# Generating Systems from Multiple Levels of Abstraction

Martin Girschick, Thomas Kühne, and Felix Klar

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

**Abstract.** We describe our prototype implementation for Architecture Stratification supporting system descriptions at multiple levels of abstraction for developing complex software systems. Our tool transforms both model and code fragments in parallel using refinement transformations which are specified with a combination of "Story-Driven-Modeling" and Java code. Multi-level editing is enabled by allowing additive modifications at lower abstraction levels that are retained on re-generation. We present a case study illustrating the application of a number of design patterns and show how our approach can be used to tie in a generic framework by automatically generating the corresponding glue code.

## 1 Introduction

Large and complex systems cannot be adequately captured with a single description only. If the level of abstraction is high—so that an overall architecture can be recognized—too little can be said about important details of the system. If the level of abstraction is low—so that these details can be examined—it is difficult to see high-level structures among the low-level details.

Architecture Stratification [3], therefore, uses multiple system descriptions at the same time, each fully specifying the system at a given level of abstraction. Thus, single levels do not only present an optimal mix of overview and detail for various stakeholders, but also separate and organize a system's extension points, patterns, and cross-cutting concerns.

In this paper we briefly introduce the stratification approach and our current implementation, a plugin for the Fujaba CASE tool (section 2). We then demonstrate the utility of our current implementation by means of a small case study (section 3). Subsequently, we discuss issues of enabling multi-level editing in a stratified architecture (section 4) and, finally, address related and future work (sections 5 & 6) before concluding in section 7.

## 2 Architecture Stratification

Most model-driven approaches and supporting tools interpret the transformation from a source model to executable code as a monolithic step. Only a few tools actually support this transformation as a series of model-to-model transformations (see section 5).

In [2] Atkinson and Kühne introduce the concept of Architecture Stratification in which a system is described on several levels of abstractions. Each level (called a stratum) introduces additional concepts to a software system until the most concrete stratum describes the final system. Note that each stratum describes the entire system, with respect to the level of abstraction it addresses. Conceptually, editing of all strata is possible at all times and changes in one stratum are propagated both upwards and downwards along corresponding refinement relations within the stratified architecture. Our current research, however, focuses on the downward propagation using model transformations.

While we are currently describing systems with UML class diagrams plus associated code, this does not exclude other forms of behavior specification or the use of domain-specific languages. One can easily imagine a gradual transformation of a system description in a domain-specific language at the top-level stratum into a specification expressed in a standard language (such as the UML) for which standard code generation techniques exist.

## 2.1 Refinement Annotations

We guide, and in fact trigger, model transformations by annotating models with so called "refinement annotations" which are linked to corresponding "refinement transformations". As these transformations typically need to consider multiple model elements at a time and sometimes even need information that is not present in the model, the corresponding annotations feature "links" to other model elements (e.g., enumerating the observers for a given subject in the context of the Observer pattern [6]), and can be further 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 used within 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 use a visual approach to specifying transformation parameters. Obviously, our notation is less space efficient than stereotypes, but we support a "collapsed" presentation mode that is visually as non-intrusive as stereotypes.

Our prototype tool supporting Architecture Stratification has been implemented as a plugin—called SPin (Stratification PlugIN)— for the CASE tool Fujaba [11]. It supports the introduction and parameterization of annotations using an annotation editor. Once a model is completely annotated, the user may use the context menu of an annotation to initiate the corresponding transformation process. Further automation steps, such as a persistent selection of a set of annotations to be unfolded for a particular stratum and the recursive unfolding of strata until the most detailed level has been reached, are not yet implemented but will be supported in future versions of SPin.

### 2.2 Refinement Transformations

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

Transformations may be specified using a visual *Story Driven Modeling* (SDM) [11] approach based on graph transformations and/or Java code, which in combination results in a maximum of flexibility but also limits the automatic support—e.g., regarding tracebility—the system may currently provide (see also section 6).

