Aspect-Oriented Language Mechanisms for Component Binding

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

Abstract. Domain Specific Languages (DSLs) are programming languages customized for a problem/solution domain, which allow development of software modules in high-level specifications. Code generation is a common practice for making DSL programs executable: A DSL specification is transformed to a functionally equivalent GPL (general-purpose programing language) representation. Integrating the module generated from a DSL specification to a base system poses a challenge, especially in a case where the DSL and the base system are developed independently. In this paper we describe the problem of integrating domain-specific modules to a system non-intrusively and promote loose coupling between these to allow software evolution. We present our on-going work on aspect-oriented language mechanisms for defining object selectors and object adapters as a solution to this problem.

1 Introduction

Complex systems are created by assembling software components of various types and functions. Reuse is essential and components created for a system are required to continue working after the system has evolved. Some components may be domain-specific, meaning their structure and functionality can be defined using the fundamental concepts of the relevant domains. A domain-specific language (DSL) provides expressive power over a particular domain. It allows software development with high-level specifications; if general-purpose programming languages are used, development may take a considerable programming effort.

The specifications written in a DSL can be processed in various ways. These are comprehensively described in [4] and [3]. Generative programming [2] is one of the processing options and has become highly popular with the emergence of user-friendly language workbenches. Most language workbenches provide a means to develop a compiler for the DSL, facilitating code generation in general-purpose languages. (A comparison matrix for language workbenches can be found in [1].)

In this paper we focus on the integration of components into target systems. "Component" is a very general concept and it can be realized in different forms,

depending on the system. We particularly focus on a subset of components, *domain-specific components*, which are instances of domain-specific meta-models. The component structure is described with a DSL and the semantics are embedded into code generation templates, which are used to generate a component that is tailored towards a base system's requirements.

Integrating a generated component into a system poses three main challenges. (1) When adding unforeseen functionality to a system, no explicit *hooks* exist for attaching the generated component. In this case it may be necessary to modify the generated code, the system code or both to make the connection, which will expose the system developer to the implementation details of the generated code. (2) The interfaces of the generated component and the target system should be compatible to work together, which is generally not the case. Then one of the interfaces should be adapted, possibly by modifying the system's or the component's implementation or their type-system. (3) When the component or the target system evolves, the links between them must be re-established.

Current aspect-oriented languages offer mechanisms to modularly implement solutions for the first challenge. It can be solved by defining pointcuts that are used as hooks to a system. The second challenge is our main focus. Existing AO-languages offer limited mechanisms for implementing adapters between interfaces. AspectJ inter-type declarations can be used to make system classes to implement appropriate interfaces, however this approach is type-invasive. CaesarJ offers a more declarative approach with *wrappers*, but their instantiation requires pointcut declarations or they should be explicitly instantiated in the base system. The links mentioned in the third challenge are the adapter implementations mentioned in the second challenge and they represent the binding between two components. However current AO languages do not offer a declarative way for describing such a binding; an imperative programming language will lead to less readable and less maintainable implementation, which is fragile against software evolution.

2   Approach

In order to overcome the shortcomings of the existing approaches we intend to design a declarative way of implementing object adapters which is used together with a specialized pointcut for selecting objects. The object adapter pattern is common practice for binding two components that have incompatible interfaces. Our approach is aspect-oriented and it will provide the means to non-intrusively define and instantiate object adapters, inside aspects. These adapters represent links between the component and the system; their declarative design requires a declarative way of selecting the adaptee objects.

In order to select objects to be adapted, we have designed a new pointcut mechanism called *instance pointcut* which selects sets of objects based on the execution history. An instance pointcut definition consists of three parts: an identifier, a type which is the upper bound for all objects in the selected set, and a specification of relevant objects. The specification utilizes *pointcut expressions* to

select events that define the begin and end of life-cycle phases and to expose the object. At these events, an object is added or removed from the set representing the instance pointcut. It is possible to access all objects currently selected by an instance pointcut and to be notified, when an object is added or removed. New instance pointcuts can be derived from existing ones in several ways. Firstly, a new instance pointcut can be derived from another one by restricting the type of selected objects. Secondly, a *subset* or a *super-set* of an existing instance pointcut can be declared whereby the specification of the life-cycle phase is either narrowed down or broadened. Finally, instance pointcut declarations can be composed arbitrarily by means of boolean operators.

