Tool Support for Architecture Stratification

Thomas Kühne (kuehne@informatik.tu-darmstadt.de)
Martin Girschick (girschick@informatik.tu-darmstadt.de)
Felix Klar (felix@klarentwickelt.de)

Fachgebiet Metamodellierung
Fachbereich Informatik
Technische Universität Darmstadt, Germany

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 coding. The transformations affect both model and associated code and provide for multi-level editing by allowing additive modifications at lower levels that are sustained on re-generation. In this paper we motivate architecture stratification, describe how to define and use transformations, present a case study, and discuss design alternatives for supporting multi-level editing.

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 forest 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].

In earlier work we developed and described the Fujaba [NNZ00] plugin SPin (→ **S**tratification **P**lug**in**) [KKG05] which supports the automatic transformation of models into more detailed versions and thus represents basic support for Architecture Stratification. In this paper we build on this work by adding a discussion and an (implemented) proposal for multi-level editing. We subsequently describe how to use SPin for architecture strati-

fication (section 2) and present a corresponding case study (section 3). We then discuss its extension to multi-level editing (section 4). Finally, we address related and future work (sections 5 & 6) and conclude in section 7.

2 Architecture Stratification with SPin

Figure 1 illustrates the basic idea of Architecture Stratification: A linear hierarchy of connected levels of abstraction. While in principle, both downward- (development) and upward- (re-engineering) generation directions are possible (and are supported by SPin), we currently focus on the downward direction. Note, however, 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, but in this paper we restrict ourselves to downward propagation of changes.

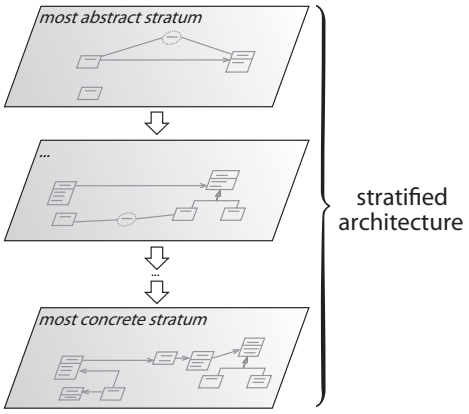

Figure 1: *Architecture Stratification*

Our SPin 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 either using basic types (e.g., with an integer specifying how many times an element should be replicated to support redundancy, or a string specifying the name of a class that should be generated).

We therefore 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.

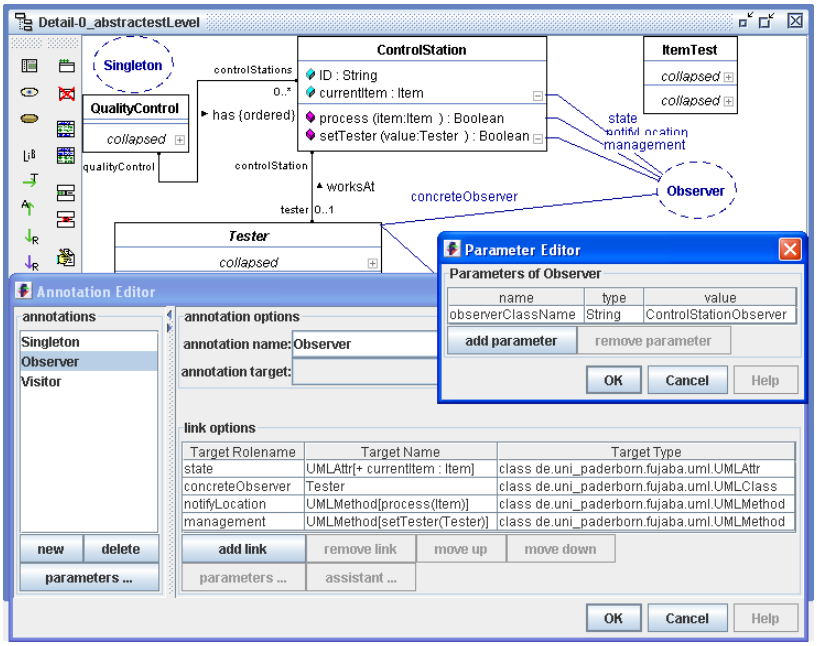

Figure 2: *Editing Refinement Annotations*

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 (also see the example in section 3). 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"): Link "concreteObserver" 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

The rules, triggered by an unfolding of an annotation, are completely user defined. SPin only provides the machinery for creating, using, and executing rules. The rules themselves are part of a rule library, which can be extended dynamically, i.e. while the Fujaba environment is being used. Currently, the rules have to be 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 6).