In order to facilitate the creation of new transformation rules, SPin provides wizard-like functionality, automatically creating necessary elements for the transformation specification: After entering the transformation name and further data, SPin generates the body of an SDM diagram, which is responsible for checking the applicability of the transformation rule. This check is specified using Fujaba's SDM graphical notation for pattern matching, which is more self-explanatory and easier to maintain than Java code.

In addition to the graphical notation, however, SDM diagrams can also contain source code fragments (see the next section for an example). Of course, the transformation rule author is not only able to fill in the main transformation part, but also free to completely change and amend the generated transformation specification template. Once the transformation rule has been completed, it can be exported to the transformation rule library.

## 3 Case Study

In the following we demonstrate Architecture Stratification and our corresponding prototype tool by applying three well-known design patterns [6] and by generating "glue" code, tying a general visualization framework into a main application in order to add automatic visualization support.

### 3.1 System Description

Our case study concerns the simulation of the quality control aspects of an assembly line. The most abstract stratum is shown in Figure 1 and represents 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`). The method `process` checks a given item passed to it by the single instance of class `QualityControl`. Items are represented by the abstract class `Item` and the concrete classes `Nut` and `Screw`. Adding a new item at the beginning of the assembly line shifts the existing items to the next control station (`QualityControl.processOne`).

Each control station is attached to a tester which checks the current item. For each tested 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` and `Extensometer`). The purpose of class `ItemFactory` is to create objects of type Screw and Nut in a random fashion, in order to test the quality control features. All mentioned methods already contain Java code which represents the application specific functionality for this (comparatively high) level of abstraction.

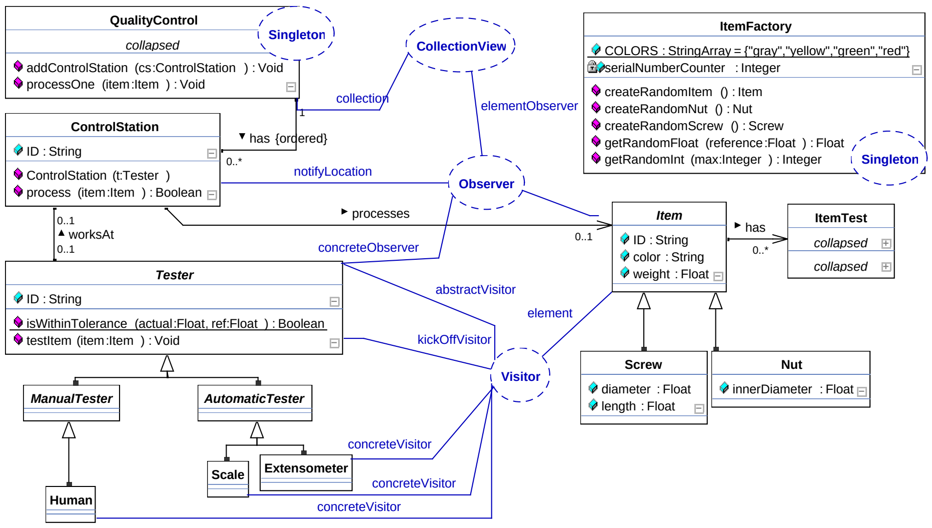


**Fig. 1.** Case Study: Most Abstract Stratum

## 3.2 Annotations

Both `QualityControl` and `ItemFactory` are annotated with a *Singleton* refinement annotation to make sure that only one instance of these classes exists respectively.

The *Visitor* annotation defines which product hierarchy (see the link *element* to `Item`) is inspected by which measurement facilities (the leafs of the `Tester` hierarchy, see the link *abstractVisitor*). The three *concreteVisitor* links designate the methods that need to receive an implementation in a stratum lower down the hierarchy. Finally, link *kickOffVisitor* defines the method that shall invoke the visitor.

