Class Composition in FACE, a Framework Adaptive Composition Environment

Theo Dirk Meijler*, Serge Demeyer*, Robert Engel**

(*): University of Berne, Software Composition Group, {meijler, demeyer}@iam.unibe.ch,
(**): Washington University, Computer Science Department, engel@cs.wustl.edu

Abstract

Creating applications using object-oriented frameworks is often difficult, since subclassing plays a too important role. Subclassing is a "white-box" form of reuse, and thus requires the developer to understand the underlying implementation. In the approach described in this paper, class composition is introduced as a form of black-box class reuse. It may be seen to extend the concept of parameterized (generic) classes, especially since it allows mutual relationships as parameters. We illustrate class composition on basis of a design pattern.

1 Introduction

When comparing the development of applications using frameworks [7.] to the development of applications using libraries or from scratch, using a framework is —after a learning period— significantly less labor intensive [12.]. Thus frameworks have a large commercial value.

Still, the use and evolution of a framework has many pitfalls. Realizing a system using a framework often demands the creation of subclasses, which requires a profound knowledge of the implementation of the superclass [8.],[13.] and the way classes are supposed to cooperate. Moreover, the dependency of the subclass from the superclass may lead to the situation where changes in the implementation of the superclass may invalidate the subclass [8.],[13.]. These problems mainly originate from the fact that subclassing is an open, "white-box" form of reuse, meaning that the subclass implementor basically reuses the superclass by extending on its implementation. White-box reuse is often contrasted to black-box reuse, where the developer does not need to know how a piece of software is implemented to reuse it. A well known example of this is object composition. Objects can be reused without knowing how they have been implemented, by specializing them through parameters and links to other objects. Object composition is therefore recommended and applied increasingly [4.] as a reuse form in frameworks. However, since object implementations must still be adapted, class reuse is still required, and therefore subclassing seems still to be needed as well.

Genericity [5.],[1.] is a form of class reuse that *is* black-box. A generic class can already be reused by only parameterizing it, e.g., a class implementing generic list-elements can be reused by parameterizing it with the class name characterizing the objects to be stored in the list. Still, so far genericity has not challenged subclassing as being the main form of class reuse, the reason being that the current use of genericity does not provide enough flexibility.

FACE is a “Framework Adaptive Composition Environment” which main intention is to simplify framework usage. We have developed a form of black-box class reuse that extends on genericity, while totally hiding class implementation. The main extension of genericity is that relations between classes can be used to specialize those classes. We do not principally restrict what these relationships may mean. The idea is, that the application developer can just parameterize and compose classes in order to achieve the class specialization necessary within the Framework. Since the classes hide their implementation we in fact call them “class components”. That application development in this way is possible is a result of the fact that the Framework imposes the class specializations that make sense anyhow, and thus the fully unconstrained form of specialization through subclassing is not needed.

Design patterns [4.] are patterns of class cooperations that have proven themselves to be viable solutions to frequently occurring problems in framework design. Since the application of design patterns therefore forms an important part of the design of a good framework, it is a logical step to illustrate the feasibility of the FACE approach by applying it to individual patterns. An applied (but isolated) design pattern is seen as a kind of “mini-framework” in which the major aspects of the FACE approach may be shown as we do in this paper. Especially, we show that in order to have black-box class composition one needs to have various “families” of class components, each of them having their own family-specific specialization interface. In section 2 we shall (roughly) show how the isolated application of a pattern may indeed be mapped to a class composition, where for each of the roles in the pattern there is a corresponding family of class components with a corresponding specialization interface.

In order to realize a framework that allows for application development in this way, one needs to define the various families of class components and define their specialization interfaces. This includes rules describing whether members of which class component families may be linked to members of which other families. Moreover, it involves, next to implementing the “base” functionality, implementing how the run-time system behavior depends on the information in the class composition. In section 3, we shall show how a separate layer of meta-classes is used to define the various families needed in the class composition for the pattern, and their composition rules. It is this layer which makes FACE “framework adaptive”. In section 4 we shortly mention how information in the class composition gets run-time meaning. In section 5 we give some related work, in Section 6 we conclude.

