Automated Support for Framework-Based Software Evolution

Tom Tourwé
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
Email: tom.tourwe@vub.ac.be

Tom Mens
Service de Gnie Logiciel
Universit de Mons-Hainaut
Av. Champs de Mars 6, 7000 Mons, Belgium
Email: tom.mens@umh.ac.be

Abstract

In this paper, we show how elaborate support for framework-based software evolution can be provided based on explicit documentation of the hot spots of object-oriented application frameworks. Such support includes high-level transformations that guide a developer when instantiating applications from a framework by propagating the necessary changes, as well as application upgrading facilities based on these transformations. The approach relies on active declarative documentation of the design and evolution of the framework's hot spots, by means of metapatterns and their associated transformations.

1. Introduction

Over the past years, object-oriented software development based on framework technology has become extremely popular and has gained widespread acceptance. The major reason is that such a development method offers significant software engineering benefits: it allows design reuse, as opposed to mere code reuse, reduces application development time, improves evolvability, promotes consistency between applications, and so on [7, 21]. In other words, it brings us closer to a product-line approach of software development, where entire software families are being developed as opposed to stand-alone software applications [31].

The most important asset offered by an application framework is its design, that should be flexible and reusable to allow developers to build numerous applications within the same application domain. The design defines the specific places where the framework can be extended with application-specific code (the so-called *hot spots* of the framework [24]) and imposes particular constraints upon the application's implementation. In order to conform to the framework specification, the instantiated applications must *fill in* the appropriate hot spots and adhere to the framework design, to ensure that no constraints are violated.

In practice, it turns out that the design of the framework is not adequately documented, and as a consequence, neither are the hot spots and the constraints [25, 3]. As a result, these are only implicitly present in the implementation. It should thus come as no surprise that correctly instantiating an application from a given framework is a complex and error-prone task. Many times, applications do not fill in the appropriate hot spots, or use these hot spots in the wrong way, and thereby violate the intended design of the framework.

The above problem is aggravated by the fact that a framework is inevitably subject to constant evolution, as requirements are changed, added or removed. Clearly, such evolution affects existing instantiations, which may need to be updated. Such updating requires very detailed and specific information about how each instantiated application reuses the framework's design and how this design has evolved. Since neither the design, nor the instantiation or evolution of a framework are adequately documented, the update process is labour-intensive and error-prone.

To alleviate these problems, we propose to document a framework's hot spots by means of *metapatterns* [24], an advanced abstraction of design patterns that was conceived out of the observation that many design patterns share the same underlying structure. Although design patterns are useful for our purposes as well, we deliberately choose not to use them as a basis for our approach, because the number of design patterns is quite large and this would endanger the scalability of our approach. The information conveyed within a metapattern allows us to document the hot spots in an accurate manner and to specify in which ways they can be filled in. As such, each metapattern comes with a number of transformations, that prescribe the specific changes that should be applied to fill in its hot spots. These transformations can be explicitly and formally defined and can be automated. They thus serve as a basis for an approach that supports instantiating concrete applications from a framework, change propagation and framework evolution and up-

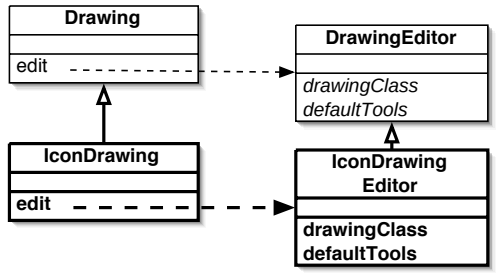

Figure 1. A Change Propagation Example

grading.

In what follows, we first clarify the problem statement by providing two examples (Section 2). Afterwards, we elaborate upon the specific solution we propose to alleviate these problems (Sections 3 and 4). In Section 5, we show how this solution effectively addresses the problems specified above. Section 6 discusses tool support, Section 7 lists related work, Section 8 contains future work, and Section 9 presents our conclusions.

2. Running Examples

In this section, we present two typical problems encountered when using a framework-based software development method: change propagation and application upgrading. Both examples are taken from the documented evolution of HotDraw, a popular framework that has been used to build numerous graphical applications [2].

2.1. Change Propagation

An application developer instantiating [1] a concrete application from a framework has to ensure he provides application-specific code for all necessary hot spots. As such, various additions to the basic framework are required. In particular, classes should be added and the appropriate methods should be overridden in these classes. Although seemingly simple, this can be quite a tedious and error-prone task. For one, the developer is expected to be able to identify all necessary hot spots and to know how these should be filled in. An additional problem is that, more often than not, dependencies exist between these hot spots. Filling in one particular hot spot may require filling in other hot spots as well. This phenomenon is known as the *ripple effect* [34]. Furthermore, the particular way in which one hot spot is filled in may constrain the way these other hot spots are filled in as well. Fulfilling all these requirements turns out to be a tedious task given the lack of suitable and decent documentation.