The *Observer* annotation defines `Tester` to be an observer of `ControlStation`. Link *notifyLocation* defines the method which triggers update notifications to attached observers, whereas *state* refers to the state to be observed.

Annotation *CollectionView* is special in the sense that it refers to another annotation (see section 3.4) and in the sense that it does not represent a standard pattern application but is responsible for attaching a generic visualization framework to the system. This way, we obtain a rather fancy visualization of our sample system (almost) for free.

### 3.3 Generic Visualization Framework

Attaching a framework, which supports some subsystem functionality, to an application usually involves creating "glue code" which somehow connects the application to the framework. We have created a simple visualization framework in order to demonstrate how to use Architecture Stratification for integrating such supporting frameworks. Our sample framework is specialized to visualize collections of objects. The observer pattern is used to observe changes on this collection. When a new object is added to the collection, its visual representation is also added to the visualization and a new observer instance is created in order to watch for changes within the added object so that they can be visualized accordingly.

There is no need to specify which attributes exist within the visualized object. Our framework uses (Java-) introspection[1] to determine all attributes and their associated values and renders them in a table format. In addition, the class name is used to load a representing icon, if available, which is shown within the visualization. If an attribute named "color" is found, the icon also reflects the current "color" of the object.

By using introspection we offer a simple, generic solution to visualize object attributes. Note, however, that this technique has two disadvantages: First, introspection is not very efficient and may slow down an application considerably. Second, more complex data structures, e.g., with nested objects, cannot be rendered adequately by just using generic strategies.

Yet, these shortcomings are a phenomenon of the particular approach used for the visualization framework. All it takes to address them is an alternative approach to tying in a less autonomous version of the framework into the application. Instead of using a set of refinement annotations (*CollectionView* being one of them), which straightforwardly map the application to the framework which then uses introspection, it is of course possible to generate specialized code that specifically renders some object properties and others not. We are currently working on parameterized refinement transformations to better control the visualized information and make it more efficient at the same time.

### 3.4 Refining the System

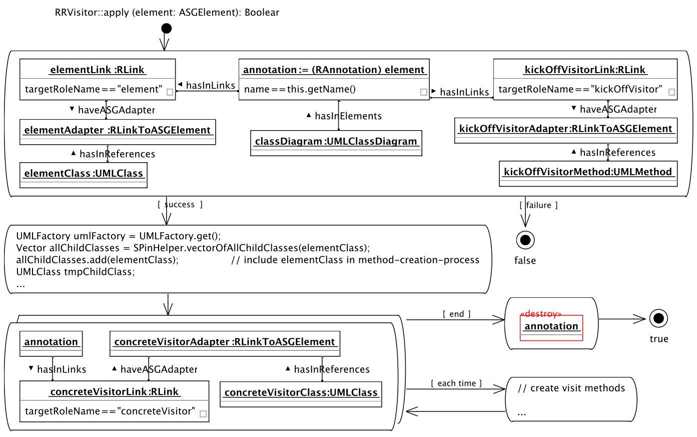
There are several alternatives to start refining the most abstract stratum of Figure 1, but let us begin by unfolding annotation *CollectionView*. The associated refinement transformation first creates a new *Observer* annotation. In the final system it is responsible for updating the visualization when new objects are added to the collection. Based on its *collection* link the "Collection View" annotation unfolds to a new *Observer* annotation with three links:

[1] Incorrectly, referred to as "reflection" by the corresponding Java API.

– *notifyLocation*, referencing the `addControlStation` method within class `QualityControl`.
– *state*, referencing a freshly created association between `QualityControl` and `ControlStation`, named `lastItemOfControlStations`.
– *concreteObserver*, referencing the `CollectionVisualizer` class, which is part of the already existing visualization framework and is therefore included using a "reference" stereotype.

