Nr.: FIN-009-2008

Transforming Object-Oriented Design Pattern Structures into Layers

Martin Kuhlemann

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical Report

Nr.: FIN-009-2008

Transforming Object-Oriented Design Pattern Structures into Layers

Martin Kuhlemann

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 10 MDStV):

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Martin Kuhlemann
Postfach 4120
39016 Magdeburg
E-Mail: martin.kuhlemann@ovgu.de

http://www.cs.uni-magdeburg.de/Preprints.html

Auflage: 61

Redaktionsschluss: 20.10.2008

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

Transforming Object-Oriented Design Pattern Structures into Layers

Martin Kuhlemann
University of Magdeburg, Germany
kuhlemann@iti.cs.uni-magdeburg.de

Abstract

Design patterns improve desired properties of object-oriented software. The Jak language implements layered designs that follow similar goals as design patterns, like wrapping methods on demand. In prior research we compared design pattern implementations coded in different languages, like Jak. We did not evaluate the approaches of implementing a pattern but only used the implementations for comparing languages. Therefore the compared implementations were rather small and do not say whether the proposals scale. In this study we transform and reimplement instances of all Gang-of-Four design patterns in programs like JHotDraw or Berkeley DB into Jak counterpart designs proposed earlier. We show to what extend and under which conditions object-oriented design patterns and Jak mechanisms correspond, e.g., for adding methods to classes on-demand. Most of the problems we found while transforming pattern implementations were related to diverse implementation noances of the same pattern in different programs. Second, some of the Jak approaches caused code replication or additional effort for ensuring consistent Jak module composition. Finally, passing parameters to one method in different configurations of Jak layers was more problematic than we expected before.

1. Introduction

Design patterns improve reusability for pattern-code, flexible composition of modules, maintainability, and other desired properties [6]. While design patterns can be reused as common designs across applications, source code that implements a design pattern possibly cannot. Several researchers addressed this lack of code reuse and proposed modules that implement design patterns and that can be reused [3, 13, 8, 1, 7]. In previous work [10], we analyzed these implementations and transformed some of them [8] into Jak counterparts. Our Jak implementations exposed several benefits compared to their object-oriented original. We additionally considered alternative pattern implementations that simplify the pattern implementations for Jak [9].

In [10, 9], we compared languages – AspectJ and Jak – using implementations of the *Gang-of-Four (GoF)* design patterns but did not evaluate the implementation proposal itself. Therefore, several questions remained unanswered: (1) can the Jak proposal implementation modularize pattern implementations in existing programs? (2) Which problems occur when transforming an object-oriented implementation of a design pattern into its Jak counterpart to leverage from Jak's benefits already shown? (3) Are the transformations revertible?

In this study, we transform pattern implementations for all 23 GoF design patterns in three programs using Jak mechanisms as proposed in [10, 9]. JHotDraw[1] [4, 16] is a GUI framework and is implemented using numerous design patterns; Berkeley DB[2] is an embedded database engine for Java, and *Expression Product Line (EPL)* [12] evaluates mathematical expressions. We concentrate on design properties that may prevent the transformation of object-oriented patterns to Jak counterparts. Therefore, we tested our layer composition results for type errors using a compiler.

In our study we found that only half of the pattern implementations could be transformed to the proposed Jak design – remaining patterns demand for changes or extensions to their Jak implementation proposal. The main problems that occurred relate to providence of parameters, code replication, and consistency of Jak module composition. Even though the intent of some patterns match Jak mechanisms directly, like adding methods to classes on demand or wrapping methods on demand, structural information got lost during transformation into Jak. Among others these problems contradicted and prevented our assumption of these transformations of object-oriented pattern implementations into Jak pattern implementations being reversible.

2. Background

In this section we review design patterns and Jak.