A particular example of the ripple effect in the HotDraw framework is depicted in Figure 1. The `Drawing` class provides a hot spot that allows an application developer to define an application-specific drawing class. In this particular case, the hot spot is filled in by defining an `IconDrawing` subclass. One particular dependency defined by the HotDraw framework is that each drawing has an associated drawing editor, that can be used to edit the application-specific drawing. In order to satisfy this constraint, the developer has to fill in another hot spot, by defining a subclass `IconDrawingEditor` of class `DrawingEditor`. Moreover, another dependency requires him to link the newly added classes by providing two methods: a `drawingClass` method on class `IconDrawingEditor` that returns the particular drawing class associated with the editor, and an `edit` method on class `IconDrawing` that opens the appropriate editor on the particular drawing object. Even more, the `DrawingEditor` class hierarchy contains a hot spot that allows developers to specify the tools that are applicable upon a specific drawing. These tools should be registered by the `defaultTools` method. The `IconDrawingEditor` class should thus also define such a method.

As this example illustrates, what appears to be one small change, turns out to be a cascade of many different changes. Identifying the particular hot spot to be filled in, and the way in which this should be achieved, is already quite cumbersome given the lack of suitable documentation. Consequently, identifying the relationships between the different hot spots, deriving which additional hot spots should be filled in and knowing how to do this, is extremely difficult without some form of (partially) automated support.

2.2. Support for Application Upgrading

Given the size and complexity of current-day frameworks, it comes as no surprise that these frameworks and their applications are developed, maintained and evolved by several teams of developers. A typical situation that occurs in such a context is that one team is in charge of developing and evolving the framework itself, while one or more teams have the responsibility of instantiating this framework to different applications. When a new version of the framework is released, each instantiated application should preferably be upgraded to work with this new version, as it may contain enhanced or improved functionality. Such upgrading is far from trivial, however, and can lead to a number of *upgrade conflicts*, which need to be resolved in order to ensure the correct behaviour of the instantiated application [19].

[1] Note that in this paper, we consider instantiation and evolution of a framework to be two distinct activities. Both can however make changes to the framework, and can thus be considered as an evolution of the framework.

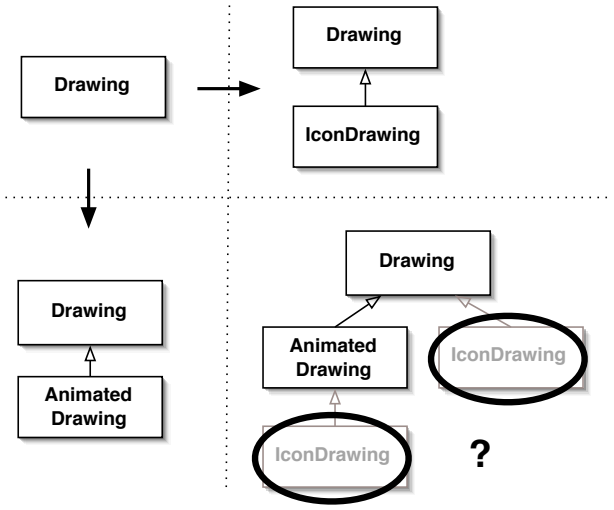

Figure 2. An Example Software Upgrade Conflict

As an example, consider the situation depicted in Figure 2 taken from an actual evolution of the HotDraw framework. The evolution of the framework introduces an `AnimatedDrawing` subclass of the `Drawing` class (left part of the figure), with the specific intent of isolating all animation behaviour in this new class. In parallel, and unaware of this specific framework evolution, an instantiation of the HotDraw framework provides a new subclass `IconDrawing` of class `Drawing` (upper part of the Figure). Clearly, this creates an upgrade conflict when we want to upgrade the framework instantiation to use the new version of the framework: the `IconDrawing` class may rely on behaviour of the `Drawing` class that has now been moved to the `AnimatedDrawing` class. It is clear that we cannot simply merge the two changes and be done with it. To resolve this situation, the application developer needs semantic information to determine whether the `IconDrawing` class should remain a subclass of `Drawing` or whether it should be changed to `AnimatedDrawing`.

Upgrade conflicts occur because one change relies on assumptions that are broken by another change that is applied in parallel. In the concrete example presented above, the application developer that adds the `IconDrawing` class correctly assumes the `Drawing` class provides the only available hot spot to be filled in. By extending the framework, an additional hot spot is introduced, thereby breaking the assumption made by the framework instantiator. Since there exists very little documentation about how the framework's design is instantiated or evolved, such conflicts can only be detected by manual inspection. Clearly, this is once again an error-prone and time-consuming task.

2.3. Discussion

The two problems sketched above are due to two principal shortcomings in current-day framework-based software evolution: the lack of suitable documentation of the framework's design and the way it evolves, and the lack of active support for framework instantiation and application upgrading.

A framework defines strict rules for its instantiation, that should be adhered to at all times, by each application derived from the framework. However, a developer currently has very little support for instantiating the framework, to help him ensuring that the appropriate rules are adhered to, and to automate some of his tasks. Therefore, he has no other option than to perform these tasks manually. We believe automated support is feasible, however, provided that the framework hot spots and instantiation rules are known beforehand.