Subsequently, the "Collection View" transformation checks for an *elementObserver* link, referring to an *Observer* annotation. If this exists (as in our example), the transformation adds a second *concreteObserver* link to it, linking it to the class `ElementVisualizer`, which is also part of the visualization framework. If no link named *elementObserver* is found, the transformation will search for links *state* and *notifyLocation* which are used to parameterize a new *Observer* annotation. The link *concreteObserver* will again refer to the class `ElementVisualizer`.

Finally, class `QualityControlVisualizerStartup` is created so that it may set up the visualization framework and attach it to the quality control system.

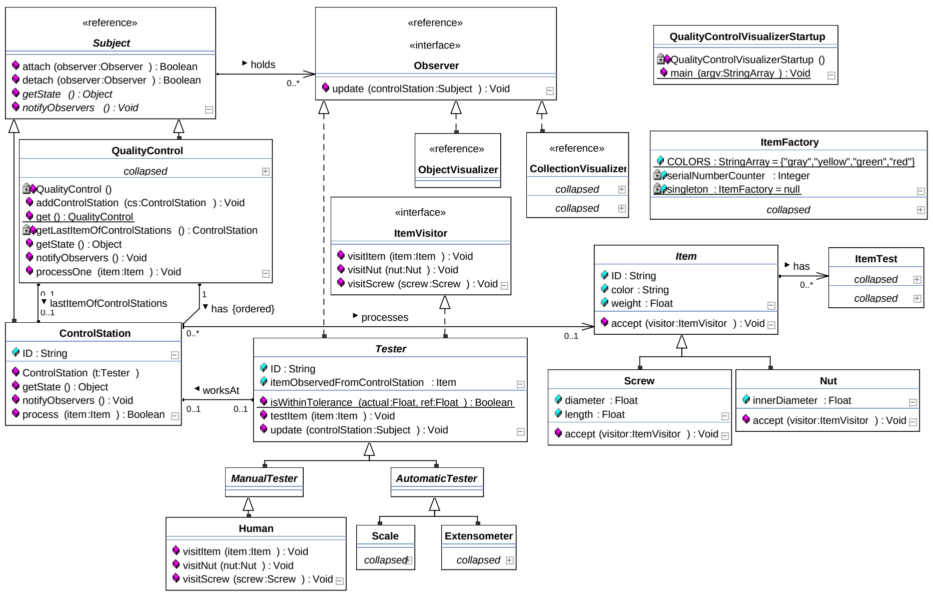

RRVisitor::apply (element: ASGElement): Boolean
elementLink :RLink
targetRoleName=="element"
hasInLinks
annotation := (RAnnotation) element
name==this.getName()
hasInLinks
kickOffVisitorLink:RLink
targetRoleName=="kickOffVisitor"
haveASGAdapter
elementAdapter :RLinkToASGElement
hasInElements
classDiagram :UMLClassDiagram
haveASGAdapter
kickOffVisitorAdapter:RLinkToASGElement
hasInReferences
elementClass :UMLClass
hasInReferences
kickOffVisitorMethod:UMLMethod
[ success ]
[ failure ]
false
[ end ]
«destroy»
annotation
true
annotation
concreteVisitorAdapter :RLinkToASGElement
hasInLinks
haveASGAdapter
hasInReferences
concreteVisitorLink :RLink
targetRoleName=="concreteVisitor"
concreteVisitorClass:UMLClass
[ each time ]
// create visit methods
...

```
UMLFactory umlFactory = UMLFactory.get();
Vector allChildClasses = SPinHelper.vectorOfAllChildClasses(elementClass);
allChildClasses.add(elementClass);            // include elementClass in method-creation-process
UMLClass tmpChildClass;
...
```

**Fig. 2.** Excerpt of Visitor Refinement Transformation

In addition to the two visible annotation links, shown in Figure 1, annotation *CollectionView* contains three string parameters, which may only be inspected with the annotation editor. They define the labels of three buttons, which are shown below the visualization. The refinement transformation generates the respective code fragments, which create the buttons and add them to the visualization window. A stratum designer will eventually also have to provide appropriate behaviour for each button into the generated code. Following our philosophy to enable editing on all strata, this additional code is maintained so that it is retained on re-generation, i.e., re-application of the *CollectionView* transformation rule. We will elaborate on this functionality in section 4.

