SLE2012

Kardelen Hatun, Christoph Bockisch, and Mehmet Akşit

TRESE, University of Twente
7500AE Enschede
The Netherlands
http://www.utwente.nl/ewi/trese/
{hatunk,c.m.bockisch,aksit}@ewi.utwente.nl

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.
3. The interface of the components is not compatible and they should be adapted.

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.

Handling 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.

Handling the third challenge requires a means to express the mapping between components. In this paper we present new language mechanisms to express such mappings and provide improvement on the solutions of the first two

challenges. Our approach consists of two language concepts that work together; *instance selectors* and *adapter declarations*.

In the original GoF book, Adapter Pattern's purpose is defined as "converting the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces". Adapters are an important part of a component-based system, since they are the building blocks of binding.

Our approach is designed as an addition to Aspect-Oriented Programming (AOP)(**?**). AOP is used to modularize crosscutting concerns and with its 'weaving' mechanism, it is possible to change the behavior or the structure 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.

However implementation of adapters in current AOP approaches is type invasive. In AspectJ Adapter Pattern is implemented via inter-type declarations which alter the type system by making the adaptee implement a certain interface. This changes the type hierarchy of the adaptee. Also the Adapter Pattern disappears into the AspectJ syntax, which diminishes the visibility of the pattern. CaesarJ has an explicit syntax for defining adapters, which are referred to as wrappers in CaesarJ. However CeaserJ, although it gives great power over separation of concerns, tend to become too fragmented, hurting maintainability.

2 The Binding Language

2.1 Instance Selectors / Pointcuts

2.2 Adapter Declarations

3 Comparative Evaluation

This section will include an example and compare solutions in AspectBind, AspectJ and CaesarJ.

4 Related Work