2 The “Abstract Factory” Design pattern as a Class Composition in FACE

Figure 1 shows a class composition in the FACE approach, that realizes a specific use

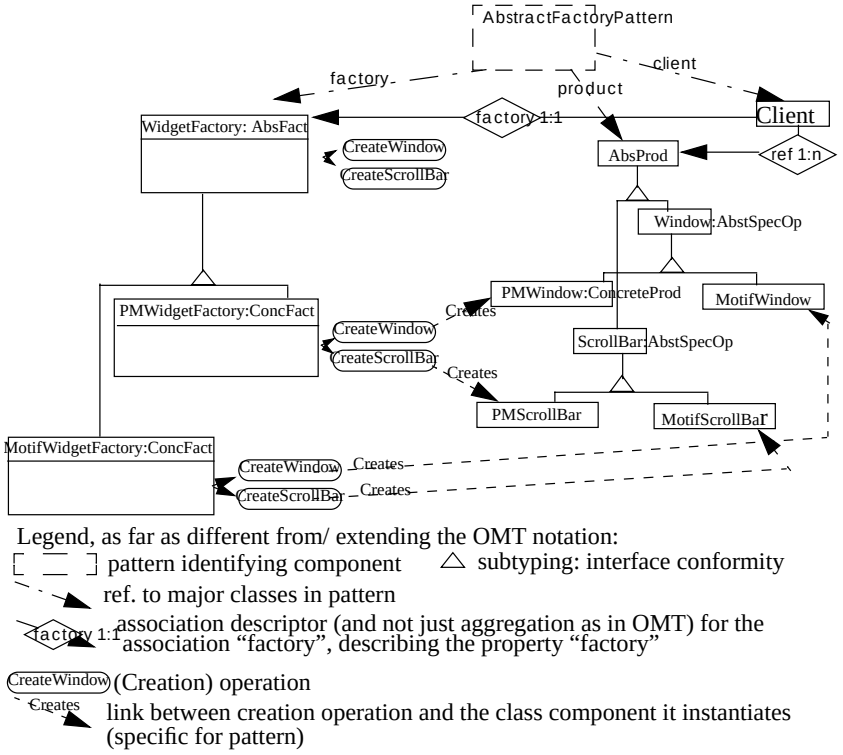

Figure 1. A class composition representing a typical usage of the AbstractFactory pattern. Note that the name of the family of the class component, e.g. ConcFact for PMWidgetFactory, could not be given everywhere.

of the abstract factory pattern containing a factory for creating user interface widgets and windows. Before going into an explanation of what such a composition means, a short explanation of the pattern: The Abstract Factory design pattern is used when client objects want to create certain objects (e.g., window objects, scrollbar objects) but the instantiation of these objects should not commit the client to choosing a specific implementation for these objects (e.g., either Presentation Manager Window or Motif Window). The basic idea of the pattern is to delegate the creation to a special factory object upon a client request. By using polymorphism, the factory object’s class determines how the factory object reacts to a request for a certain object. E.g., when re-

quested to create a window a MotifWidgetFactory object will return a Motif window, a PMWidgetFactory a PMWindow.

Note that for each of the different roles in the class composition above, (either client, factory, or product) there exists a different “family” of special-purpose class components, having each its own specialization interface. In the figure, the name of this family is given after the “:” in the name of the class component. We highlight an important part of the class composition for the applied pattern. A component of the type “ConcFact”, is specialized by the set of “Create” operations that it has, e.g., “CreateWindow” and “CreateScrollbar”. Moreover, each of these operations is specialized by a link (relationship) to the product class that it creates. Such explicit relationships between components in the schema thus replace the “traditional” form of specialization where factory classes are specialized by writing these create methods.