Likewise, little to no support is available for framework upgrading. Once again, this is due to a lack of documentation: since instantiating and evolving a framework is a manual task, no traces of the applied changes are left. This makes it very hard for a developer to detect upgrade conflicts, let alone identify ways to alleviate them. Furthermore, the lack of explicit documentation rules out automated support for application upgrading completely. We consider such support possible however, if both the instantiation and evolution process are explicitly documented. This would enable us to reason about the applied changes, detect possible upgrade conflicts and propose adequate solutions for them.

3. Metapatterns

In this section, we introduce the notion of metapatterns, that will be used to explicitly document a framework's design by means of its hot spots. The next section will then introduce metapattern transformations, that serve to explicitly document how the framework's design is reused and evolved.

3.1. General Overview

The definition of metapatterns is based on a distinction between *template* and *hook* methods, the corresponding *template* and *hook* classes and the specific ways in which these classes are related:

- A *template* method is a concrete method that calls some other methods, which are the *hook* methods. Hook methods can be abstract methods, regular methods with a default implementation intended to be overridden, or template methods in their turn.

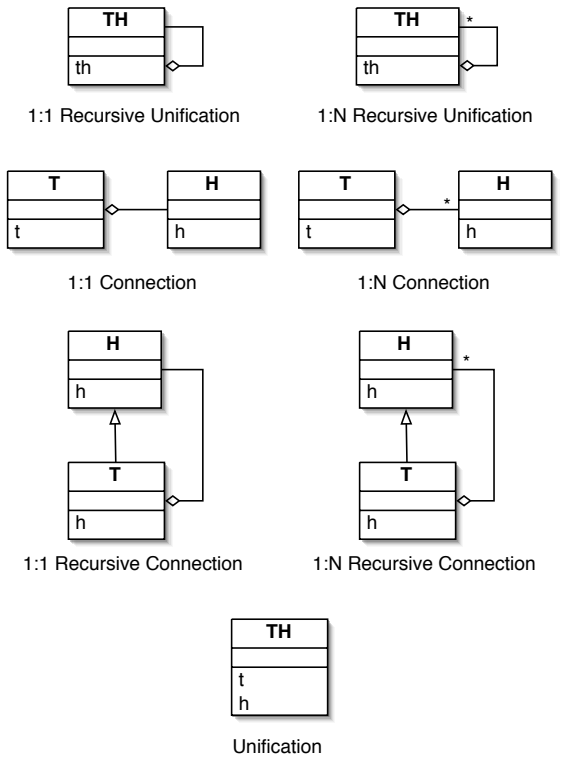

Figure 3. The Existing Metapatterns

- A *template* class is a class that implements a template method, and similarly, a *hook* class is a class that implements a hook method.

Template classes need to be combined with hook classes, in order for template methods to be able to call hook methods. A metapattern captures a particular combination of a template class and a hook class. Two aspects influence this combination:

1. The cardinality of the association relationship between the template class and the hook class. An object of the template class may refer to exactly one object of the hook class, or it may refer to multiple objects of this class.

2. The hierarchical relationship between the template class and the hook class. The template class and the hook class may be unified into one class, or they may or may not be related via an inheritance relation.

Figure 3 shows the metapatterns as defined by [24]. Our own definition of metapatterns is an extension of this classification, since we explicitly include the notion of a class hierarchy. Consequently, we are able to represent hook hierarchy participants (as in [24]), as well as template hierarchy participants (not present in [24]), and this allows us to define two additional interesting metapatterns. For the sake of the discussion in this article, however, the precise details and differences between our definition and the one in [24] are irrelevant.

3.2. Notation

Before continuing with the formal definition of metapatterns, we should first explain our notation. In the remainder of the paper, a single class is denoted with an uppercase letter (e.g., *C*), while a lowercase letter (e.g., *m*) represents a method. Each method name is preceded by a class name that indicates where that method is defined. For example, $C :: m$ means that the class *C* defines a method m. We extend this notation to $C :: \mathcal{M}$, where $\mathcal{M}$ represents a set of methods. If *C* is a class in the framework, we use *hierarchy(C)* to denote the class hierarchy generated by *C*, i.e., class *C* together with all its direct and indirect descendants. A class hierarchy is also denoted textually by a $\mathcal{H}$ symbol. A class hierarchy $\mathcal{H}$ always defines an $inherits$ relation which forms a partial order. It has a unique maximal element $root(\mathcal{H})$ and a set of minimal elements $leafs(\mathcal{H})$. Intuitively, the maximal element of a class hierarchy corresponds to the root of that hierarchy, while the set of minimal elements corresponds to the set of leaf classes of the hierarchy.[2] Note that leaf classes are not required to be direct subclasses of the root class. If a class hierarchy $\mathcal{H}$ implements a set of methods $\mathcal{M}$ we use the familiar notation $\mathcal{H} :: \mathcal{M}$. Formally, this notation implies the following two constraints:
(i) $\forall m \in \mathcal{M}$: m is "abstract" in $root(\mathcal{H})$
(ii) $\forall m \in \mathcal{M}$: $\forall C \in leafs(\mathcal{H})$: m is "understood" by C

As a special case of "abstract", we also allow the method to provide a default implementation that must be overridden in subclasses. m is "understood" by C means that the class C itself, or one of its ancestors, should provide a concrete implementation of method m.

