Type-Safe Delegation for Dynamic Component Adaptation

Günter Kniesel

University of Bonn
gk@cs.uni-bonn.de, http://javalab.cs.uni-bonn.de/research/darwin/

The aim of component technology is the replacement of large monolithic applications with sets of smaller components whose particular functionality and interoperation can be adapted to users' needs. However, the adaptation mechanisms of component software are still limited. Current proposals concentrate on adaptations that can be achieved either at compile time or at link time ([1], [2]). There is no support for *dynamic component adaptation*, i.e. unanticipated, incremental modifications of a component system at run-time. This is especially regrettable since systems that must always be operational would profit most from the ability to be structured into small interchangeable components that could evolve independently and whose functionality could be adapted dynamically.

Existing component adaptation techniques are based on the replacement of one component by a modified version. This approach is unapplicable to dynamic adaptation: at run-time components cannot simply be replaced or modified because their "old" version might still be required by some other parts of the system. Thus we are faced with the problem to change their behaviour solely by adding more components.

This problem has two aspects. On one hand, the new components must be used instead of the old ones by those parts of the system that should perceive the new behaviour. This requires the component infrastructure to allow "re-wiring", i.e. dynamic modification of the information and event flow between components. On the other hand, the new and the old component must work together "as one". One reason might be that both have to manage common data in a consistent fashion. Another reason arises from the initial motivation of component-oriented programming, incrementality: the new component should not duplicate functionality of the old one. Thus there must be some way for the new component to "inherit" all unmodified behaviour but substitute its own behaviour where appropriate.

In traditional, statically typed, class-based object models, where component interaction at run-time is solely based on message sending, this is impossible to achieve without compromising reuse ([1]). An interesting alternative is the concept known as *delegation* ([5]). An object, called the *child*, may have references to other objects, called its *parents*. Messages for which the message receiver has no matching method are automatically forwarded to its parents after binding their implicit *self* parameter to the message receiver. Thus, all subsequent messages to *self* will be addressed to the message receiver, allowing it to substitute its own behaviour for parts of the inherited one.

Many authors have acknowledged the modelling power and elegance of delegation but at the same time criticised the lack of a static type system that made delegation incompatible with traditional object models. It is the main achievement of DARWIN ([3]) to have shown that type-safe dynamic delegation with subtyping *is* possible and *can*

be integrated into a class-based environment. Compared to composition based only on message sending, delegation in the DARWIN model is easy and results in more reusable designs because

- it requires minimal coding effort (addition of a keyword to a variable)
- it introduces no dependencies between "parent" and "child" classes, allowing parent classes to be reused in unanticipated ways without fear of semantic conflicts and child classes to adapt themselves automatically to extensions of parent types (no "syntactic fragile parent class problem").

In the context of component-oriented programming, type-safe delegation enables extension and modification (overriding) of a parent component's behavior. Each extension is encapsulated in a separate component instance that can be addressed and reused independently. Delegating child components can be transparently used in any place where parent components are expected.

Unlike previous approaches, which irrecoverably destroy the old version of a component, delegation enables two types of component modifications.

Additive modifications are the product of a series of modifications, each applied to the result of a previous one. They are enabled by the recursive nature of delegation: each new "extension component" can delegate to the previous extension. Additive modifications meet the requirement that the result of compositions / adaptations should itself be composable / adaptable.

Disjunctive modifications are applied independently to the same original component. They can be implemented as different "extension components" that delegate to the same parent component. Disjunctive extensions are most useful in modeling components that need to present different interfaces to different clients.

A sketch of DARWIN and a detailed description of the way in which it supports independent extensibility of components and dynamic component adaptation is contained in [4].

References

1. Harrison, William and Ossher, Harold and Tarr, Peter. Using Delegation for Software and Subject Composition. Research Report RC 20946 (922722), IBM Research Division, T.J. Watson Research Center, Aug 1997.
2. Keller, Ralph and Hölzle, Urs. Supporting the Integration and Evolution of Components Through Binary Component Adaptation. Technical Report TRCS97-15, University of California at Santa Barbara, September 1997.
3. Kniesel, Günter. *Darwin - Dynamic Object-Based Inheritance with Subtyping*. Ph.D. thesis (forthcoming), University of Bonn, 1998.
4. Kniesel, Günter. Type-Safe Delegation for Dynamic Component Adaptation. In Weck, Wolfgang and Bosch, Jan and Szyperski, Clemens, editor, *Proceedings of the Third International Workshop on Component-Oriented Programming (WCOP '98)*. Turku Center for Computer Science, Turku, Finland, 1998.
5. Lieberman, Henry. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 21(11):214–223, 1986.