling approach exhibits some similarities to stratification. The annotations within a stratum can be compared to the features which can be selected in staged configurations. While annotations allow more flexibility, staged configurations are easier to create and use as the features are limited to a defined set and less complex than arbitrary refinement transformations.

Almeida et al. approach system design through multiple levels of abstraction, not dissimilar to Architecture Stratification [ADP+05]. They present a number of "design operations" for describing the transformations between abstraction levels. They, however, are not concerned with an automated transformation process, as the selection of elements plus the invocation of transformations are performed manually.

5 Future Work

The current version of SPin offers a limited set of transformation rules only. Although these are user extensible, the utility of SPin would be increased if it already came with a rich set of ready-to-use rules. We plan to apply Architecture Stratification to much bigger and more complex examples and correspondingly expect to identify and implement a richer library of SPin transformation rules.

Employing stratification in its intended form with SPin is currently hindered by the fact that only manual, stepwise initiations of transformations are supported. In order to fully automate the generation of a complex system from a simple system, it is necessary to automate the process of unfolding annotations. This also includes the specification of the order in which annotations are to be unfolded. This ordering, however, is neither difficult to work out, nor should it be part of an automated process. Annotations exhibit natural dependencies and lend themselves to generate levels of system concerns [AK03]. It is therefore the task of the system architect to select which of the annotations are addressed at each specific abstraction level. As a result, future versions of SPin should provide a configuration system, allowing users to specify and store their annotation processing orders.

The current release of SPin specifies transformation rules with imperative instructions, including unconstrained Java code. This implies that there is no easy way to automate traceability, e.g., for forward updates or backward-navigation. We are therefore investigating the usage of graph rewriting approaches [Kön05], e.g., Triple Graph Grammars [Sch94]. In addition to providing a way to automatically maintain consistency links, such bi-directional transformation rules would also represent an attractive facility for reverse engineering, i.e., starting from a complex system and simplifying the system by either using refinement rules in the "reverse" direction and/or creating and applying dedicated "abstraction rules".

SPin will significantly benefit from the new features of Fujaba 5. For instance, the then available support for multiple projects will enable developers to create rules in one project and immediately apply them in another. Moreover, users will then be able to more easily navigate back and forth between different strata.

6 Conclusion

In this paper we have presented the Fujaba plugin SPin as developed at the department for "Metamodeling and its Application" at the Darmstadt University of Technology. We have documented our progress in providing tool support for Architecture Stratification, presenting a prototype—drawing on Fujaba features such as *Story-Driven-Modeling*—that provides promising development prospects. Since SPin is able to dynamically integrate new rules, the development of the main system model and corresponding rules, can proceed in an interleaved and very interactive manner.

Of particular value is our approach of transforming both model elements and associated code in sync with each other. We can thus obtain a *fully* specified, complex system by starting from a simple system and applying a succession of refinement steps. Refinement rules are user-definable, typically using a convenient mix of SDM (for pattern matching) and Java (for an unconstrained definition of transformations). Their usage is indicated by using a concise—collaboration-like—notation for refinement annotations that enables transformation parameters to be specified both visually (through labeled links to any modeling element, including attributes and methods) and non-visually (through primitive parameter types entered into corresponding dialogs).

Despite the limitations of the current Fujaba version, i.e., the lack of support for multiple projects (strata) and, consequently, missing support for maintaining consistency between model contents (strata elements), we have managed to draw on its fine parts, e.g., *Story-Driven-Modeling* for pattern matching and an adaptable UML metamodel for supporting refinement annotations, to create a prototype supporting Architecture Stratification.

References

[ADP$^+$05] João Paulo Almeida, Remco Dijkman, Luís Ferreira Pires, Dick Quartel, and Marten van Sinderen. Abstract Interactions and Interaction Refinement in Model-Driven Design. In *Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)*, pages 273–286, Twente, Netherlands, September, 19-23 2005.

[AK03] Colin Atkinson and Thomas Kühne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.

[CHE04] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In Robert Nord, editor, *Proceedings of the Third Software Product-Line Conference*, Lecture Notes in Computer Science. Springer-Verlag, September 2004. (Note: this paper is superseded by the extended journal version Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models for Software Variability: Process and Management).

[GHJV94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture*. Addison-Wesley, 1994.

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2003. Addison-Wesley.

[Kön05] A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice, Satellite Workshop of MODELS 2005, Montego Bay, Jamaica*, 2005.

[MB03] Frank Marschall and Peter Braun. Model Transformations for the MDA with BOTL. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, CTIT Technical Report TR-CTIT-03-27*, University of Twente, June 2003.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA Environment. Technical report, Computer Science Department, University of Paderborn, 2000.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 141–163, London, UK, June 1994. Springer Verlag. Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science.

[WJ04] Weerasak Witthawaskul and Ralph Johnson. An Object Oriented Model Transformer Framework based on Stereotypes. In *3rd Workshop in Software Model Engineering at The Seventh International Conference on the Unified Modeling Language, UML 2004*, Lisbon, Portugal, October 10-15 2004.