3.3. Formal Definition

Formally, a **metapattern** MP is defined as a tuple $\langle P, R \rangle$, where P is a set of participants and R is a set of relations that hold between these participants. An **instance** MP_i of a metapattern MP is a mapping of concrete software artifacts (classes, class hierarchies, methods, method sets, and variables) onto the participants of the metapattern.

Each participant $p \in P$ is a tag-value pair where the tag denotes the specific role the participant fulfils in the metapattern, and the value is the software artifact that plays this role in the metapattern. For example, a class hierarchy $\mathcal{H}$

[2] We assume in this paper that we have single inheritance. Therefore, there is a unique maximal element, the root class.

that plays the role of hook hierarchy participant is denoted by $(hookhierarchy, \mathcal{H}) \in P$.

Each relation $r \in R$ specifies how the different participants of the metapattern are related and how they interact with one another. Examples of such relations are $understandsMessage(C, m)$, that specifies that class C understands method m, or $inherits(\mathcal{H}_1, \mathcal{H}_2)$, that specifies that class hierarchy $\mathcal{H}_1$ is a subhierarchy of class hierarchy $\mathcal{H}_2$ (in other words, the root class of $\mathcal{H}_1$ is a (possibly indirect) subclass of the root class of $\mathcal{H}_2$). The formal definition of all the relations that can hold between two participants is not included here for lack of space. In [29], these relations are implemented as logic predicates in *SOUL*, a logic programming language implemented on top of the Smalltalk object-oriented programming language.

3.4. Example

As a concrete example, the *Unification* fundamental metapattern is formally defined as follows:

$unificationMP = \langle P, R \rangle$, where
$P = \{\ (hookhierarchy, \mathcal{H}),$
$\quad (templatemethods, \mathcal{H} :: \mathcal{M}_t),$
$\quad (hookmethods, \mathcal{H} :: \mathcal{M}_h)\ \}$
and
$R = \{\ understandsMessage(root(\mathcal{H}), \mathcal{H} :: \mathcal{M}_t),$
$\quad definesMethod(root(\mathcal{H}), \mathcal{H} :: \mathcal{M}_h),$
$\quad understandsMessage(leafs(\mathcal{H}), \mathcal{H} :: \mathcal{M}_h),$
$\quad invokes(\mathcal{H} :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h)\ \}$

This metapattern contains three participants: a hierarchy $\mathcal{H}$, a set of template methods $\mathcal{H} :: \mathcal{M}_t$ and a set of hook methods $\mathcal{H} :: \mathcal{M}_h$. The template class and the hook class of this metapattern are one and the same class: the root class of the hierarchy $\mathcal{H}$ (see Figure 3). This class implements all of the template methods $\mathcal{H} :: \mathcal{M}_t$, each of which invokes one or more of the hook methods from $\mathcal{H} :: \mathcal{M}_h$. These hook methods are defined by the root of the hierarchy and are provided with a concrete implementation for all concrete leaf classes of the hierarchy.

In the HotDraw example, one of the instances $un1$ of the above metapattern can be obtained by means of the following participant mapping:

$P = \{$
$\quad (hookhierarchy, hierarchy(DrawingEditor)),$
$\quad (templatemethods,$
$\quad\quad hierarchy(DrawingEditor) :: \{initialize\}),$
$\quad (hookmethods,$
$\quad\quad hierarchy(DrawingEditor) :: \{defaultTools\})\ \}$

4. Metapattern Transformations

In this section, we introduce metapattern transformations, that can be used to add participants to a metapattern instance, and as such document the instantiation and evolution of a framework.

4.1. General Overview

Because metapatterns are used to implement hot spots, they implicitly contain knowledge about how these hot spots need to be filled in. Such filling in consists of adding the appropriate participants to the various metapattern instances, which corresponds closely to adding application-specific behaviour to the framework. As we have argued before, this is often not a matter of one single change, but may require many successive changes in order to guarantee that the appropriate design constraints are preserved. We make such knowledge and the corresponding changes explicit, by providing *metapattern transformations* that can be used to support a developer when instantiating the framework. Not only does this improve the quality of the resulting applications, since all required hot spots are filled in in the appropriate way, it also enables us to reason about the instantiation and evolution of the framework at a higher level of abstraction.

4.2. Formal Definition

Each metapattern defines only two different transformations: one adding a class participant and one for adding a method participant:

- $addClass(\mathcal{H}, C)$ takes a hierarchy $\mathcal{H}$ and a class C as arguments and adds this class to the leafs $leafs(\mathcal{H})$ of the hierarchy $\mathcal{H}$ of the metapattern. A precondition of this transformation is that the class is not part of the hierarchy: $C \notin \mathcal{H}$. Note that this transformation can be used to add both hook class and template class participants, depending on which class hierarchy we pass as the first argument (the *hookhierarchy* or *templatehierarchy* participant respectively).
- $addMethod(\mathcal{H} :: \mathcal{M}, m)$ adds method m to the set of methods $\mathcal{H} :: \mathcal{M}$ of the metapattern. An obvious precondition of this transformation is that this method is not yet part of the method set: $m \notin \mathcal{H} :: \mathcal{M}$. Similar to the *addClass* transformation, this transformation can be used to add both hook method and template method participants, depending on which set of methods we pass as the first argument.