What remains to be done, in order to completely refine the system description to its most detailed level, is to unfold both *Singleton* and the *Visitor* annotation. The latter is associated with a refinement transformation that nicely illustrates how transformations involving iteration over model elements can still rather concisely be captured using Fujaba's Story driven modelling (see Figure 2). The transformation's first activity matches the annotation links *element* and *kickOffVisitor*, then the necessary model elements are created (omitted for brevity from Figure 2). The next two activities create the `visit...` methods: The bottom left-hand side activity iterates over all *concreteVisitor* links, executing for each the bottom right hand side activity (of which we will see a detailed version in section 4). After all `visit...` methods have been created, the annotation is removed from the diagram, resulting in the final system structure as shown in Figure 3.



«reference»
Subject
attach (observer:Observer ) : Boolean
detach (observer:Observer ) : Boolean
getState () : Object
notifyObservers () : Void
holds
0..*
«reference»
«interface»
Observer
update (controlStation:Subject ) : Void
QualityControlVisualizerStartup
QualityControlVisualizerStartup ()
main (argv:StringArray ) : Void
QualityControl
collapsed
QualityControl ()
addControlStation (cs:ControlStation ) : Void
get () : QualityControl
getLastItemOfControlStations () : ControlStation
getState () : Object
notifyObservers () : Void
processOne (item:Item ) : Void
«reference»
ObjectVisualizer
«reference»
CollectionVisualizer
collapsed
collapsed
ItemFactory
COLORS : StringArray = {"gray","yellow","green","red"}
serialNumberCounter : Integer
singleton : ItemFactory = null
collapsed
«interface»
ItemVisitor
visitItem (item:Item ) : Void
visitNut (nut:Nut ) : Void
visitScrew (screw:Screw ) : Void
Item
ID : String
color : String
weight : Float
accept (visitor:ItemVisitor ) : Void
has
0..*
ItemTest
collapsed
collapsed
0..1
lastItemOfControlStations
0..1
1
has {ordered}
processes
0..*
0..1
ControlStation
ID : String
ControlStation (t:Tester )
getState () : Object
notifyObservers () : Void
process (item:Item ) : Boolean
worksAt
0..1
0..1
Tester
ID : String
itemObservedFromControlStation : Item
isWithinTolerance (actual:Float, ref:Float ) : Boolean
testItem (item:Item ) : Void
update (controlStation:Subject ) : Void
Screw
diameter : Float
length : Float
accept (visitor:ItemVisitor ) : Void
Nut
innerDiameter : Float
accept (visitor:ItemVisitor ) : Void
ManualTester
AutomaticTester
Human
visitItem (item:Item ) : Void
visitNut (nut:Nut ) : Void
visitScrew (screw:Screw ) : Void
Scale
collapsed
Extensometer
collapsed

**Fig. 3.** Case Study: Most Detailed Stratum

## 3.5 Generating the System

Once the most detailed stratum has been obtained by unfolding refinement annotations, Fujaba's codegenerator may be used to generate the executable code. This involves converting all model-related constructs (classes, associations, etc.) into plain Java code and combining it with the code that has been accumulated by all strata refinements.

Note that the most abstract stratum already contains code that defines the behavior for a simple system at this (high) level of abstraction. The methods `process` and `processOne` belong in this category, as they contain the application specific code which cannot be generated automatically. In lower strata this code is enhanced to support functionality, such as observer notification.