Design patterns. Design patterns are suggestions for solving recurring and diverse programming tasks [6]. Patterns propose classes to play dedicated roles of the pattern where each role originates similar code for these classes across applications. For example, the Observer pattern relates two sets of objects where *Subject* objects modify *Observer* objects on change. To implement the role of a subject (that is observed) or observer (that observes) according classes implement pattern related interfaces with methods. In case of the Observer pattern, all *Subject* classes (in every application) are meant to provide an observer-access method (that may manipulate a list field which references observers) and *Observer* classes are meant to implement a uniform *notify* method to allow a subject object to trigger its events to any type of observer uniformly [6].

Different researchers proposed design patterns [5, 17, 18, 11, 18] – in this work we concentrate on the *Gang-of-Four (GoF)* design patterns [6] because they are widely known and domain independent.

Jak A *collaboration* is a set of objects and a protocol that determines how the objects interact [15, 14]. Collaborations in Jak superimpose interacting classes with sets of interacting *class refinements* [2]. Class refinements add new fields and methods to superimposed classes or wrap methods of the base classes.

In Figure 1, the class refinement of the Jak collaboration in Figure 1b contains a class refinement *Point* (refinements are exposed using *refines* keyword, e.g., Line 12) which adds a field *color*

[1] http://sourceforge.net/projects/jhotdraw/

[2] http://www.oracle.com/database/berkeley-db/je/

```
1  public class Point {
2   private int x;
3   private int y;
4   public void setX(int newX){
5    this.x=newX;
6   }
7   public void setY(int newY){
8    this.y=newY;
9   }
10 }
```

(a)

```
12 refines class Point {
13  private Color color;
14  public void setColor(Color newC){
15   color=newC;
16  }
17  public void setY(int newY){
18   Super.setY(newY);
19   color = Color.RED;
20  }
21 }
```

(b)

Figure 1. Jak concepts used in that paper.

(Line 13) and a method *setColor* (Lines 14-16) to the class *Point* of Figure 1a. Wrapping methods to add statements is done by overriding these methods in class refinements. Method *setY* of class *Point* in Figure 1b refines method *setY* of the original class *Point* (Fig. 1a) by overriding. This overriding method calls the refined method using *Super* (Fig. 1b, Line 18) and adds statements (Line 19). In the remaining paper, we use mainly graphical UML-like representations of these classes instead of code.

3. Case Study

Jak and design patterns partly aim at common goals; both apply certain properties to a base structure of classes, e.g., design patterns as well as Jak add methods to objects of classes without editing these classes permanently (e.g., with the Visitor pattern or Jak refinements), design patterns as well as Jak concepts also add statements to methods on demand. While the object-oriented concepts used in design pattern implementations (inheritance and delegation) allow different variants at runtime, classes composed using Jak refinements do not vary at runtime – we call Jak variation that is based on composing refinements to be *static* and its implementation *static* too. In contrast, pattern variation is *dynamic* because object properties may be attached and vary at runtime. To prove the conjecture that certain patterns are equivalent to static counter-implementations in Jak, we – additionally to [10] – evaluated static pattern-implementations proposals [9] to modularize pattern code in this study.

3.1 Transformation Examples

In this section we describe in detail three patterns with representative implementations of our study – Adapter, Template Method, and Visitor. For them we describe observations when implementing and transforming their pattern structure into static and dynamic Jak variants. Note that these three patterns implement Jak mechanisms of on-demand addition of methods to classes or on demand wrapping of methods.

Adapter. The design pattern Adapter changes the interface of a class such that the class implements an interface of a reusing application [6, p.139]. Using design patterns, objects of an adapter class wrap objects of the adaptee class and forward method requests to

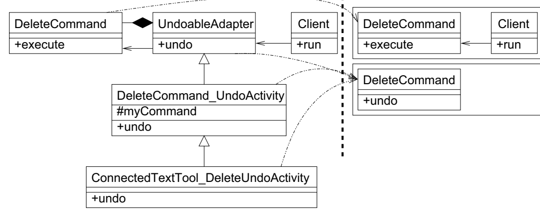

Figure 2. Sample transformation of object-oriented Adapter to Jak counterpart.

```
1 DeleteCommand cmd = new DeleteCommand("Delete",
    editor());
2 return new ConnectedTextTool_DeleteUndoActivity(cmd,
    getConnectedFigure());
```

