Using Collaborations to Encapsulate Features? An Explorative Study

Martin Kuhlemann, Norbert Siegmund
Faculty of Computer Science
University of Magdeburg
Magdeburg, Germany
{mkuhlema,nsiegmun}@ovgu.de

Sven Apel
Department of Informatics and Mathematics
University of Passau
Passau, Germany
apel@uni-passau.de

***Abstract*—A feature is a program characteristic visible to an end-user. Current research strives to encapsulate the implementation of a feature in a module. Jak is a language extension to Java that allows programmers to encapsulate implementations of features in the form of a collaboration. In prior work, we and others faced problems when using collaborations in Jak and alike languages with too high expectations, e.g., to encapsulate widely scattered code of features such as transaction management in data bases. In this paper, we explore which criteria feature implementations must fulfill so that they can be encapsulated in Jak. The criteria that we found decisive are: *granularity* of code elements that should be encapsulated in a collaboration, *object-level extension* by features, and *object-oriented connections* of a feature's code elements. We finally present a general guideline when to encapsulate a feature with a collaboration in Jak. Practitioners can now evaluate in advance whether Jak collaborations are suited to encapsulate their feature or not.**

I. Introduction

Collaboration-based design (CBD) extends object-oriented design by the concept of a collaboration [1]. A *collaboration* is a module that encapsulates code fragments of an object-oriented program, e.g., statements, members, and classes. A collaboration can be composed flexibly with other collaborations to obtain complete software products. Usually, the code of one collaboration implements one complete feature, i.e., a user-visible program characteristic [2]. By flexibly composing collaborations, stakeholders can select features of a program [1].

Jak is a language that adds the collaboration concept to Java [1]. In prior work, we and others faced problems when we tried to encapsulate features with collaborations in Jak and alike languages, e.g., problems of granularity [3], parameter passing [4], or code replication [5]. Often the problems were encountered too late to make an easy design shift so the developers often refrained from reimplementing the features and instead tried to circumvent the problems (often with bad designs [3], [4]). We argue that developers can benefit from a guideline that describes for which kind of feature Jak collaborations are appropriate and for which features they are not. We propose such a guideline to avoid design flaws and reimplementation effort.

We conducted a case study and then we developed the guideline. In this study, we transformed implementations of a number of features from object-oriented Java code to collaboration-based Jak code. The features we choose were implemented using object-oriented design patterns [6].[1] We choose them because their patterns occur in many places contributing to many feature implementations.

We developed general criteria which a feature must fulfill so that it can be encapsulated properly with a Jak collaboration. The guiding criteria that we found decisive are: *granularity* of code elements related to a feature, *object-level extension* by features, and *object-oriented connections* of features' code elements. Based on these criteria we defined a guideline on how to use Jak-like collaborations. Developers can evaluate *in advance* whether collaborations can encapsulate the feature properly that is to be developed.

II. Background: Collaboration-Based Design with Jak

Jak adds support for collaborations to Java [1]. A collaboration encapsulates a set of classes and class refinements. A *class refinement* encapsulates members, which are added to classes or wrap methods of classes. Wrapping of methods allows programmers to add statements to the beginning and end of methods. The elements encapsulated in a collaboration can be composed with elements of other collaborations. Composing multiple collaborations finally synthesizes a compilable program.

In Figure 1, we show the two collaborations *Base* and *PointObserver*. *Base* contains a class Point. *PointObserver* contains a class refinement, which adds members and statements to the previously added class Point. It adds a field observer and a method setObserver to Point (Lines 7-10). Method setY of refinement Point (in *PointObserver*) refines method setY of class Point of *Base*, i.e., it adds statements to this method with an overriding mechanism (Jak's overriding keyword is Super, Line 12). The refinement adds the subject role of the Observer design pattern to Point.

In our study we will analyze granularity of transformations. For that, we consider member and class introduction as well

[1]Design patterns are descriptions of recurring development tasks and their according standard solutions [6]. A *pattern instance* is code that implements a pattern. Many Jak implementation approaches for patterns were introduced before [7]. However in [7], we did not evaluate Jak for implementing features but just compared mechanisms of Jak-like languages.

Collaboration *Base*

```
1  public class Point {
2    private int y;
3    public void setY(int newY) {
4      this.y=newY;
5    }}
```

Collaboration *PointObserver*

```
6  refines class Point {
7    private IObserver observer;
8    public void setObserver(IObserver newO) {
9      observer=newO;
10   }
11   public void setY(int newY) {
12     Super.setY(newY);
13     observer.update();
14   }}
```

Fig. 1. A sample class refinement in Jak.

as method wrapping *coarse-grained* transformations and transformations targeting statements or parts of statements as *fine-grained*.