Note that these definitions are independent from a particular kind of metapattern, as they are expressed solely in terms of metapattern participants.

4.3. Example

As a concrete example, we can apply an *addClass(hierarchy(DrawingEditor), IconDrawingEditor)* transformation to metapattern instance *un1* to add the `IconDrawingEditor` class to the `DrawingEditor` class hierarchy (see Figure 1). Besides effectively adding the class, the transformation should also ensure to preserve the constraints of the *Unification* metapattern. Therefore, it should not only add a new class, but should also provide an implementation for all appropriate method participants. In this case, the `DrawingEditor` hierarchy is the *hookhierarchy* participant and therefore, all of its leafs should understand the `defaultTools` method, since this is the *hookmethod* participant. An implementation for this method is thus provided for the `IconDrawingEditor` class.

5. Support for Change Propagation and Upgrading

In this section, we explain how metapatterns and metapattern transformations can be use to provide support for change propagation and application upgrading.

5.1. Change Propagation

Support for change propagation is based upon *intradependencies* and *interdependencies* between metapattern instances, and the way these dependencies affect metapattern transformations that are applied. We will explain this issue in detail in the following subsections.

Intradependencies of a Metapattern Instance As mentioned before, each template method calls at least one hook method, and therefore, each template class is parameterised with a hook class. Such dependencies between class and method participants in a metapattern instance are called *intradependencies*.

Intradependencies influence the application of transformations on a metapattern instance. Clearly, whenever a new template class is added to a such an instance by means of an *addClass* transformation, a corresponding hook class should be added as well. Likewise for template and hook methods. The influence of intradependencies on metapattern transformations is thus summarised as follows:

$$\begin{aligned} &addMethod(\mathcal{H} :: \mathcal{M}_t, m_1) \Rightarrow \\ &\qquad addMethod(\mathcal{H} :: \mathcal{M}_h, m_2) \\ &addClass(\mathcal{H}_t, C_1) \Rightarrow \\ &\qquad addClass(\mathcal{H}_h, C_2) \end{aligned} \tag{1}$$

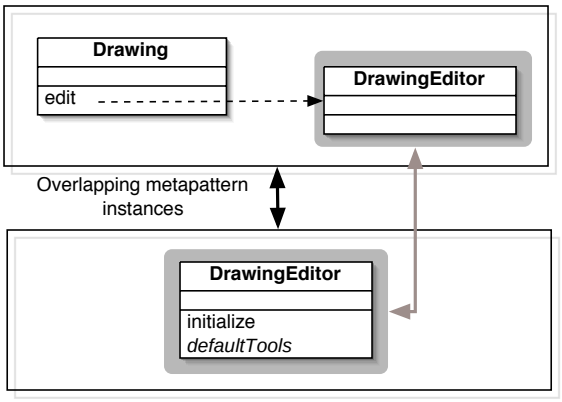

Figure 4. Interdependent Metapattern Instances

Interdependencies between Metapattern Instances When a metapattern transformation adds a new class participant to a class hierarchy in a particular metapattern instance, the same class should be added to all other metapattern instances in which the class hierarchy participates. Dependencies between different metapattern instances are based on shared participants, and are called *interdependencies* (as opposed to *intradependencies* discussed above).

Formally, two metapattern instances MP_i and MP_j are interdependent whenever a class hierarchy $\mathcal{H}$, that is a participant of MP_i, is also a participant of MP_j or a subhierarchy of this participant:

$$inherits(\mathcal{H}_i, \mathcal{H}_j) \tag{2}$$

$\mathcal{H}$ can be a *hookhierarchy* or *templatehierarchy* participant in either instance.[3]

Note that, since this definition of interdependence is defined in terms of the class hierarchy participants of the involved metapattern instances, we can specify the condition for interdependence independently from the kinds of metapattern that are involved.

Based on Equation 2, an *addClass* transformation applied to metapattern instance MP_i can give rise to an *addClass* transformation on the interdependent metapattern instance MP_j:

$$\begin{aligned} &addClass(\mathcal{H}_i, C) \Rightarrow addClass(\mathcal{H}_j, C) \text{ if} \\ &\qquad inherits(\mathcal{H}_i, \mathcal{H}_j) \end{aligned} \tag{3}$$

[3] Although there are other interdependencies between two metapattern instances, we will not discuss them here due to space restrictions. For a more elaborate discussion on this topic, we refer to [29].

This in essence forms the basis for our change propagation algorithm.