(a)

```
1 return new
    ConnectedTextTool_DeleteUndoActivity(editor(),
    "Delete", getConnectedFigure());
```

(b)

Figure 3. Inlined adaptee instantiation.

the adaptee objects. Forwarding methods are named as in the desired reusing-application's interface. Conceptually, adapter classes introduce new methods into an adaptee class such that the class implements a desired interface.

In prior work, we examined two Jak implementation variants for the adapter pattern [9] where the adapting methods are introduced into an existing adapter class or into the adaptee class.

Inserting adapter methods into an existing adapter class of JHotDraw was straightforward after extracting it from there; in the stripped adapter class no method was left. However, if no adapter refinement is applied and thus no methods are added to the empty adapter class, it remains empty and method calls to the adapter class cause compiler errors. Furthermore, in JHotDraw, the adapter class is part of a hierarchy; if no method is added to this adapter class it may inherit inappropriate adapter methods; thus, subclasses must be ensured to not inappropriately extend the inherited methods.

Inserting adapter methods into adaptee classes via refinements is more complex. To transform an adapter class into an adaptee refinement, we (1) pushed the adapter class into a refining collaboration, (2) moved the adapter class to the package of the adaptee class, and (3) applied type changes to the adapter class (define the class to refine the adaptee class, redirect calls to the former adaptee object). In Figure 2 the adapter subclass *ConnectedTextTool_DeleteUndoActivity* turns into a refinement of the adaptee class *DeleteCommand* it wraps.

In JHotDraw, adapter classes inherit in deep inheritance hierarchies but when turned into an adaptee refinement these adapter refinements are no longer allowed to inherit classes.[3] Both solutions we considered have pros and cons. First, adapter-subclasses can inline all their superclasses (collapsing the hierarchy) before the adapter class is transformed into a refinement – this causes code replication for different adapter classes inlining the same superclass and precludes back-transformation. Second, all classes of the adapter hierarchy could be transformed into distinct refinements of the adaptee class; these *additional* refinements then must be applied when the transformed adapter class is applied

[3] In Jak, refinements generally cannot inherit from additional classes to avoid multiple inheritance.

and – when applied – the refinement from a former superclass of the transformed inheritance hierarchy must be applied *before* its former subclasses (i.e., the refinement created from the top-most hierarchy class must be applied first and hierarhy leaf-classes last). For Adapter, we decided to inline the superclasses, i.e., in Figure 2 classes *UndoableAdapter*, *DeleteCommand_UndoActivity*, and *ConnectedTextTool_DeleteUndoActivity* compose to one refinement *DeleteCommand* (We show an example of the alternative approach for Template Method.).

Calls or references (e.g., static types of variables) to the adapter class were manipulated to call or reference respectively the (refined) adaptee class. Among others we updated instantiations of adapter classes; these instantiations now instantiate an adaptee class and provide adaptee instantiation parameters. They do no longer provide an adaptee object because now the transformed adapter constructor is added to the adaptee class with refinements. In Figure 3a, different values are passed to instantiate the delegatee *DeleteCommand* (Line 1) which is then passed to instantiate the adapter *ConnectedTextTool_DeleteUndoActivity* object (Line 2); when transforming the adapter class into an adaptee refinement, the constructor of the refined class takes all (former adaptee and adapter) parameters (Fig. 3b). Finally, we changed several package and import statements because the adapter class has been moved and renamed.

Template Method. In prior work we analyzed two variants for implementing Template Method in Jak [9]. The first variant is a Jak counterpart of another study's AspectJ implementation [8] and detaches the *algorithm method* from the template class, a method that defines a common schedule for variant sets of primitive operations (defined in subclasses). The second proposal is static in that different sets of primitive operations are inserted into the template class using refinements, i.e., the algorithm together with one set of primitive operations are composed to one class and this composition cannot vary at runtime.