SPin equips the UML class diagram editor with a "create rule"-action, which invokes a "new rule" dialog. After entering the rules 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 capablities, resulting in a semi-graphical implementation which is more self-explanatory and easier to create and 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, since this is how most refinement rules look like—without their core transformation part and any further checks as to whether the rule is really applicable.

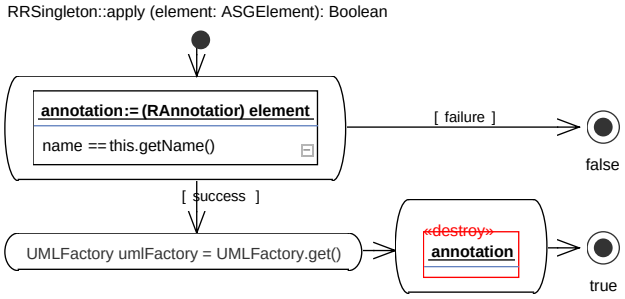

Figure 3: *Generated `apply`-method*

Indeed, 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]—even though Architecture Stratification is in general about much more than mere "pattern application". Figure 4 shows the high-level view on the system's structure.

3.1 System Description

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 (class *ControlStation*). These stations check items (abstract class *Item*) passed to them by the assembly line. Our example features one concrete item type only (class *Screw*).

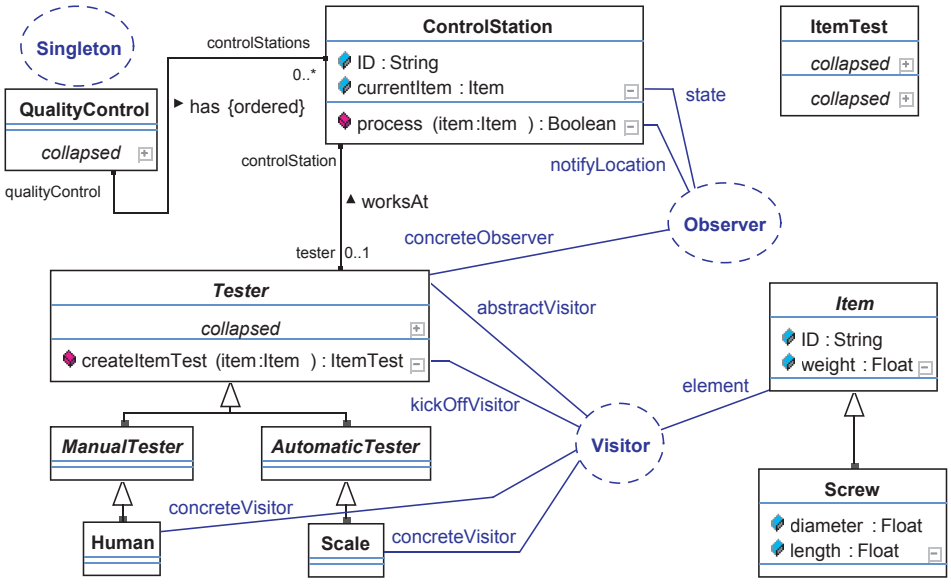

Figure 4: *Case Study: Most Abstract Stratum*

Control stations feature a tester which checks the current item. For each observed item a test report (class *ItemTest*) is created. Testers come in two kinds: manual testers, like humans, 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, we use a robot (class *Scale*) that checks an item's weight.