Examples Figure 1 contains three different metapattern instances of only two metapatterns. Besides metapattern instance *un1*, defined in Section 3.4, we can also identify two instances of the *Creation* metapattern:

cr1 which consists of class `Drawing` as the template hierarchy participant, class `DrawingEditor` as the hook hierarchy participant and method `edit` as the template method participant. Formally:

$$\begin{aligned} P = \{ & \\ & (templatehierarchy, hierarchy(Drawing)), \\ & (hookhierarchy, hierarchy(DrawingEditor)), \\ & (templatemethods, \\ & \quad hierarchy(Drawing) :: \{edit\}) \} \end{aligned}$$

cr2 which consists of class `DrawingEditor` as the template hierarchy participant, class `Drawing` as the hook hierarchy participant and method `drawingClass` as the template method participant. Formally:

$$\begin{aligned} P = \{ & \\ & (templatehierarchy, \\ & \quad hierarchy(DrawingEditor)), \\ & (hookhierarchy, hierarchy(Drawing)), \\ & (templatemethods, \\ & \quad hierarchy(DrawingEditor) :: \\ & \qquad \{drawingClass\}) \} \end{aligned}$$

Figure 4 shows an example of the interdependence between instances *cr1* (upper part) and *un1* (lower part). These two instances are interdependent because the class hierarchy defined by the `DrawingEditor` class participates in both instances as a *hookhierarchy* participant. As such, Equation 2 is satisfied:

$$\begin{aligned} & inherits(hierarchy(DrawingEditor_{cr1}), \\ & \quad hierarchy(DrawingEditor_{un1})). \end{aligned}$$

Additionally, we can identify the following interdependencies:

- Instance *cr1* is interdependent with instance *cr2*, due to two interdependencies. First, the `Drawing` class is a *hookhierarchy* participant of *cr1* and a *templatehierarchy* participant of *cr2*.

  $$\begin{aligned} & inherits(hierarchy(Drawing_{cr1}), \\ & \quad hierarchy(Drawing_{cr2})). \end{aligned}$$

  Second, the `DrawingEditor` class is a template hierarchy participant of *cr1* and a hook hierarchy participant of *cr2*:

  $$\begin{aligned} & inherits(hierarchy(DrawingEditor_{cr1}), \\ & \quad hierarchy(DrawingEditor_{cr2})). \end{aligned}$$

- Instance *cr2* is interdependent with instance *un1* since the `DrawingEditor` class is a *templatehierarchy* participant in *cr2* and a *hookhierarchy* participant in *un1*:

  $$\begin{aligned} & inherits(hierarchy(DrawingEditor_{cr2}), \\ & \quad hierarchy(DrawingEditor_{un1})). \end{aligned}$$

Now that we have identified the interdependencies, we are ready to show how they can be used to propagate changes throught the framework.

Instantiating the HotDraw framework by adding a concrete `IconDrawing` class should be specified by means of the appropriate metapattern transformation applied to one of the three metapattern instances. Note that we do not care about which specific metapattern instance the transformation is applied to, since the change propagation strategy will make sure the necessary changes are propagated appropriately.

Suppose we model the change by means of an *addClass(hierarchy(Drawing), IconDrawing)* transformation applied on metapattern instance *cr1* that adds the class as a template leaf participant to *cr1*. Conform Equation 1, this also requires the introduction of a corresponding hook leaf class, so an additional *addClass(hierarchy(DrawingEditor), IconDrawingEditor)* transformation is mandatory on the same instance. This will add the `IconDrawingEditor` class as a hook class participant to instance *cr1*.

Due to the interdependencies identified above, the same class should also be added to the interdependent metapattern instances. The second *addClass* transformation on instance *cr1* thus yields the following additional transformations (due to Equation 3):

- an *addClass(hierarchy(DrawingEditor), IconDrawingEditor)* operation on instance *un1*, which will add the `IconDrawingEditor` class as a hook leaf participant to instance *un1*.
- an *addClass(hierarchy(DrawingEditor), IconDrawingEditor)* operation on instance *cr2*, which will add the `IconDrawingEditor` class as a template leaf participant to instance *cr2*.

As we have explained previously, in practice, all these transformations should preserve the metapattern constraints. Therefore, they do not only add the appropriate classes to the framework's implementation, but also provide these classes with appropriate definitions for the required methods. For example, the *addClass(hierarchy(Drawing), IconDrawing)* transformation on instance *cr1* adds the `IconDrawing` class as a subclass of class `Drawing`, and additionally provides a default implementation for the `edit`

method or prompts the developer to provide the implementation.

5.2. Support for Application Upgrading

The approach we propose to detect possible upgrade conflicts is an operation-based merge algorithm that is an extension of similar techniques proposed by [19, 27]. It consists of mutually comparing the changes applied to a framework by means of metapattern transformations, and defining the conditions that lead to a possible conflict when the transformations are applied in parallel.

The condition for a *possible incorrect superclass* conflict, which is the conflict depicted in Figure 2, is the following:

$$\begin{array}{c} possibleIncorrectSuperclass(C_1, C_2) \\ if \\ addClass(\mathcal{H}_1, C_1) \parallel addClass(\mathcal{H}_2, C_2) \text{ and} \\ inherits(\mathcal{H}_1, \mathcal{H}_2) \end{array} \quad (4)$$

Such a conflict is caused by applying an *addClass* transformation in parallel (hence the $\parallel$ (parallel) notation) with another *addClass* transformation. Both operations each independently introduce a new class in the same class hierarchy (denoted by means of the $inherits$ relation), and may thus conflict with one another. There are of course many other conflicts that can be detected in a similar way [29], such as a *naming* conflict that arises when two transformations add a class (or a method in a class) with the same name, or a *constraint violation* conflict, that occurs because one transformations adds a method and a second transformations adds a class that does not implement the new method. We refer to [29] for an overview and a detailed discussion.