The first variant of detaching the algorithm method from the template class caused subtle problems in mixin layer implementations.[4] In JHotDraw the interface *Figure* of the abstract template class *AbstractFigure* declares the method *clone*, a method which then was implemented by the *AbstractFigure* subclass. *AbstractFigure defines* this method which we therefore consider an algorithm method. We moved all algorithm methods (and so the *clone* method) into a refinement of the stripped class *AbstractFigure* that then only contained declarations for primitive operation methods. With mixin compilation this stripped class *AbstractFigure* was transformed into an abstract class that is subtyped by classes transformed out of its former class refinements; thus, the abstract *AbstractFigure* class with the primitive operations did not define the *clone* method – this raised an compiler error. To fix that error, we noticed the algorithm method *clone* in advance in the stripped class with the primitive operations.[5] We needed advance notices like this and others (like *hooks methods*) for several patterns to allow methods of a preceding refinement to call methods a succeeding refinement introduces; we also needed advance notices (hooks) to re-introduce statements into methods, statements that belonged to a pattern implementation and were separated in a refinement (Jak cannot add a statement at arbitrary points into methods but can only wrap them.).

Note, while noting in advance the algorithm method *clone* for mixin layers is a very special observation, it shows the combinatorical complexity that layer decomposition may introduce and a developer has to keep track off; it also shows the need for hook methods that overcome limitations of Jak's join point model. In our example, the developer has to foresee for all module compositions, that once a *clone* method is introduced into an interface, the *refinement* that declares its composed class to implement this interface, defines or inherits a method *clone* – this may need restructuring of existing Jak modules.

The static implementation of the pattern Template Method adds primitive operation methods directly into the template class. For that, we transformed the primitive operation's classes (subclasses of the template class) into refinements of the template class[6] and made the template class a concrete class.

We changed the static type of objects of the former primitive operation classes into the type of the template class (that is now refined to expose primitive operations) – this caused type errors. In JHotDraw individual primitive operation classes expose unique interfaces because each primitive operation class defines additional methods. Thus, when we transformed these individual classes into refinements of the template class, the set of methods provided by the composed template class is unique to this configuration too. This caused calls to individual methods (on objects that had the static type of the according individual operation class before) to fail for certain layer combinations. Constructive user intervention is needed to delete these failing calls or to create stubs for the individual methods so that they are available in every layer combination. For the first option, all calling methods must be *replicated* in every refinement layer to strip the calls to methods unavailable if this particular layer applies; the second option *proliferates* the design with additional methods (hooks) that may be unneeded for most of the layer combinations.

In Template Method, we faced problems of parameter passing when transforming the primitive operation classes into refinements. In the original implementation of JHotDraw different primitive operation classes were instantiated in different contexts and with different parameters; thus a unified way of instantiating the composed template class was impossible, i.e., different instantiation calls lack access to appropriate constructor parameters for different compositions of refinements. If one refinement introduced more than one constructor, instantiation calls that formerly instantiated another class of primitive operations became ambiguous and required developer intervention.

In JHotDraw primitive operation classes implement new methods and new data types that are organized in a deep inheritance hierarchy. Intermediate classes of that hierarchy are instantiable on their own and thus their code is transformed into separate refinements anyway. This led us to the approach of organizing class refinements according to their former relationships in the inheritance hierarchy instead of inlining the superclasses (as described for Adapter).

In the static implementation variant for Template Method of JHotDraw assignments became unsound because the new composed template class lacks to subtype interfaces in some configurations (certain subtype declarations for the template class were declared only by some refinements – former subclasses). The developer again is needed to intervene.

[4] Mixin layers are a possible compilation result for Jak layers. Mixin layers simulate class composition by transforming refinements into subclasses of an automatically renamed base class.