Let's have a closer look at the Visitor-annotation (as the Observer-annotation has already been described in section 2.1): It features link "element", specifying which class(es) are the element(s) (to be visited) of the Visitor pattern, 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 (see section 4.2). 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. We have never implemented the smarter approach of providing the extra code as a parameter to the "Observer"-annotation, since we now have an even better solution (see section 4).

Similar arguments apply to the Visitor realization (see the resulting system structure in Figure 5). All `visitXZY(...)` methods now need corresponding implementations.

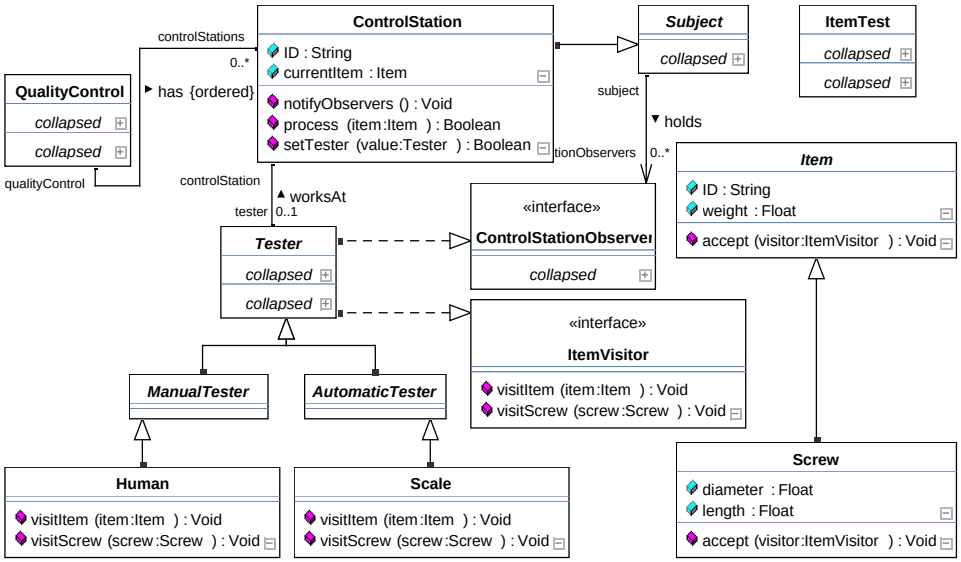

Figure 5: *Case Study: Most Detailed Stratum*

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.

However, we are still missing some method bodies and method body fragments. The various ways in which these may be provided and a description of our implementation are the subject of the next section.

4 Multi-Level Editing

According to Architecture Stratification, the most abstract stratum already contains a simplified system description including its behavior (in our example expressed through method bodies only). Figure 6 shows such parts of the system description using the color orange at the top stratum.

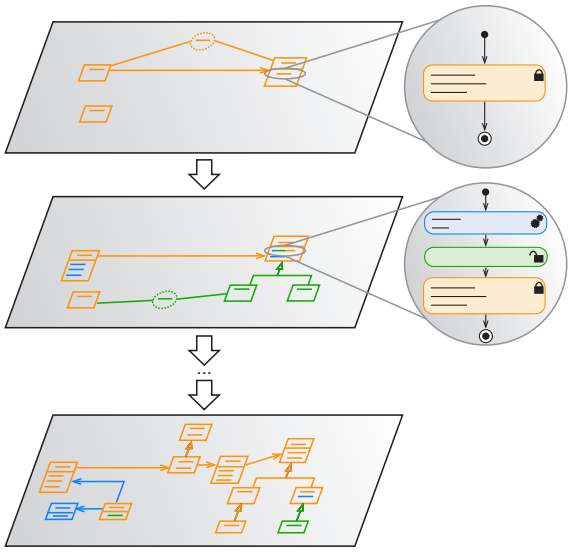

Figure 6: *Model and Code Transformation*

As the system is refined in lower strata, however, additional classes and code have to be provided. Some of this code can be generated by the refinement rules directly (e.g., the Singleton implementation). Figure 6 shows such parts in blue. Note that generated elements (blue) at one stratum will be treated as existing elements (orange) at the stratum below.