III. Case Studies

For different features, we encapsulated code, that implements a feature, into collaborations – each feature we selected for our study is implemented using an isolated pattern. For several features, we also applied collaboration concepts to reimplement the pattern's solution (e.g., of wrapping). We argue that the structure of the studied features reoccurs in the structure of many other features because pattern implementations occur frequently and expose various shapes. If features are implemented without design patterns our criteria and guideline might not match.

Study setup: We encapsulated in collaborations code of features which are implemented by instances of the Gang-of-Four design patterns [6]. Specifically, in the three programs JHotDraw[2], Berkeley DB[3], and Expression Product Line [8] we transformed object-oriented implementations of Gang-of-Four patterns into collaboration-based implementations.[4] JHotDraw (30K lines of code) is a GUI framework; Berkeley DB (90K lines of code) is an embedded database engine for Java; Expression Product Line is an evaluator for mathematical expressions.

A. Results

We summarize interesting problems encountered during our study. We group these problems into categories *granularity*, *data type changes*, *object-level extension*, and *object-oriented connections between code elements*:

Granularity: In JHotDraw, an Adapter pattern instance is used to implement the *Undo* functionality for deleting figures. We analyzed different variants to implement this adapter with

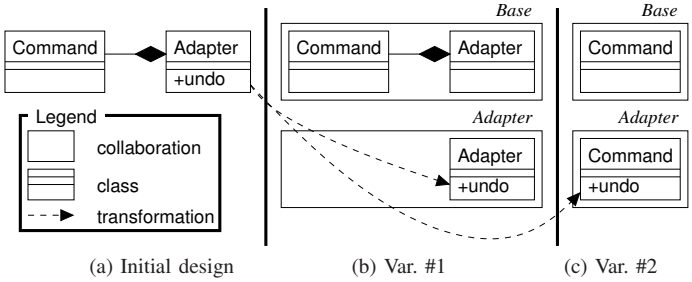


Fig. 2. Transforming the Adapter instance towards collaborations.

collaborations in which adapter methods are added with refinements (1) to an empty adapter class or (2) to the adapted class. With regard to variant #1 (Fig. 2b), we cannot separate calls to adapter methods into collaborations, which already encapsulate the adapter methods, because the calls were located in between other statements or were expressions inside other statements – Jak refinements however only may wrap methods. As a workaround, we added hook methods (or decomposed the methods in other cases) such that we can wrap these new methods. In variant #2 (Fig. 2c), we had to transform adapters that were composed from superclasses. We ended up with either few big collaborations, which replicate code, or with numerous small collaborations with less code replication but complex inter-dependencies.

Jak does not allow fine-grained extensions, e.g., adding formal parameters to methods, method calls to arbitrary methods, or changes to the return type. This caused problems for implementing instances of the patterns Bridge, Mediator, Observer, Prototype, and Template Method and hampered us or prevented us to implement Jak refinements.

Data type changes: To reuse code of an existing implementation of the pattern Composite, we had to change a field's type. Jak does not support such transformation so we changed the code by hand (we changed a field's type from Vector to List). If the transformed code would have been generic [9] our adjustment would have been less problematic.

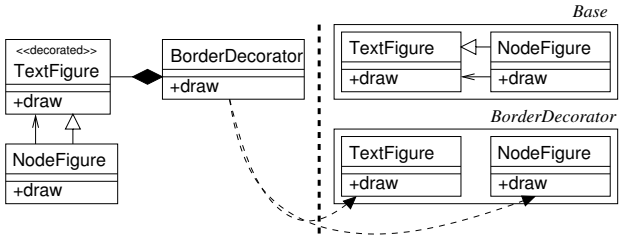
Object-level extension: Strategy and State pattern instances allow developers to choose algorithms for objects of a *context class* [6]. In our study, we planned a design where we choose an algorithm by selecting a collaboration. The selected collaboration then should refine the context classes to add the strategy or state implementation statically to them. Unfortunately, strategy objects are polymorphic and assembled dynamically with delegation. So we cannot design an individual refinement which can be selected statically and which comprises one strategy or state algorithm. Additionally, references to objects of the pattern's classes are set to null and tested for null and code is executed based on this test. These tests require object semantics and cannot be modeled with Jak-like collaborations. Hence, the transformations failed for Strategy and State.

For the pattern instance of Singleton, we faced problems when moving its code into a collaboration (replacing constructors with factory methods that are refined to return the same object at every call). We could not encapsulate all the

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

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

[4]When (a) transforming the whole implementation of a pattern instance includes repetitive tasks and (b) the effort to perform them all (manually) was too high then we concentrated on a sub-implementation requiring these tasks to be performed less often.