Naturally, we can not detect each and every possible conflict. Only when the evolution and instantiation of the framework can be expressed as a series of metapattern transformations will we be able to detect such conflicts. When two developers manually evolve the framework in parallel, conflicts can be introduced but will remain undetected by our approach.

Example The *possible incorrect superclass* conflict in Figure 2 occurs because two conflicting metapattern transformations are applied in parallel. The addition of the `IconDrawing` class can be specified as an *addClass* transformation on instance *cr2*. Likewise, the addition of the `AnimatedDrawing` class can be specified by the same transformation on instance *cr1*. As such, Equation 4 is satisfied and a conflict is reported. To summarise, the parallel application:

$$\begin{array}{c} addClass_{cr2}(hierarchy(Drawing), IconDrawing) \\ \parallel \\ addClass_{cr1}(hierarchy(Drawing), \\ AnimatedDrawing) \end{array}$$

leads to a conflict because

$$\begin{array}{c} inherits(hierarchy(Drawing_{cr2}), \\ hierarchy(Drawing_{cr1})) \end{array}$$

Note that the conflict would also be reported if both changes were modelled using transformations on the same metapattern instance. For example, adding the `IconDrawing` class could also be modelled by an `addClass` transformation on instance *cr1*. The above conflict would then also detected, because Equation 4 then trivially holds.

6. Tool Support

The approach we have proposed here has been implemented in the *SOUL* logic meta programming environment [33]. The tool we implemented provides support for framework-based software evolution that includes:

1. providing a design pattern view over the source code of the framework. A developer is expected to annotate a framework's design with information regarding the design patterns used. This information is then translated automatically into the corresponding information about metapatterns, in order to be able to apply our techniques. Furthermore, the inter- and intradependencies between the metapatterns is computed automatically. Although manual annotation may come about as a burden for the developer, we believe it is not as bad as it seems. The information about design patterns can be used for many other purposes as well, as has been argued many times [15, 1].

2. providing a list of transformations that a developer can apply to instantiate or evolve a framework. The list includes high-level transformations, such as *design pattern specific* or *framework-specific* transformations, that are defined in terms of metapattern transformations. It should be considered as a complement to the list of refactorings that current-day integrated development environments offer, since it can be used in exactly the same intuitive way. Moreover, the tool actively guides the developer when performing such changes, by pointing out the hot spots that need to be filled in and identifying the transformations that this requires. Whenever possible, the tool fills in some of the hot spots automatically.

3. providing support for manual evolution. Although not discussed in this paper, the tool can use the metapattern relations to verify whether a framework's design still adheres to the appropriate constraints after it has been changed manually. As illustrated in [29], this approach effectively allowed us to detect a number of conflicts in the HotDraw framework.

4. logging all transformations performed by the different developers on a specific version of the framework, reason about them and detect possible upgrade conflicts. Whenever such a conflict is detected, the tool presents the developer with a list of actions that he can undertake to resolve the particular conflict.

Of course, the tool is only a research prototype, and does not come with all the bells and whistles that should be present if it is to be used in an industrial setting. This should be considered future work.

7. Related Work

Many tools exist that are able to verify whether a developer instantiates a framework in a correct way [11, 12, 23, 5]. All these tools are based upon the idea of a task list, that contains remaining tasks to be performed by the developer. These tools do not guide the developer by providing transformations that perform part of the tasks automatically, however.

Other environments do provide automated transformations [26, 4, 14], and are based upon the concept of refactoring, first identified in [22]. However, they do not automatically guide a developer by telling him when or where the transformations should be applied, nor do they provide support for upgrading.

[9, 17, 30, 10] propose tools that provide extensive support for working with design patterns and frameworks. The tools provide automated transformations that can be used to evolve design pattern instances in predefined ways at a high level of abstraction. These transformations are similar to our metapattern transformations, but the tools' primary focus is not change propagation or application upgrading, so these problems are not addressed. [20] presents a similar tool that does provide (basic) support for such activities.

Many tools and techniques exist for identifying possible evolution conflicts caused by different developers evolving an application in parallel [16, 8, 6, 13]. All of them are based upon comparing lower level (statement level) changes to the source code. As a result, they can only detect and report conflicts at a low level of detail.

8. Future Work

In order for our approach to work, we need explicit information about the metapattern instances present in a framework. Our current tool gathers such information from design pattern information specified by the developers. Other experiments seem to point out that design patterns can also be detected automatically in the source code [18, 32, 28]. We have not yet experimented with these techniques, but they seem very worthwhile to investigate in future work, as they may relieve the developer from annotation the framework manually and may thus reduce the risk of errors.

As already mentioned, an absolute prerequisite for the approach presented here, is that the developers explicitly invoke metapattern transformations to evolve or instantiate a framework. This could be considered a disadvantage, since currently frameworks are changed manually more often then not (although automated refactorings form the exception that confirms the rule [26]). However, we firmly believe that such information can be extracted automatically from the source code, by comparing different versions of the framework. Some preliminary experiments have already been set up, although there is much more work that remains before any general conclusions can be drawn.