The remaining category of elements (shown in green in Figure 6) relates to elements which have to be introduced at this stratum. A simple example for this category are the `visitXYZ(...)` methods (see classes *Human* and *Scale* in Figure 5). These are relatively easy to deal with, since they are uni-colored green, i.e., they contain new code only, rather than a mixture of old (orange), generated (blue), and new code (green).

4.1 Preserving New Parts

Even with a straightforward case as the Visitor methods, though, one still needs to take measures to prevent the new methods from being overridden upon re-generation steps. If a new unfolding of the "Visitor"-annotation (e.g., since the original system or the refinement translation was changed) creates new `visitXYZ(...)` stubs, we do not want to loose the existing method bodies. Three main strategies exist for dealing with this problem.

4.1.1 Free Editing

Any change in a stratum is possible and supported by making all such edits persistent. If a re-generation occurs, its resulting elements have to just change those (orange & blue) parts in a stratum, which are controlled by the stratum above.

This approach has a certain appeal as it allows full control at each stratum without loosing edits upon re-generation. This, of course, only works if one does not allow editing of orange parts, or finds a strategy to communicate any operations on orange parts, e.g., split-ups with intermediate green parts, to the stratum above, so that it can guide re-generation results accordingly.

Despite the fact that this strategy would have been very difficult to implement with the current Fujaba version (which does not support multiple projects (strata) at a time and is not yet well-equipped to support consistency updates between strata), there is a another good reason against such an "anarchistic" approach to stratum modification: If any of the refinement transformations have to be changed, e.g., since a supporting technology, such as a certain middleware solution, they assume has changed, the "free edits" in all strata below this point are potentially subject to change. Even with a good traceability mechanism in place, which will be able to locate all such parts, and an interactive scheme, allowing one to adapt the "free edits" to the new situation, one still faces a maintenance challenge. As there is no way to restrict the edits, these may aggravate the maintenance challenge by exhibiting more dependencies on provided elements in their stratum then strictly necessary.

4.1.2 No Editing

A solution to the problem outlined above, is to disallow any editing in a stratum (except the top stratum) and to provide any extra parts in parameters of refinement annotations. This ensures full top-down re-generation without any danger of loosing extra (green) parts. However, this implies that for instance the `visitXYZ(...)` methods need to be written as code snippets supplied to the "Visitor"-annotation. This is not only artificial, as the code cannot be written in its natural context, but also gets more difficult when dealing with a mixture of existing, generated, and extra code.

4.1.3 Constrained Editing

For the latest incarnation of SPin we, therefore, chose a compromise and restrict editing to a few, well-known parts in a stratum. These parts are identified by the refinement transformations and, possibly, by the stratum above. Consider Figure 7, which shows how we exploit Fujaba's SDM feature that enables users to provide code (or alternative ways of describing behavior) within blocks of activity diagrams. The left part of Figure 7 shows a method body that has been split into two blocks on purpose (see the next paragraph below). The right part shows the result after a certain refinement translation, which inserted a number of new code bits (blocks *generated.1* & *generated.2*) and supplied block *generated.2* with two user definable pre- and post code blocks. This way, a designer may insert any appropriate code at this stratum and can be sure that these additions will not be lost upon re-generation. Any extra (green) blocks are saved in addition to the standard Fujaba model and are retrieved once a re-generation occurs.