It is important to point out that our approach allows *generating* code without compromising the need to add *hand-written* code. In addition to modifying existing code, refinement transformations create code blocks which are later filled with custom code by the developer. Examples for such code blocks are the behavior for the visualization buttons and the implementation of the `visit...` methods of the Visitor pattern. The absence of such code blocks—which allow stratum designers to fill in behavior that cannot possibly be generated—will be the rule rather than the exception. Only transformations as simple as those implementing the Singleton pattern, will be executable without further assistance by stratum designers. How to support the persistence of stratum designer supplied code blocks, i.e., how to make them survive future regeneration steps, is the subject of the following section.

## 4 Multi-Level Editing

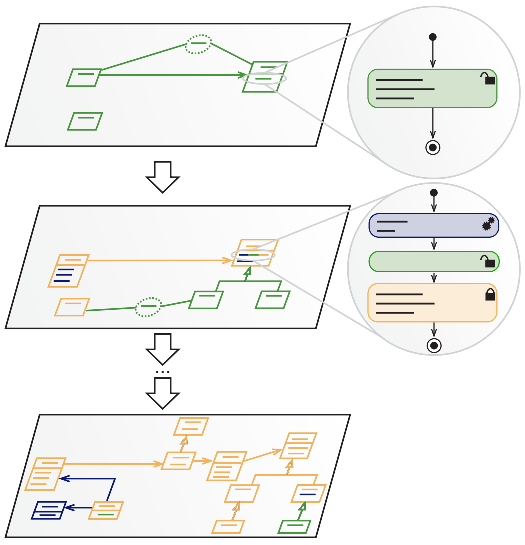


**Fig. 4.** Transformation with Hook Spots

According to Architecture Stratification, the most abstract stratum already contains a simplified system description including its behavior. Figure 4 shows these user-supplied code and model fragments using the color green.

As the system is refined in lower level strata, however, additional classes and code have to be provided. Some of this code can be generated by the refinement transformations directly (e.g., the “Singleton” implementation). Figure 4 shows such parts in dark blue. Note that generated elements (dark blue) or user-supplied elements (green) at one stratum will be treated as predefined/old elements (bright orange) at the stratum below.

A simple example for the user-supplied (green) category are the `visit...` methods (see classes `Human` and `Scale` in Figure 3). These are relatively easy to deal with, since they are uni-colored green, i.e., their method bodies contain new code only. In general, methods may contain a mixture of old (orange), generated (blue), and new code (green). Consider the lower circle in Figure 4 which starts with a generated fragment (cogwheel symbol) followed by a green user supplied fragment (open padlock symbol) ending with an orange colored fragment (closed padlock symbol) which has been transferred from a stratum above.

### 4.1 Preserving New Parts

Even with a straightforward case, such as the Visitor methods, 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 transformation rule was changed) creates new `visit...` stubs, we do not want to loose the existing method bodies. Three main strategies exist for dealing with this problem:

**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 must change only 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 to the stratum above.

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 another good reason against such an "anarchistic" approach to stratum modification: If any of the refinement transformation rules have to be changed, e.g., since a supporting technology—such as a certain middleware solution—changes, the "free edits" in all strata below this point are potentially subject to change. Even with a good traceability mechanism in place (which would 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 than strictly necessary.

**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, e.g., the `visit...` 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.

**Constrained Editing** Based on the above observations we 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 5, 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 upper part of Figure 5 shows a method body that has been split into two blocks on purpose (see the next paragraph below). The lower part shows the result after a refinement transformation, which inserted a number of new code bits (blocks *generated.1* & *generated.2*) and supplied the 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.

The green *new.1* block has been inserted by the refinement step not because the refinement transformation 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 transformation.

Since the green parts designate areas of variability (just like "*hot spots* in frameworks" [12]) and fill in parts as predetermined by the stratum above (just like "*hook* methods" in the design pattern "Template Method" [6]) 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.