3 Configuring FACE by means of Meta-classes

To make class compositions as shown in section 2 possible, the various class component families must be defined together with their specialization interface. In Figure 2 below, we show how each of the role-specific families of class components can be defined by so-called meta-class components: class components of which the instances are class components. The figure also illustrates how the specialization interface can be defined for the instances of meta-class components through so-called association descriptors that link those meta-class components. For example, the association descriptor “creates” between “ConcrCreateOp” and “ConcreteProd” shows how a create operation (a component that is an instance of “ConcrCreateOp”) can have a link to the product class (an instance of “ConcreteProd”) it creates.

Note that the meta-classes defining the various families of possible class components are themselves not just ordinary classes, but also class *components*: They have a black-box specialization interface in the sense that they can only be specialized by means of association descriptors. In this way the framework developer who creates such a meta-class composition has the same advantages, namely black-box composition, in creating this composition as the developer of an application has in creating a class composition.

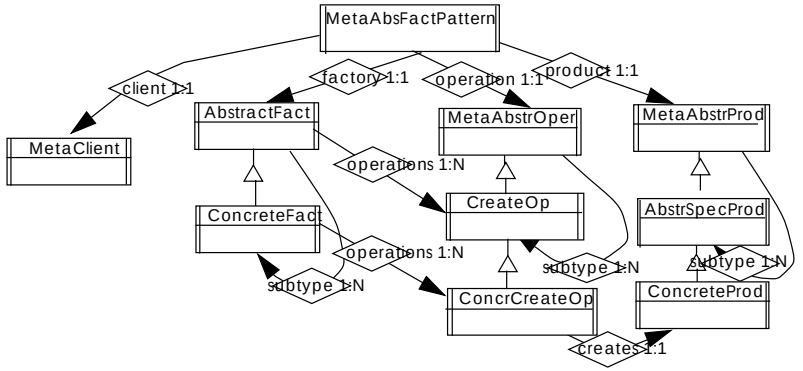

Figure 2. Composition of meta-class components that define the families of class components used in Fig 1. and the way these may be composed. Metaclass components are indicated with double lines at the sides. These define classes of class components, e.g., all Concrete Widget class components in Figure 1are instance of “ConcreteFact”. Association descriptors are indicated by diamonds.

4 Giving run-time meaning to a class composition

To give a class composition such as shown in Figure 1 a run-time meaning, our current approach is to interpret[1] the information in the composition. The class composition is represented explicitly at run-time as objects that represent the class components and relationships. This is quite natural, since the composition will normally be created interactively as an object structure by the application developer anyway. The class components are thus run-time represented as objects but they function as classes, in the sense that they can be requested for instances. When requested for instances, they (e.g., by copying a prototype that they carry, or calling a constructor), will return an instance object. The instance objects will have a generic implementation, implemented in the “underlying” object-oriented language[2]. The implementation is generic in the sense that these instance objects will query the objects representing the class composition to adapt their behavior to the parameters.

1. Another possibility could be to generate “real” classes of some existing object-oriented programming language.
2. which can be anything: so far, implementations have been done in Self and C++

5 Related work

Although this paper is not meant to give a full overview of related work, we mention the most influential sources of inspiration:

- Composition at the level of classes is especially encountered in “generic” constructs, where classes can be parameterized with other classes, such as in the template classes in C++ [9.]. Also the work of Batory et. al., [2.], is based on parameterizing class level components with class level components. Our work enhances the scope of possible relationships between classes thus allowing for more expressivity.
- The commercial work on componentware [14.], such as Visual Basic [11.] Our work gives application of components a much greater scope, since we unify work on components with work on object-oriented frameworks.
- Work on meta object protocols [6.], where the correspondence between meta-classes and classes is similar as between meta-class components and class components in our model.