(a) Initial design (b) Transformed CBD

Fig. 3. Transformation of a Decorator instance (simplified).

```
1 refines class BinaryTreeLeaf implements VisitableNode {
2   public void accept(Visitor visitor) {
3     visitor.visitLeaf(this);
4   }}
```

(a)

```
1 refines class LineConnection {
2   public void visit(FigureVisitor visitor) {
3     visitor.visitFigure(this);
4   }}
```

(b)

Fig. 4. Name conflict prevents reuse for Visitor code.

feature-related code into the collaboration because we could not determine which code is related to the feature or depends on the pattern instance respectively. In particular, we could not determine execution paths which semantically rely on the pattern instance – a so-called feature mining problem [10].

Object-oriented connections between code elements: The object-oriented implementation of Decorator allows a decorator object to wrap objects of multiple classes which are connected by inheritance. When turning a decorator class into a refinement of a decorated class, we must replicate the refinement for every decorated class that can be instantiated. In the initial object-oriented design (Fig. 3a), objects of BorderDecorator class decorate objects of classes TextFigure *and* NodeFigure which both can be instantiated. However, refining both classes with the decorator's code (Fig. 3b) applies the decoration twice accidentally for objects of NodeFigure. Objects of NodeFigure of which methods are decorated inherit and override methods of TextFigure which are decorated as well. A super call in the method NodeFigure.draw then causes the decoration to execute for TextFigure.draw *and* for NodeFigure.draw.

When we transformed the object-oriented implementation of pattern Proxy, more classes remained than we expected. That is, in the object-oriented design, single proxy objects wrap objects of different classes but some methods of the proxy objects do not wrap anything. Proxy methods that wrapped methods before became refinements of the methods they wrapped. Methods of the proxy class, which did not wrap anything, were not removed from the proxy class. But, as collaborations cannot encapsulate methods outside classes, the proxy classes remained to encapsulate these connected members. Collaboration names are no qualifiers of encapsulated code and cannot be used to reference this code.

For the Visitor pattern in our study we tried to reuse refinements, which add accept methods and visitor classes, from an existing CBD Visitor implementation [7]. However, we failed reusing because methods and classes have incompatible names in JHotDraw and the existing implementation. Specifically, we could not reuse a refinement of class BinaryTreeLeaf (Fig. 4a) but had to implement a similar refinement for class LineConnection (Fig. 4b). We thus had to introduce a kind of code replication.

Other patterns: We were able to transform the implementations of Abstract Factory, Command, Facade, Factory Method, Flyweight, Interpreter, Iterator, and Chain of Responsibility to collaborations without new interesting problems. We faced problems for transforming the Memento instance but they do not allow us to conclude on the expressiveness of Jak [11].

IV. Discussion

In our case studies, we faced more problems to implement or encapsulate the features with Jak than we expected:

Granularity: Most of the transformations were problematic or failed because Jak collaborations could not encapsulate fine-grained code elements, like formal parameters or method calls. Extensions to Jak that support fine-grained changes to code thus appear promising.

Object-level extension: Some patterns target to add a property to individual objects. A Jak-like refinement adds a property to all objects of a class but not just to individual objects. We argue that object-level properties conflict with class-level refinements, and thus CBDs as in Jak inherently cannot encapsulate instances of such patterns.

Object-oriented connections between code elements: We faced problems when different refined classes participate in the same inheritance hierarchy. We also observed a problematic balance between complex collaboration dependencies and code replication. Both are possible fields of future research.

Generally, we found it disturbing that a collaboration name is not a qualifier usable to reference the collaboration's code. To use this code, we put it into publicly accessible classes (which can be referenced), although members in these classes are referenced from within one collaboration only. We argue that there are situations in which implementation details of a separated pattern instance should only be visible within the encapsulating collaboration or visible to features which *explicitly* extend this collaboration.

The transformations that we performed were simple but laborious.[5] We had to implement these transformations manually because there is no sufficient support to restructure and transform code of CBDs.

A. Guideline

Now we give a *conservative* guideline so that developers can evaluate *in advance* whether Jak collaborations are suited to encapsulate a pattern instance. We allow cases in which Jak collaborations could be appropriate although our guideline

[5]For some object-oriented implementations the transformation into collaborations took up to three days (on average it took a few hours) because we had to restructure a major part of the program.

denies this issue, i.e., we allow false negatives. We found that Jak collaborations can encapsulate pattern instances whose implementation (1) is coarse-grained, e.g., whose implementation involves adding just classes and methods, (2) does not extend classes connected through inheritance, and (3) does not apply properties to individual objects of one class only. When pattern instances are known to become fine-grained, known to extend connected classes, or known to extend single objects of one class, Jak collaborations should not be used to encapsulate them.