[5] Note that the opposite decision of moving the primitive operation declarations into a refinement of a class containing the algorithm method *clone* causes similar problems. This is because in the mixin compilation the algorithm methods of a superclass call primitive operation methods of the subclass while these primitive operation methods are undefined in the stripped superclass.

[6] We moved the primitive operation class into a refinement layer and into the template class' package. We then made the former subclass a refinement of the template class.

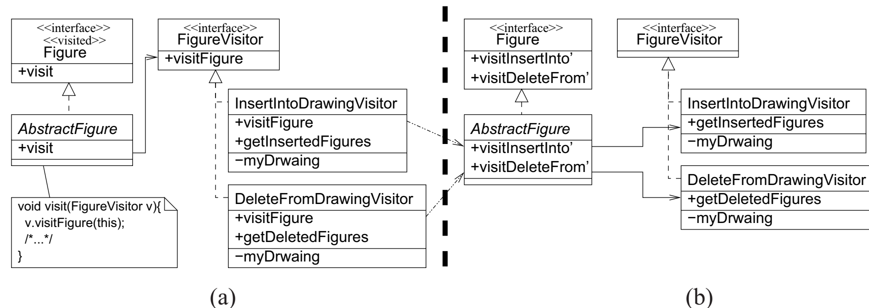

Figure 5. Decomposed Visitor remain as parameter object.

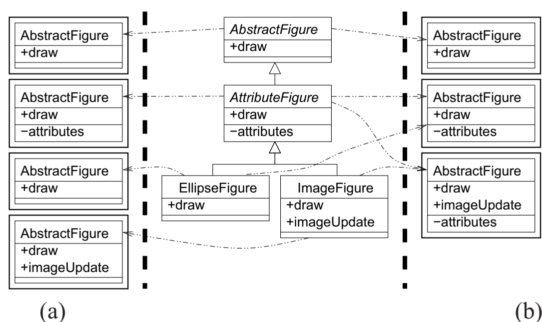

Figure 4. Deep inheritance hierarchy to Jak refinements.

In Figure 4 we show an example from JHotDraw. To transform the primitive operation classes *EllipseFigure* and *ImageFigure* into refinements of the template class *AbstractFigure* we had two options: (1) we could transform intermediate classes like *AttributeFigure* into their own refinements (Fig. 4a), or (2) we could inline these intermediate classes into each subclass before transforming each subclass into a refinement (Fig. 4b; done before for Adapter). The first variant needs to organize the refinements from the individual classes in order and presence according to their position in the former hierarchy, e.g., the *AttributeFigure* refinement must be applied whenever the *EllipseFigure* or *ImageFigure* refinement is applied and then must precede these refinements.[7] The second variant (Fig. 4b) replicates code in the transformation result and precludes back-transformation because methods are mixed and inlined. In either case, objects were retyped to the template class *AbstractFigure* which (1) disallowed calls to individual methods like *imageUpdate* for configurations without the method defining layer *ImageFigure*, and (2) caused problems of parameter passing because class *ImageFigure* takes an additional parameter for its constructor, this parameter now must be provided everywhere the lifted class *AbstractFigure* is instantiated (even there where *EllipseFigure* was instantiated before) – this was impossible.

Visitor. For Visitor we also analyzed two approaches of implementations that use Jak mechanisms. The Visitor pattern aims at adding methods to classes without editing these classes' definitions. The pattern uses double dispatch where a uniform *accept* method of the visited class to extend calls a specific method of its visitor parameter. The called visitor method corresponds to the dynamic type of the visited object – the visited object's type is extended virtually.[8]

The Jak implementation [9] separates all pattern related code fragments into refinements, like the *accept* method, the visitor classes, or subtype declarations toward pattern-related interfaces. A static way to implement the Visitor pattern's aim is to move methods (that were virtually introduced into classes) from the visitor classes into the hosting classes with class refinements.