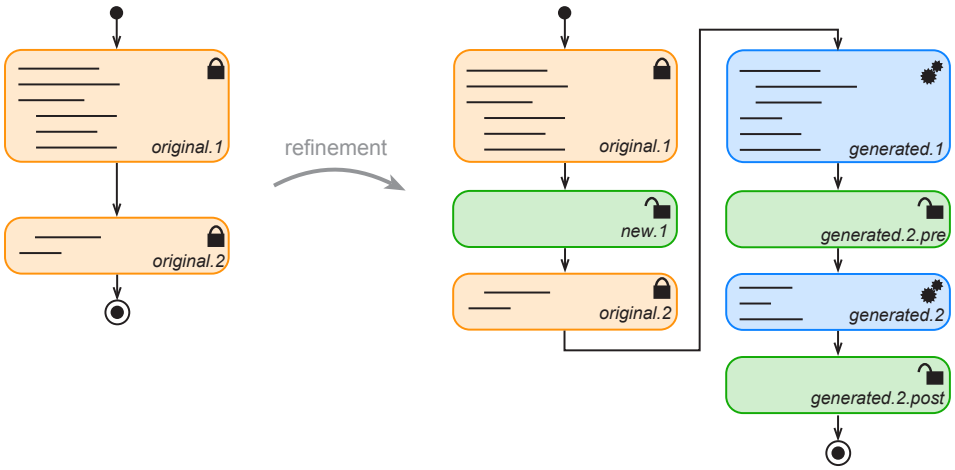

Figure 7: *Code Block Categories*

The green *new.1* block has been inserted by the refinement step not because the refinement rule author has foreseen the need to provide extra code (as with the pre- & post-blocks), but since the original method at the stratum above contained a block transition between *original.1* and *original.2*. This way, a stratum designer can induce the creation of free-editing blocks and, hence, add to the ones already created by the refinement rule.

Since the green parts designate areas of variability (as "*hot spots* in frameworks") and fill-in parts as predetermined by the stratum above (as "*hook*-methods" in the design pattern "Template Method") we call them *hook spots*. Note that while our implementation currently supports such green parts for code fragments only, they are in general also applicable for any other modeling element types, e.g., classes and associations.

4.2 Completing the Case Study

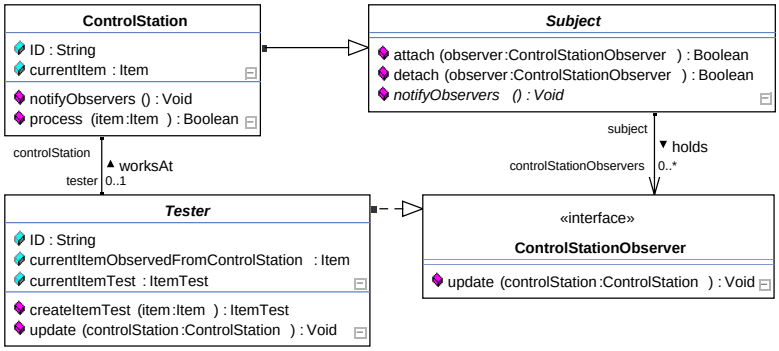

Figure 8: *Observer Refined*

We can now describe how to use hook spots to complete the case study from section 3. Of particular interest is the `update`-method of *Tester* (see Figure 8), since it contains a mixture of generated and custom extra code. This method needs to react to notification messages from subject *ControlStation*, i.e., test an item whenever it has arrived at a control station. Figure 9 shows the generated `update` method, featuring a first part which a) makes sure that the subject can be accessed, b) that indeed a change in the control station occurred, and c) sets up two variables for further use. The second part then has to be implemented by a stratum designer and will be retained upon re-generation.

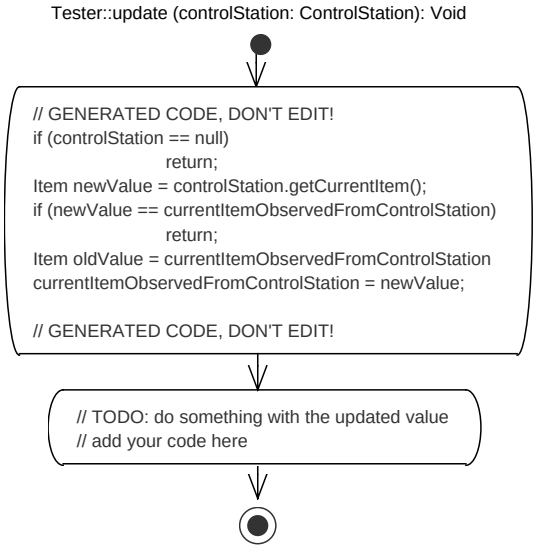

Figure 9: *Observer Hook Spot*

A similar hook spot is used for notifying all observers of a state change. The refined visitor pattern supports hook spots in all `visitXYZ(...)`-methods of visitor classes, marked with “concreteVisitor”, and in the method marked with “kickOffVisitor”.