V. Related Work

Several researchers proposed languages similar to Jak, e.g., [12], [13]. Evaluating these languages is possible future work. We conjecture that the general criteria, which we found, are decisive to implement features in these languages, too.

Kästner et al. and Ye et al. observed that granularity is a decisive criterion when encapsulating features in modules [3], [14]. We confirm these results – we often had to add hook methods to encapsulate statements of method calls. In addition to prior work, we identified the criteria of *object-oriented connections* and *object-level extension* to be decisive for encapsulating features with Jak collaborations.

In another line of research, Kästner analyzed whether AspectJ is suitable for encapsulating features with aspects [15]. In contrast to his work, we concentrated on encapsulating code of various design pattern instances rather than code of self-chosen features. We argue to ensure this way that the features, which we encapsulated, cover the shape of a wide range of features. In addition, we determined general criteria that decide when Jak collaborations can encapsulate a feature.

Several researchers proposed to encapsulate design pattern instances in modules, e.g., [16], [17]. However, they all do not deal with CBDs and they all do not determine which *general* criteria decide language concepts' applicability. In prior work [7], we analyzed aspect-based implementations of patterns [17] and transformed them into Jak implementations. Here, we reuse the Jak implementation approaches (and tried to reuse code) side by side with new approaches in non-trivial programs to develop a general guideline for Jak.

VI. Conclusion

Collaborations are intended to encapsulate features in programs. In a study, we evaluated this aim for a variety of features implemented by design patterns. We found that collaborations can encapsulate some pattern implementations but not every implementation of collaborating program elements can be encapsulated with collaborations as in Jak.

By analyzing the problems we found, we identified criteria which guide whether Jak collaborations are well-suited to implement a particular feature: (1) the *granularity* of the code to encapsulate in a collaboration, (2) *object-level extension* by the feature to encapsulate, and (3) *object-oriented connections* between code elements of the feature to implement. Based on these criteria, we presented a guideline when to use collaborations in Jak-like languages in order to encapsulate a feature.

Acknowledgments

The authors thank Don Batory and Maider Azanza for helpful comments on this work. Martin Kuhlemann was supported and partially funded by the *DAAD Doktorandenstipendium*, number D/07/45661. Norbert Siegmund was funded by the German Ministry of Education and Research (BMBF) in the ViERforES project (http://vierfores.de/), project number 01IM08003C. Sven Apel's work was supported in part by the German Research Foundation (DFG), project number AP 206/2-1.

References

[1] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

[2] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[3] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the International Conference on Software Engineering*, 2008, pp. 311–320.

[4] M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier, "Avoiding variability of method signatures in software product lines: A case study," in *Workshop on Aspect-Oriented Product Line Engineering*, 2007, pp. 20–25.

[5] S. Apel, "The role of features and aspects in software development," Ph.D. dissertation, Faculty of Computer Science, University of Magdeburg, 2007.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[7] 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 Objects, Models, Components, Patterns*, 2008, pp. 121–140.

[8] 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*, 2005, pp. 169–194.

[9] S. Apel, M. Kuhlemann, and T. Leich, "Generic feature modules: Two-staged program customization," in *Proceedings of the International Conference on Software and Data Technologies*, 2006, pp. 127–132.

[10] N. Loughran and A. Rashid, "Mining aspects," in *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2002, pp. 12–18.

[11] M. Kuhlemann, "Transforming object-oriented design pattern structures into layers," Faculty of Computer Science, University of Magdeburg, Tech. Rep. 9, 2008.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of the European Conference on Object-Oriented Programming*, 2001, pp. 327–353.

[13] S. Herrmann, "Object teams: Improving modularity for crosscutting collaborations," in *Proceedings of the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, 2002, pp. 248–264.

[14] P. Ye, X. Peng, Y. Xue, and S. Jarzabek, "A case study of variation mechanism in an industrial product line," in *Proceedings of the International Conference on Software Reuse*, 2009, pp. 126–136.

[15] C. Kästner, "Aspect-oriented refactoring of Berkeley DB," Master's thesis, University of Magdeburg, Germany, Mar. 2007.

[16] B. Meyer and K. Arnout, "Componentization: The Visitor example," *IEEE Computer*, vol. 39, no. 7, pp. 23–30, 2006.

[17] 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*, 2002, pp. 161–173.