SLE/AOSD Paper

No Author Given

No Institute Given

1 Introduction

Software evolution requires integration of new components with the old ones. In an ideal component-based system, this integration should be seamless meaning the legacy components remain untouched and the interface of the new component is fully compatible with the existing interfaces. Unfortunately such systems do not exist; as a result the integration is seldom seamless. Throughout the paper we will refer to the integration of two software components as a *binding*. We have defined three major challenges regarding binding.

1. When components evolve, the links between them must be re-established.
2. When adding unforeseen functionality to a system, no explicit hooks exist for attaching the new component. In this case it may be necessary to modify code to make the binding, which will create cluttered code.
3. There may be structures in the new and the old component, which conceptually overlap but provide different interfaces. The mapping between these structures should be defined, in order to avoid inconsistencies.

Handling the first challenge requires a *maintainable* way of expressing binding. It is possible to program binding according to some foreseeable evolution scenarios. However in today's component-based systems, third-party software is widely used. So when the interface offered by a third-party software changes, it is necessary to re-program the binding. *Reusable binding structures* and expressing binding in a *concise* manner is then valuable to reduce this programming effort.

Solving the second challenge requires a means to expose certain information in an application's control flow and inject additional behavior to the control flow. Context exposure can be done via object-oriented programming(OOP), by providing classes which store and expose context information. Injecting additional behavior can be achieved via design patterns like dependency injection or decorator. However these methods are valid for planned extensions, and they will not be sufficient when a new component needs to access an unexposed context. Another issue is linked to the nature of binding two components. Since components need to be connected through possibly multiple points the **binding concern** becomes cross-cutting. It has been shown that OOP is not effective in modularizing such cross-cutting concerns.

Solving the third challenge requires a means to express the mapping between components. **This is going to be explained after AOP and how it offers solution to the first two challenges is mentioned.**

Aspect Oriented Programming (AOP) is designed to modularize crosscutting concerns and with its 'weaving' mechanism, it is possible to change the behavior (advice weaving) or the structure (AspectJ:inter-type declarations, CaesarJ: family polymorphism, wrappers) of an implementation without altering the implementation itself. These properties of AOP make it a desirable candidate for modularizing the binding concern.

AOP is effective for achieving loose-coupling. It can capture information or inject behavior from a component without being acknowledged. AOP also facilitates the OO way of loose-coupling, which done via interfaces. It is possible to declare subtypes of an interface and provide the subsequent implementation in an aspect. AOP is also efficient in localizing a concern. So when two components are bound using AOP, the binding implementation will be in one place. This is an important property for maintaining the modules providing loose-coupling.

A pointcut is a program construct that selects join points and expose context at those points **(AspectJ in Action book)**. Hence by using pointcuts we can define entry points to a system to inject new behavior. Of course this approach is limited with the expressiveness of the join-point model of the AO language. But this is a potential AO solution for the second challenge.

The third challenge can also be addressed by AOP. However a solution in current AO languages have drawbacks. **(I think here we should really discuss JasCo)**. Let us look at a simple example and evaluate the possible solutions in AspectJ and CaesarJ.