6 Conclusion

Black-box reuse is a form of reuse where the person who reuses does not need to know the internals, the implementation, of a software component in order to reuse it. This form of reuse is often connected to object composition, where the encapsulation of objects allows them to be reused by parameterization and composition only. This is contrasted to white-box reuse, where in order to reuse, implementation needs to be known and understood, which makes it more difficult and error prone. This is encountered in the most common used form of class reuse: subclassing. Subclassing is not only white-box, it is also very general, any kind of specialization is possible. This while a framework very often imposes specific specializations that “make sense”. We therefore extend on the other already known form of class reuse, which does not have these problems: genericity. Especially, the parameterization of classes (“class components”) with meaningful relationships between them is a new form of genericity. In the paper we have illustrated this approach by treating a specific application of the “Abstract Factory” design pattern as a mini-framework. We have shown how important aspects for specializing the applied pattern, e.g., which concrete factory class instantiates which concrete product, can be specified by composition only.

How this approach scales up for large frameworks is outside the scope of this paper. We are doing, and have done [10.] work on this. A major issue is whether the approach indeed provides the developer with enough flexibility and expressivity. We believe that this is so for the following reasons:

1. In this paper we have presented forms of class parameterization with relative little information contents. The approach allows for more complex forms of parameterization, with as the most general, but least controllable possibility the parameterization with procedures.
2. Our approach is (therefore) related to existing “componentware” approaches [11.] These have turned out to be not only useful, but very successful.
3. Use of this extended form of genericity by the application developer does not preclude the use of “old-fashioned” subclassing by the framework developer to introduce new kinds of parameterizable class components. Moreover, when there is a need for adapting the composition and parameterization possibilities of the application developer, this can be done by adapting the meta-class composition, and correspondingly adapting the run-time meaning. The latter is also done in “normal” object-oriented programming which does not create a bigger obstacle than “traditional” framework extension.

References

1. American National Standards Institute, Inc., *The Programming Language Ada Reference Manual*, G. Goos and J. Hartmanis (Ed.), LNCS 155, Springer-Verlag, 1983.
2. Don Batory and Sean O’Malley, “The Design and Implementation of Hierarchical Software Systems With Reusable Components,” *ACM Transactions on Software Engineering. and Methodology*, October 1992.
3. Grady Booch and James Rumbaugh, *Unified Method for Object-Oriented Development Version 0.8*, Rational Software Corporation, 1995.
4. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.
5. Joseph A. Goguen, *Principles of Parameterized Programming*, In T.J. Biggerstaff and A.J. Perlis (Ed.), Software Reusability vol. I, ACM Press & Addison-Wesley, Reading, Mass., 1989, pp. 159-225.
6. Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press (Ed.), 1991.
7. Ralph E. Johnson and Brian Foote, ‘‘Designing Reusable Classes,’’ *Journal of Object-Oriented Programming*, vol. 1, no. 2, 1988, pp. 22-35.
8. John Lamping, ‘‘Typing the Specialization Interface,’’ *Proceedings OOPSLA ’93, ACM SIGPLAN Notices*, vol. 28, no. 10, Oct. 1993, pp. 201-214.
9. Simon Lippman, *The C++ Primer, Second Edition*, Addison-Wesley, 1991, (3).
10. Theo Dirk Meijler, ‘‘User-level Integration of Data and Operation Resources by means of a Self-descriptive Data Model,’’ Ph.D. Thesis, Erasmus University Rotterdam, Sept. 1993.
11. Microsoft Corporation, *Visual Basic Programmer’s Guide*, 1993.
12. Simon Moser and Oscar Nierstrasz, “Measuring the Effects of Object-Oriented Frameworks on the Software Process,” submitted for publication, IAM, U. Berne, Dec. 1995.
13. Patrick Steyaert, Carine Lucas, Kim Mens and Theo D’Hondt, “Reuse Contracts: Managing the Evolution of Reusable Assets,” *Proceedings OOPSLA’96, ACM SIGPLAN Notices*.
14. James Udell, “Componentware,” *Byte*, vol. 19, no. 5, May 1994, pp. 46-56.