We tried to reuse code from the former study for the Visitor implementation in JHotDraw but we encountered naming conflicts for methods.[9] Thus, we applied the concept of separating methods and classes to JHotDraw instead of reusing existing classes from our former study. We detached *accept* methods, visitor classes, and interfaces into refinements.

In [9], *accept* methods were added to the visited class by adding a subtype declaration toward a method defining class; however, this was not possible for JHotDraw. Visited classes in JHotDraw (that should define *accept*) already subtype classes and – due to the single inheritance constraint of underlying Java – cannot subtype additional classes. Therefore, we inserted the *accept* methods into the visited classes using Jak refinements. Since we only detached them from there before this was no big deal.

In the static implementation we moved the visitor methods into refinements of the classes that each visitor method virtually extended[10]; thus we extended according classes individually with Jak refinements instead of extending them virtually in a visitor object. Finally, we transformed visitor triggering statements to call the new methods instead.

In case the visitor class only includes methods virtually added to classes, the visitor class becomes empty and may be deleted. In JHotDraw some fields and methods are used within the visitor only and not added virtually to classes – to comprise these methods and fields we kept some visitor classes. Therefore, visitor classes are still passed to the new visited classes' methods as parameter container but they are not longer used for double dispatch.

In Figure 5, we exemplify our transformation for the Visitor pattern using JHotDraw. In Figure 5a the visited class *AbstractFigure*

[7] Note, the composition could also compile without layer *AttributeFigure* or with reordered layers but this is not equivalent to the former inheritance hierarchy.

[8] Calling the *accept* method of object *A* with a visitor parameter is equivalent to calling an added method on the object *A*.

[9] In JHotDraw, visitor methods were named according to the class they virtually extend instead of simply being called *visit* as in [9], e.g., visitor method *visitFigure* virtually extends the *Figure* class; double dispatch methods of the visitor class where named *visit* instead of *accept*. In the following we name double dispatch methods of the visited classes *accept* and virtually added visitor methods *visit*.

[10] In object-orientation the double dispatch method of the visited class, forwards its self-reference as an argument to an overloaded method of the visitor. Thus, the visitor method's argument type is the type this method is virtually added to.

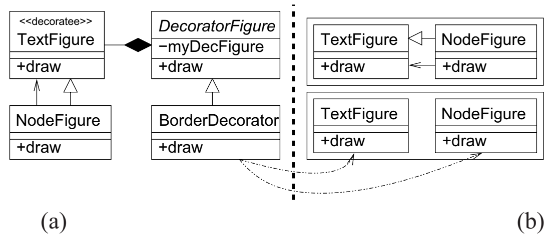

Figure 6. Doubled application of a feature.

accepts visitor objects for its *visit* method and calls the appropriate visitor method *visitFigure* with its self-identity as an argument. We inlined the *visitFigure* of every visitor class into the double dispatch method *visit* (that we duplicated and renamed before) of the visited class. The two visitor classes *InsertIntoDrawingVisitor* and *DeleteFromDrawingVisitor*, therefore, produce two methods (*visitInsertInto* and *visitDeleteFrom*; Fig. 5b) added to all visited classes.[11] The moved methods access fields and methods of their visitor class, members that cannot be moved to a certain class, like method *getInsertedFigures*. Instead of moving these members, still the visitor classes *InsertIntoDrawingVisitor* and *DeleteFromDrawingVisitor* encapsulate these members and thus became argument containers for *visitInsertInto* and *visitDeleteFrom* respectively.

3.2 Study Results

In the preceding section we exemplified the most common problems encountered when transforming object-oriented pattern implementations into Jak counterparts. In this section we summarize the whole case study and just shortly show interesting problems we found. We had no problems transforming the implementations for Abstract Factory, Command, Facade, Factory Method, Flyweight, Interpreter, Iterator into the Jak counterparts.

The static Bridge implementation (concrete-implementor-classes become refinements of the abstraction class) needed us to inline deep inheritance hierarchies of the concrete-implementor-classes. Additional parameters for the abstraction class's constructors caused problems of parameter passing.

