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

Component interact through their interfaces. In component-based systems, there are cases where a certain functionality is provided by a third-party software which usually comes with an incompatible interface, which makes connecting new components and the legacy components a challenge. For example the legacy component has a private variable for which it does not provide setters and getters. Then accesing this variable requires either making it public or modifying the legacy interface to offer setters and getters for it. Such changes may cause undesired effects on the system, like a security violation due to changed visibility. So an important requirement of *binding* two components is to encapsulate the binding declarations in a separate module, while leaving the bound components untouched. In this study we present a language which satisfies this requirement and offers a reusable, maintainable and concise way of expressing binding.

Our language is composed of two main structures; **instance pointcuts** and **adapter declarations**. Instance pointcuts are specialized pointcuts, which are used to capture the instances of a type, which at some point in their life cycle become relevant. This can be creation of the instance, calling a certain method on the instance or passing the instance as an argument. The conditions for becoming relevant is defined in the pointcut expression. Adapter declarations provide a declarative syntax for implementing object adapters, where the objects to be adapted are selected by instance pointcuts. The benefit of our approach partly comes from utilization of Aspect Orientation (AO), which allows modularization of binding concern. But the major benefit is due to the marriage of two new language concepts which take object adapters to a new level.

The traditional object adapter is shown in Figure *(objectadapter)*. Instance pointcuts provide means to select a subset of instances that belong to a specific type. It is also possible to select all subtype instances of a supertype, since the instance pointcut captures dynamic types. In adapter declarations this subset can be adapted by referring to the instance pointcut. An adapter declaration consists of a unique name, an instance pointcut reference and a list of interfaces to be implemented. The interface methods are overriden in the adapter declaration, referring to its *adaptee* where adaptee is an instance belonging to the set of instances selected by the instance pointcut. When an instance pointcut expression matches a join-point and that instance pointcut is referred to by an adapter

declaration, then an adapter instance is automatically created containing the instance in the matched join-point. In an OO approach this would require adapter instantiations at various points in the code, which will cause tangling of adapter instantiation concern.

It is also possible to have an inheritance hierarchy among adapters, by defining *abstract adapters*. Abstract adapters do not have to implement all the interfaces they declare, whereas *concrete adapters* have to provide an implementation for every interface they declare or for the unimplemented interface declared by their super-adapter. Concrete adapters can also override the implementations of their super-adapters. This abstraction mechanisms leads to maintainable adapters and reduce programming efforts should the components evolve.

The AO nature of our approach allows all of these features to be encapsulated in an aspect. The concern is localized therefore there's no scattering and tangling. Moreover a run-time library for querying and retrieving adapters is created. With this library the user can query adapters by using an instance, a type, an adapter name or a combination of these as a key. This allows accessing adapters in a flexible way, whereas in the traditional OO implementation, it is only possible to access an adapter is by using its instance name.

2 The Binding Language

Instance pointcuts and adapter declarations are language concepts and ideally every aspect-oriented language can be extended to host them. We will discuss possible extentions in other well known AO-languages at the end of this section. We have implemented a prototype in AspectJ. While we have extended the syntax of AspectJ for instance pointcuts and adapter declarations, we have reused AspectJ's primitive pointcuts for the instance pointcut expression.

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