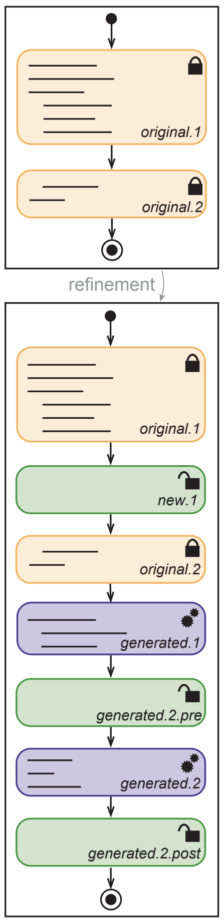


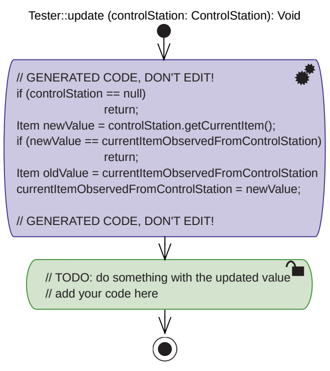
**Fig. 5.** Code Block Categories

## 4.2 Completing the Case Study

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 3), 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 6 shows the generated `update` method, featuring a first (blue) 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 (green) part then has to be implemented by a stratum designer and will be retained upon re-generation.

The visitor refinement transformation supports hook spots for the method referenced by *kickOffVisitor* and for all `visit...` methods of concrete visitor classes. The transformation code is called for each *concreteVisitor* link, adding the necessary `visit...` methods (including the respective hook spots) to the linked classes.

A third example for hook spots can be found in the framework integration annotation *CollectionView*: The current implementation of our visualization framework uses Java Swing as the GUI toolkit. If the implementation were adapted, e.g., in order to support the Eclipse SWT toolkit, then parts of the generated "glue code", which are responsible for creating and adding the buttons, would need to be modified. As these orange code blocks are maintained by the refinement transformation, they can easily be exchanged leaving the green user-supplied code blocks (containing the action code for the buttons) unchanged.



```
// GENERATED CODE, DON'T EDIT!
if (controlStation == null)
                return;
Item newValue = controlStation.getCurrentItem();
if (newValue == currentItemObservedFromControlStation)
                return;
Item oldValue = currentItemObservedFromControlStation
currentItemObservedFromControlStation = newValue;

// GENERATED CODE, DON'T EDIT!
```

```
// TODO: do something with the updated value
// add your code here
```

**Fig. 6.** Observer Hook Spot

## 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., they specialize on *exogenous* transformations [10]. Tools that support model refactorings can—according to [10]—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 tools provide basic support for defining vertical endogenous transformations as well. The following paragraphs shortly describe related commercial and academic tools, comparing them with our approach.

Together Architect[2] 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 process and will eventually support fully automated refinement 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[3] 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

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

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.

Similar to SPin, the model transformation framework "Mercator" [14] also uses UML class diagrams and corresponding model annotations to control transformations. It 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. Mercator provides both model to model and model to code transformation but does currently not offer the capability to add user provided code.

The "Bidirectional Object-Oriented Transformation Language" (BOTL) [9] 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 experience has been 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[4] 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", 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. This is supplemented by the so called "blueprints" which are similar to model templates. 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 MDSD is based on domain-specific languages (DSLs) instead of UML. So called "Software Factories" [7] 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, the "variability points" of Software Factories 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 [4]. This multi-layered modeling approach exhibits some similarities to stratification. The annotations within a stratum can be compared to

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

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 [1]. 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.

None of the above mentioned approaches support Fujaba's *Story-Driven-Modeling* feature [5], which is not only very useful for the semi-graphical specification of refinement transformation rules as usable in SPin, but also provided us with the basis for creating hook spots that survive re-generation steps.

## 6 Future Work

Although the rule library is 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 extend the library with further refinement transformation rules concerning other areas like security patterns, aspects and the integration of more complex frameworks.