For the static Builder implementation an extra preceding layer defines subclass relationships. This layer also avoids advance notices.

The Chain of Responsibility proposal exposed problems when the chain classes participate a deep inheritance hierarchy. Chain methods looked very diverse, e.g., they defined statements or conditionals before forwarding requests to their successor in the chain. We had to decompose chain methods to execute the decomposed methods of different chain classes homogeneously before and after forwarding the event to the successor inside their refinement.

Generally the implementation proposal of Composite was applicable to JHotDraw. Instead of organizing children of composite objects in hashmaps, we used vectors because orderings of children was important.

The static Jak Decorator design does not deal with deep inheritance hierarchies for decorator *and* decorated classes. When turning decorators into refinements of the decorated class, we had to replicate each decoration refinement (decorator class became refinements of all decorated classes) for different decorated classes. Intermediate classes even caused problems of doubled application of feature code possibly leading to inconsistencies. For illustration we use JHotDraw; in Figure 6a the decorator *BorderDecorator* decorates objects of the classes *TextFigure* and *NodeFigure*. However, refining both classes with the decoration code (Fig. 6b) applies the decoration twice accidentially for objects of type *NodeFigure*. This is because *NodeFigure* inherits the already decorated *TextFigure* methods.

Mediator implementations in JHotDraw do more than just forwarding events to associated objects (as analyzed in [9]), e.g., they construct control panels of a GUI. Separating the mediator classes from the application raised issues of undefined parameter types and undefined return types in the remaining code.

In JHotDraw different Memento objects may be created for an object of a certain type, thus, a single memento-creating method only receiving the originator object as a parameter was not sufficient; also mapping one object to one memento object using a hashmap was insufficient.

The Observer implementation of JHotDraw can be separated as analyzed in [10]. However, reusing an abstract implementation of this prior study was impossible because in JHotDraw observer objects are passed parameters additionally. Worse, based on the event passed, different parameter values are triggered. Therefore, no single observer list together with a single update method suffices as analyzed in [10].

In the Prototype implementation detaching the prototype field from remaining code causes problems of parameter passing for methods (e.g., constructors), that initialize objects and that accept the prototype object as parameter. Factory methods returning a prototype object may – without this pattern – return new objects each time the factory methods are called (not instantiable prototype-types, e.g., interfaces, may prevent that) but that needs user intervention.

In contrast to the Proxy counterpart of [10], single proxy classes in Berkeley DB wrap different classes while other methods of this proxy class do not wrap anything. To wrap different classes by one proxy, proxy classes' methods are static and accept their delegatee as a parameter – this complements the implementation of [10] where one delegatee object for all methods of one proxy object is referenced in a proxy's field. Finally, methods of the proxy class not forwarding anything had no class they wrap and that they could have been moved to – thus, the proxy class remained comprising these methods.

Applying the Singleton decomposition approach of [10] is possible but caused errors. Detaching the Singleton design pattern from classes and apply it on demand, needs this class to be convertable, i.e., the remaining software must not rely on getting the same object everytime an object of this class is requested. In JHotDraw this was not the case, remaining classes relied on this class being a singleton class although the transformation caused no compiler errors.

Different classes that implement the design pattern State in JHotDraw could not be transformed into static refinements of the state-changing class, i.e., these classes cannot be inlined. First, state classes comprise more than just the implementation of state and can be instantiated and used independently of the state-changing class – transforming a state class into a refinement of the state-changing class would preclude this independence. Second, state classes in JHotDraw reference other polymorph state classes making it impossible to inline respective classes (the recursive reference would cause recursive and never-ending inlining).

The dynamic Jak design for Strategy does not pose a problem but the static variant does. The static variant transforms every strategy class into a refinement of the context class and establishes static variability of strategies for the context class. In JHotDraw strategy objects are not bound to one context class only and are stored and managed in container classes like lists – the result of list iterators cannot be calculated statically and so dependent code – chosing the wrong dependent code statically may cause inconsistent programs.

