Supporting Extension of Components with new Paradigms

Lutz Dominick
Siemens AG, Corporate Technology,
ZT SE 2
D-81730 Munich, Germany
Lutz.Dominick@mchp.siemens.de

Klaus Ostermann
Universität Bonn
Institut für Informatik III
Römerstr. 164, D-53117 Bonn, Germany
osterman@cs.uni-bonn.de

August 2000
Publication granted for purpose of OOPSLA 2000

Abstract

Component-based applications require more than simply components. They extend and configure components and "glue" them together to construct an application. Requirements like extensibility and configurability become mission critical. Traditional solutions in this area rely on software design patterns and have been successfully employed for a large inhouse project at Siemens. This paper examines the influence of AspectJ and Hyper/J in this field and compares them to conventional solutions. The result indicates that especially AspectJ has major limitations, which we think is due to missing theoretical foundation in the areas of type theory and integration with existing approaches.

Introduction

Large applications at Siemens and other companies are more and more based on components. On first glance, components look like perfect reusable assets. However, even if they are domain specific, they remain generalized software solutions and therefore need to be adapted to deployment-specific needs. For this reason, it is essential that components be easily extensible, parameterizable and configurable.
In a large inhouse project at Siemens, extensible components have been designed and employed for the past few years. Based on regular component technology, software architecture based on design patterns and frameworks has been used to fulfil – as far as possible – the additional requirements.
The following concerns, which are independent of the functional requirements of the application, have proved especially important for extensible and configurable components:

- **Unanticipated extensions**: The employment of design patterns [BMR+96, GHJV94, Stal00] usually helps to add variation points to a system. Although techniques like feature modelling [EC00] help to avoid unused or missing variation points, not all variation points can be anticipated. Therefore, we also need mechanisms for unanticipated extensions of a component.
- **Extensible and reusable extensions**: To master the complexity of large component systems, a component architecture needs to be scalable. Although components are self-contained, they often need multiple extensions to provide the required functionality. For example, Szyperski proposes a tiered component architecture that arranges an open set of component frameworks [Szy97]. The idea is that components can be plugged into component frameworks (i.e., are extensions of the component framework), which are themselves components and can be recursively plugged into second-order frameworks etc. We conclude that extensions to components should also have component-like properties. In particular, extensions should be *extensible* and *reusable*.

- **Dynamic and static binding**: Extensions may be bound to their base either at compile-time (*static binding*) or at runtime (*dynamic binding*). The former case allows for sophisticated compile-time checks while the latter enables more flexibility.
- **In-place and additive deployment**: An important point is the deployment of extensions. Should extensions be deployed *in-place*, i.e. the behavior of the existing component is altered and all existing users of the component participate in the extensions, or should they be deployed *additively,* i.e. a new component is created that combines the base with its extensions. For in-place extensions, a generalized Liskov Substitution Principle [Lis87] is mandatory: The augmented component has to behave like the original one because otherwise existing clients may break. The same kind of substitutability is also desirable for additive deployment because it enables a combination of the extended component with clients of the base component.
- **Modular checking**: Components come from independent sources and are used by third parties. This key concern of component technology [Szy97] has a number of severe implications. For example, integration testing is no longer feasible because the component vendor does not know the environments in which the component will be used. For this reason, a component should allow *modular checking,* e.g. separate compilation and specification of component properties only in terms of the component and the enviroment constraints it builds on.

Of course, the above list is not complete and it cannot be used as a "benchmark" for the different paradigms. Each paradigm has important properties that are not reflected in any of these notions. Nevertheless, we think that it is interesting to see how these notions can be mapped to the different extension proposals.

Extension and Configuration Paradigms

In this section, we shortly discuss each paradigm, but the focus is on AspectJ and Hyper/J. We present a summary table at the end.

- **Strategy and Observer Pattern**

These patterns are representatives for anticipated extensions. For example, the same patterns can also be applied to a certain strategy so that extensibility and reusability of the extensions are guaranteed. The binding of a base to its extension is dynamic, based on the usual subtype polymorphism. Extensions are deployed in-place, because existing instances are directly augmented. Modular checking is not an issue.

- **Decorator and Proxy Pattern**

Patterns like Decorator and Proxy allow for unanticipated extensions. Polymorphism guarantees that extensions are reusable. For example, a decorator can be used with different base classes for the "decoratee". The deployment is additive because existing references to the base are not influenced. The binding is dynamic and modular checking is no problem.

- **Inheritance**

Inheritance allows for unanticipated extensions of a base class. Inheritance is orthogonal to the base code, i.e. subclasses can be subclassed again. The reusability of extensions (the subclasses) is difficult because subclasses are fixed to their superclass and the binding is static. It is interesting to note that both concerns are addressed by recent proposals, e.g. MixedJava [FKF98] adresses reusability of subclasses and Darwin [Kni00] enables dynamic inheritance. The deployment of extensions is additive and modular checking is no issue.