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. Annotations exhibit natural dependencies and lend themselves to generate levels of system concerns [3]. It is therefore the task of the stratum designer to select which of the annotations are addressed at each specific abstraction level. As a result, future versions of SPin will provide a configuration system, allowing users to specify and store their annotation processing orders.

The current approach to specify refinement transformations with imperative instructions, including unconstrained Java code, implies that there is no easy way to automate traceability, e.g., supporting forward updates or backward-navigation. We are therefore investigating the usage of graph rewriting approaches [8], e.g., Triple Graph Grammars [13], to automatically maintain consistency links between adjacent strata. Such bi-directional refinement transformations would also represent an attractive facility for reverse engineering, i.e., starting from a complex system and simplifying the system by either using refinement transformations in the "reverse" direction or specifying and applying dedicated "abstraction transformations".

We plan to expand on our synchronized model and code transformation approach by looking at more sophisticated code transformations, i.e., strengthen the support for more involved code transformations. We will, moreover, provide hook spots for general model elements, over and above code fragments.

## 7 Conclusion

In this paper we have demonstrated the utility of Architecture Stratification, and our SPin prototype tool supporting it, by means of a small case study.

The basis of creating a hierarchy of abstraction levels, all individually describing the intended system, are model transformations. The corresponding SPin refinement transformations are user-definable, typically by using a convenient mix of SDM (for pattern matching and creation of model elements) and Java (for an unconstrained definition of transformations). Their usage is indicated by employing 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).

The creation of new transformation rules using SPin is heavily assisted by a number of convenient utilities, such as support for modifying method bodies, element creation, and synchronization with the UML metamodel. SPin also allows immediate testing and application of newly created annotations and associated refinement transformations.

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. Such refinement steps may involve refinement annotation that in turn unfold into further refinement annotations. It was thus possible to tie in a generic visualization framework to an application with minimal effort and minimal conceptual pollution of the top-level stratum.

We have furthermore presented a useful compromise for multi-level editing, ranging between anarchistic "free-editing" and obstructive "no editing", based on the concept of *hook spots* which enable controlled amendments to both model elements and code. The combination of transformation-defined and stratum designer inducible hook spots provides a scheme that is a) implementable within the current limitations of our plugin environment Fujaba and b) more than sufficient for providing extra information at lower strata.

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.

As a result our concepts and tool support go some way towards helping to deal with the complexity of today's applications. By capturing recurring software aspects by reusable transformation rules, such systems can be built faster and more reliably.

## References

1. J. P. Almeida, R. Dijkman, L. F. Pires, D. Quartel, and M. 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.
2. C. Atkinson and T. Kühne. Separation of Concerns through Stratified Architectures. International Workshop on Aspects and Dimensions of Concerns, ECOOP 2000, Cannes, France, June 2000.
3. C. Atkinson and T. Kühne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.
4. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In R. Nord, editor, *Proceedings of the Third Software Product-Line Conference*, Lecture Notes in Computer Science. Springer-Verlag, September 2004.
5. T. Fischer, J. Niere, L. Torunski, and A. 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.
6. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Addison-Wesley, 1994.
7. J. Greenfield and K. 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.
8. A. Königs. Model transformation with triple graph grammars. In *Model Transformations in Practice, Satellite Workshop of MODELS 2005, Montego Bay, Jamaica*, 2005.
9. F. Marschall and P. 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.
10. T. Mens, K. Czarnecki, and P. V. Gorp. A taxonomy of model transformations. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.
11. U. Nickel, J. Niere, and A. Zündorf. The FUJABA Environment. Technical report, Computer Science Department, University of Paderborn, 2000.
12. W. Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design. In M. Tokoro and R. Pareschi, editors, *proceedings of the ECOOP, Bologna, Italy*, pages 150–162. Springer-Verlag, 1994.
13. A. 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, 1994. Springer Verlag.
14. W. Witthawaskul and R. 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.