5 Related Work

Most tools for model-driven development specialize on generating code from a model or migrating models in one modeling language to another, i.e., exogenous transformations [MCG05]. Tools that support model refactorings can—according to [MCG05]—be classified as supporting *horizontal* endogenous transformations, whereas architecture stratification realizes *vertical* endogenous transformations. In other words, refactorings maintain the same level of abstraction whereas architecture stratification creates different levels of abstraction expressed in the same modeling language. Only few commercial tools, provide basic support for defining vertical endogenous transformations as well.

Together Architect[1] provides an extendable template-based mechanism for creating pattern applications. This mechanism can also be used to create rules that perform vertical endogenous transformations. Together Architect uses a pattern manager to apply patterns 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. Together Architect follows a purely generative approach and hence neither supports SPin's re-generation facility nor its *hook spot* approach to protect user edits from being overridden.

OptimalJ[2] from Compuware is a Java-oriented model driven development environment specialized to generate J2EE applications. Similar to SPin, it supports adding editable regions (so called "free blocks") in the source code which are retained upon re-generation. Generated source code fragments are automatically locked and cannot be edited. 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. In contrast to SPin, BOTL does not offer true support for multi-level modeling. The transformation results are always merged with the destination

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

model which may lead to unexpected results. 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 using "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. ArcStyler features protected code fragments, similar to SPin's code blocks. However, the latter offer more flexibility by supporting a mixture of existing, generated, and extension code fragments within the same method.

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 (our hook spots). However, Software Factory's so called "variability points" only add to the domain specific behavior of frameworks, which need to (pre-)exist. Software Factories also announce advanced ways of performing multi-level modeling with a grid of models, 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—selection of elements plus invocation of transformations are done manually—and have no mechanism similar to hook spots.

None of the above mentioned tools/approaches support Fujaba's *Story-Driven-Modeling* feature [FNTZ99], which is not only very useful for the semi-graphical specification of transformation rules as usable in SPin, but also provided us with the basis for creating hook spots that survive re-generation steps.

[3] `http://www.interactive-objects.com/`

6 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 free 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.

7 Conclusion

In this paper we have presented SPin and its latest extensions as developed at the department for "Metamodeling and its Application" at the Darmstadt University of Technology. We have documented our further progress in providing tool support for Architecture Stratification, presenting a prototype—drawing on Fujaba niceties such as *Story-Driven-Modeling*—that provides fascinating 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 start-

ing 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).

We have presented a useful compromise for multi-level editing, ranging between anarchistic "free-editing" and very restraining "no editing", based on the concept of *hook spots* which enable controlled amendments to both model elements and code. The combination of rule-defined and stratum designer ineducable hook spots provides a scheme that is a) implementable within the current limitations of Fujaba and b) more than sufficient for providing extra information at lower strata. We have thus overcome SPin's former weakness of requiring the user to perform manual edits to the generated models, leading to their loss upon strata-re-generation.

Despite the limitations of the current Fujaba version, i.e., no support for multiple projects (strata) and no mature support for maintaining consistency between models (strata elements), we have managed to draw on its fine parts, e.g., *Story-Driven-Modeling* for pattern matching and hook-spot codeblocks, to create a prototype supporting Architecture Stratification.

While we believe that the combination of Fujaba and SPin already constitutes a rather fascinating tool, we are convinced that further work will result in an even better demonstration of the potential of 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).

[FNTZ99] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language and Java. Technical report, AG-Softwaretechnik, Fachbereich 17, Universität Paderborn, 1999.

[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.

[KKG05] Felix Klar, Thomas Kühne, and Martin Girschick. SPin – A Fujaba Plugin for Architecture Stratification. In Holger Giese and Albert Zündorf, editors, *3rd Int. Fujaba Days 2005: "MDD in Practice"*, pages 17 – 23, September 15-18 2005.

[Kön05] A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Workshop*, 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.

[MCG05] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A Taxonomy of Model Transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[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 20^{th} 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.