- **AspectJ**

Aspect-Oriented Programming (AOP) offers *aspects* as new assets to capture properties of the system in one place that actutally crosscut several architectural artifacts simultaneously. A tool called AspectJ™ [AJ00] distributes the adherent code fragments at compile time. Aspects declare *pointcuts* and *advices*. *Pointcuts* indicate where in the base code the extension will take place. *Advices* contain the extension code itself.
AspectJ allows for a wide range of unanticipated extensions by means of advices and introductions that can be attached to a class.
Multiple aspects can be applied to the same or different classes of components and their extensions. But this does not mean that the aspects have extended each other. If an aspect extends another aspect (that contains an extension to the component), the first requires also access to the second aspect's *advice* statements, but this is not possible.
Reuse of extensions is possible by means of *abstract pointcuts* that have to be redefined in sub-aspects. This creates an inheritance hierarchy level within the set of the project's aspect classes. The disadvantage, however, is that reuse of aspect code has to be anticipated because usual (non-abstract) pointcuts cannot be redefined.
Another language shortcoming is also related to reuse. An *advice* cannot get attached to multiple classes without excessive duplication of aspect code because generic access to the target class instance is not provided any more (the former thisObject keyword*).*
The AspectJ™ weaver changes one or more of the component classes so deployment is in-place. The binding is static at the moment (dynamic binding of aspects to classes has been dropped in the recent AspectJ version).
AspectJ has also limitations concerning modular checking: Neither aspects nor classes can be compiled separately. A class or clients of that class may even rely on introductions that are defined in an extending aspect. A class can only be checked and compiled with a global view on all aspects that may influence the class.

- **Hyper/J**

Hyper/J is a tool for multi-dimensional separation of concerns [TO99]. In Hyper/J, a set of *hyperslices* is combined to a *hypermodule*. Composition with Hyper/J does not have to be anticipated. Each hypermodule can be used as a hyperslice in another hypermodule (extensibility of extensions) and each hyperslice can be used in multiple hypermodules (reusability of extensions). Like inheritance, binding in Hyper/J is static and deployment is additive. Modular checking is possible because each hyperslice has to be *declaratively complete* which means that it has to declare everything to which it refers.

The results of our investigations are summarized in the following table:

	Observer Strategy	Decorator Proxy	Inheritance	AspectJ	Hyper/J
Unanticipated extensions	No	Yes	Yes	Yes	Yes
Extensibility of extensions	Yes	Yes	Yes	No	Yes
Reusability of extensions	Good	Good	Bad	Limited	Good
Binding	Dyn.	Dyn.	Stat.	Stat.	Stat.
Deployment	In-place	Additive	Additive	In-place	Additive
Modular Checking	Yes	Yes	Yes	No	Yes

Conclusion

New paradigms and their tools provide a rich set of new features and capabilities to include anticipated and unanticipated variability within projects. Although our list of component requirements does not reflect all strengths and limitations of the paradigms, it exhibited major weaknesses. Among the different paradigms, AspectJ especially seems to have important shortcomings.
Taking the large complexity of these languages into account, we propose that considerable efforts have to be spent for work on the theoretical foundation for the required solutions. For example, the type system and its notion of substitutability are largely untouched in AspectJ and Hyper/J, although a refined type system might provide substantial benefits. There are also new mathematical approaches available such as the design algebra in [Tek00] that is based on vectors and matrices and where extensibility forms a conceptual dimension of a component.
Software engineering has made incredible progress during the past decades. As we feel, current work in the area of new paradigms is an important step forward that needs to be sustained by additional theoretical work, also for seamless integration with best practice approaches.

References

[AJ00] AspectJ™ home page http://aspectj.org, AspectJ™ version 0.7beta5, Aug 2000

[BMR+96] F.Buschmann, R.Meunier, H.Rohnert, P.Sommerlad, M.Stal, "Pattern-Oriented Software Architecture"; Wiley & Sons, 1996

[EC00] Ulrich W. Eisenecker and Krzysztof Carnecki, “Generative Programming”, Addision Wesley 2000

[FKF98] Flatt, Krishnamurthi, Felleisen, “Classes and mixins”, Proceedings POPL ‘98, 1998

[GHJV94] E.Gamma, R.Helm, R.Johnson, J.Vlissides, "Design Patterns", Addison Wesley 1994

[Kni00] Günter Kniesel, “Dynamic Object-Based Inheritance with Subtyping”, PhD Thesis, Dept. of Computer Science, University of Bonn, 2000

[Lis87] B.Liskov, "Data Abstraction and Hierarchy", OOPSLA 1987, Addendum to Proceedings, Oct. 1987

[Szy97] Clemens Szyperski, “Component Software”, Addison Wesley 1997

[Stal00] Michael Stal, "Distributed Objects from a Patttern Perspective", Conference talk, Microsoft ADC 1999

[TO99] Peri Tarr, Harold Ossher, “Hyper/J user and installation manual”, http://www.research.ibm.com/hyperspace/, 1999

[Tek00] Bedir Tekinerdogan, "Synthesis-Based Software Architecture Design", PhD Thesis, Dept. of Computer Science, University of Twente, 2000