Adapter declarations refer to the sets selected by instance pointcuts, and automatically instantiate adapters for each object in the referred set. Unlike inter-type declarations, adapter declarations are not type invasive; they are compiled to the object adapter pattern and they do not change the type hierarchy of the contained object. They also do not require explicit instantiations.

(a) The shapes hierarchy   (b) ShapeInfo class that requires two unsupported interfaces
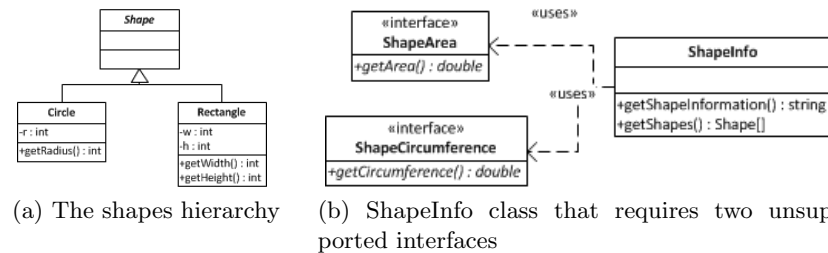
Fig. 1: Incompatible interfaces: Shape and ShapeInfo

The header of an adapter declaration consists of an identifier, the list of interfaces the adapter implements and an instance pointcut reference which contains the adaptee objects. In the body of an adapter declaration implementation of the interface methods is provided. In Figure 1a a Shape hierarchy and the interfaces offered by the classes in this hierarchy is shown. The ShapeInfo class uses ShapeArea and ShapeCircumference interfaces to query existing Shapes (Figure 1b). However none of the classes in the shapes hierarchy implements these interfaces, hence they should be adapted. Assume that there is a class called CircleCreator which has two methods: createLargeCircle and createSmallCircle. We can define an instance pointcut called largeCircles which selects the set of Circle objects that are created by the createLargeCircle method. Here instance pointcuts give us expressive power over selecting specific objects as adaptees. Listing 1 shows an example of an adapter declaration. The name of the adapter is CircleAdapter and it implements the interfaces defined in the square brackets; CircleAdapter adapts the objects selected by the circles instance pointcut. In the body of the adapter the implementations of the two declared interfaces are provided. The **adaptee** keyword refers to an object in the circles set.

```
1  declare adapter: CircleAdapter[ShapeArea, ↙
       ShapeCircumference] adapts largeCircles
2  {
3      public double getArea()
4      {
5          return Math.pow(adaptee.getRadius(),2)*Math.PI;
6      }
7      public double getCircumference()
8      {
9          return 2*adaptee.getRadius()*Math.PI;
10     }
11 }
```

Listing 1: The adapter declaration for Circle objects

3 Compilation and Run-time Support

In our prototype implementation instance pointcuts are compiled to AspectJ and Java code. Roughly, an instance pointcut is transformed to several AspectJ pointcuts, advice declarations, a set structure and methods for altering this set. Adapter declarations will also be compiled to AspectJ. According to our initial analysis an adapter declaration will map to a Java class for the adapter and advice bodies for initializing adapters. These advice bodies will reference the pointcuts generated from the instance pointcut which is referenced by the adapter declaration.

   We intend to provide run-time support for retrieving adapter instances. Adapters are automatically initialized when an adaptee object satisfying the referenced instance pointcut's conditions become available. These adapter instances can be indexed and accessed through a run-time library. To do this, we have the requirement that the results of a retrieval will always be non-ambiguous e.g. if a query to retrieve a single adapter instance, matches two adapters, then there should be appropriate resolution mechanisms or user feedback to overcome the issue.

References

1. Language workbench competition comparison matrix (2011), www.languageworkbenches.net
2. Czarnecki, K.: Overview of generative software development. In: Unconventional Programming Paradigms, Lecture Notes in Computer Science, vol. 3566, pp. 97–97. Springer Berlin , Heidelberg (2005)
3. Fowler, M., Parsons, R.: Domain-specific languages. Addison-Wesley (2010)
4. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37, 316–344 (December 2005)