[11] To show that these methods are introduced into the classes using class refinements, we annotate them with a prime.

4. Insights

During this study we learned a lot about design patterns and several myths. For instance, we learned that different implementations *of one pattern* may differ greatly in structure. We realized that complex class hierarchies (e.g., with intermediate instantiable classes) caused problems when transformed to Jak counterparts. In case of Decorator the complex hierarchy of decorated classes even led to a decoration applied twice for certain objects. In contrast to our assumptions we found that our transformations of object-oriented design patterns into Jak counterparts can hardly be reverted (especially when we transformed class associations into mapping relations of hashmaps following [10] or collapsed class hierarchies).

In our observation, static pattern implementations with Jak refinement chains were simpler (linear module organization) than class hierarchies (module organization in two dimensions, inheritance and consultation). However, we could not always retrieve these simple refinement chains when we transformed non-trivial software like JHotDraw (about 30KLOC) or Berkeley DB (about 90KLOC). Major reasons for that were problems of parameter passing and ambiguous instantiation calls.

Finally, we observed that our transformations (e.g., moving or renaming methods and classes and respective calls) were very laborious and tedious. Therefore, we would benefit from a transformation language inside Jak that allows to change a legacy code structure more flexibly. The result of every manual transformation was very inflexible in that it can only be undone manually.

5. Conclusion

In prior research we analyzed design pattern implementations that use advanced modularization mechanisms of Jak. In this work we *transformed* design pattern implementations in programs of larger scale (JHotDraw, Berkeley DB, and Expression Product Line) into Jak counterparts. We found that we could not transform every pattern implementation into a Jak counterpart due to three main reasons. First, one design pattern was implemented differently even in structure for different programs. Second, some of the design pattern implementation approaches lost structural information of the original classes which caused code replication or effort to ensure consistent Jak module composition. Third, information needs expressed as method parameters had to be fulfilled by inappropriate new callers when moving pattern methods from classes to refinements of other classes.

We conjecture that manually transforming design patterns was a very laborious task – when central classes of a software were affected we had to restructure the whole architecture of the programs under study, e.g., JHotDraw or Berkeley DB. Future work is to explore additional code transformation facilities for Jak.

Acknowledgments

We thank Don Batory and Maider Azanza for helpful comments and fruitful discussions. Martin Kuhlemann is supported and partially funded by the *DAAD Doktorandenstipendium* (No. D/07/45661).

References

[1] E. Agerbo and A. Cornils. How to preserve the benefits of design patterns. *ACM SIGPLAN Notices*, 33(10):134–143, 1998.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[3] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[4] H. B. Christensen. Frameworks: Putting design patterns into perspective. *ACM SIGCSE Bulletin*, 36(3):142–145, 2004.

[5] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*. ACM Press/Addison-Wesley Publishing Co., 1995.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] O. Hachani and D. Bardou. On aspect-oriented technology and object-oriented design patterns. In *Workshop on Analysis of Aspect-Oriented Software*, 2003.

[8] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, 2002.

[9] M. Kuhlemann. Design Patterns Revisited. Technical Report 2, School of Computer Science, University of Magdeburg, 2007.

[10] M. Kuhlemann, S. Apel, M. Rosenmüller, and R. E. Lopez-Herrejon. A multiparadigm study of crosscutting modularity in design patterns. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS EUROPE)*, 2008.

[11] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[12] R. E. Lopez-Herrejon, D. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 169–194, 2005.

[13] B. Meyer and K. Arnout. Componentization: The Visitor Example. *IEEE Computer*, 39(7):23–30, 2006.

[14] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443, 1997.

[15] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming*, 5(6):27–41, 1992.

[16] D. Riehle. *Framework Design – A Role Modeling Approach*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2003.

[17] B. Woolf. Null object. In *Pattern Languages of Program Design (PLOPD)*, pages 5–18, 1997.

[18] W. Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design (PLOPD)*, pages 345–364, 1995.