The practical validation of our approach so far was performed on the HotDraw framework. Although a real-world and popular framework, HotDraw still remains a rather small-scale artifact. The specific reason for first performing some small-scale experiments is that we wanted to fine tune our approach in a first stage, in order to be able to test it in an industrial environment only later on. As such, we required a controlled setting, in which we had many different, fixed and stable versions of the framework at our disposal. Such is very difficult to achieve in an industrial environment. Quite surprisingly, we were able to identify some important flaws and inconsistencies in the design of the HotDraw framework, despite the fact that it is regarded as a high-quality framework, is often cited as the prototypical example of a well designed framework, has been used many times and has known many revisions. This leads us to believe that our approach performs quite well. Furthermore, by incorporating automatic extraction techniques for design patterns, their intra- and interdependencies and the transformations applied to them, we effectively improve the scalability of our approach, which leaves us confident that the approach will be valuable on large-scale frameworks as well.

9. Conclusion

In this paper, we have argued that elaborate support for framework-based software evolution can be provided if explicit documentation of the framework's design, its instan-

tiation and evolution is available. The support is based on active and declarative documentation by means of metapatterns and their associated transformations. This allows us to provide automated transformations that guide developers when instantiating the framework, by propagating the appropriate changes, as well as application upgrading facilities based on these explicit transformations. By means of two typical examples taken from a real-world framework, we have shown how the approach works in practice and can be used to improve the framework-based software development process.

References

[1] K. Beck and R. Johnson. Patterns Generate Architectures. In *Proceedings of the European Conference on Object-Oriented Programming*, 1994.

[2] J. M. Brant. Hotdraw. Master's thesis, University of Illinois at Urbana Champaign, 1995.

[3] G. Butler and P. Dénommée. *Documenting Frameworks to Assist Application Developers*, chapter 7. John Wiley and Sons, 1999.

[4] I. Corporation. Intellij idea, http://www.intellij.com/idea.

[5] W. De Meuter, M. D'Hondt, S. Goderis, and T. D'Hondt. Reasoning with Design Knowledge for Interactively Supporting Framework Reuse. In *Proc. of the SCASE (Soft Computing applied to Software Engineering) Conf.*, 2001.

[6] W. K. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. In *Proc. Symp. User Interface Software and Technology*, ACM Press, 1997.

[7] M. Fayad and D. C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10), 1997.

[8] M. S. Feather. Detecting Interference when Merging Specification Evolutions. In *Proc. 5th Int'l Workshop Softw. Specification and Design*, ACM Press, 1989.

[9] G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Object-Oriented Patterns. In *Proceedings of ECOOP'97*, 1997.

[10] D. Gruijs. A Framework of Concepts for Representing Object-Oriented Design and Design Patterns. Master's thesis, Utrecht University, 1997.

[11] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Annotating Reusable Software Architectures with Specialization Patterns. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 171–180, 2001.

[12] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Generating Application Development Environments for Java Frameworks. In *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01)*, pages 163–176, 2001.

[13] S. Horwitz, J. Prins, and T. Reps. Integrating Non-Interfering Versions of Programs. *ACM Trans. Programming Languages and Systems*, 11(3):345–387, 1989.

[14] O. T. International. The Eclipse Platform, http://www.eclipse.org.

[15] R. Johnson. Documenting Frameworks Using Patterns. In *Proc. of the OOPSLA Conf. on Object-Oriented Programming, Systems, Languages and Applications*, 1992.

[16] C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1997.

[17] M. Meijers. Tool Support for Object-Oriented Design Patterns. Master's thesis, Utrecht University, 1996.

[18] K. Mens, I. Michiels, and R. Wuyts. Supporting Software Development through Declaratively Codified Programming Patterns. In *Proc. Int. Conf. Software Engineering and Knowledge Engineering*, 2001.

[19] T. Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1999.

[20] T. Mens and T. Tourwé. A Declarative Evolution Framework for Object-Oriented Design Patterns. In *Proc. Int. Conf. Software Maintenance*. IEEE Computer Society, 2001.

[21] S. Moser and O. Nierstrasz. The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer*, 29(9):45–51, 1996.

[22] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana Champaign, 1992.

[23] A. Ortigosa, M. Campo, and R. M. Salomon. Enhancing Framework Usability through Smart Documentation. In *Proc. of the Argentinian Symposium on Object Orientation*, pages 103–117, 1999.

[24] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley/ACM Press, 1995.

[25] M. Rettig. Nobody Reads Documentation. *Communications of the ACM*, 34(7):19–24, 1991.

[26] D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 1997.

[27] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proc. OOPSLA96 Conf.*, ACM Sigplan Notices, 1996.

[28] P. Tonella and G. Antoniol. Inference of Object-Oriented Design Patterns. *Journal of Software Maintenance*, 13(5):309–330, 2001.

[29] T. Tourwé. *Automated Support for Framework-Based Software Evolution*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2002.

[30] P. van Winsen. (Re)engineering with Object-Oriented Design Patterns. Master's thesis, Utrecht University, 1996.

[31] D. M. Weiss and R. Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[32] R. Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In *Proc. TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.

[33] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.

[34] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple Effect Analysis of Software Maintenance. In *Proc. COMPSAC Conf.* IEEE Computer Society Press, 1978.