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 according to a predefined software architecture.

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 developer to the generated code, defying the purpose of code generation. (2) When the component or the target system evolves, the links between them must be re-established. (3) The last issue is the mapping of the structures of DS-components to the corresponding structures in the system. A concept represented in a single structure in the DSL specification and generated as a single class, might be spread over multiple classes in the target system. This also applies for behavioral mappings. Then it is necessary to explicitly define this mapping and create the programming structures that implement it.

Current aspect-oriented languages offer mechanisms to implement solutions for the first challenge in a modular way. The first problem can be solved by defining pointcuts that are used as hooks to a system. The links mentioned in the second challenge can also be encapsulated in aspects with an approach that models binding as a cross-cutting concern. However current AO languages do not offer a declarative way for describing such a binding; an imperative implementation will lead to less readable and less maintainable implementation. The third challenge is our main focus. Existing AO-languages offer limited mechanisms for expressing such a mapping. 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.

2-8
2-1
2-2
2-3
2-4

2 Approach

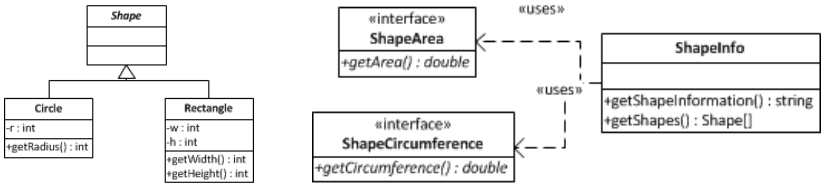
In order to overcome the shortcomings of the existing approaches we have designed a declarative way of implementing object adapters which is used together with a specialized pointcut for selecting objects. The object adapter pattern is common practice for binding two components that have incompatible interfaces.

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.

2-5
2-7
2-6

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 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 we have defined an instance pointcut called `circles` which selects the set of `Circle` objects that are created. 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 method implementations are shown. The **`adaptee`** keyword refers to an object in the `circles` set.

3 Compilation and Run-time Support

References

1. Language workbench competition comparison matrix (2011), `www.languageworkbenches.net`

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

Listing 1: The adapter declaration for `Circle` objects

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)

2-1 01.07.2012 21:15
to modularly implement

2-2 01.07.2012 21:15
"first problem" was mentioned in the previous sentence. Just say "it". By the way, don't switch between "problem" and "challenge".

2-3 01.07.2012 21:15
This needs more explanation.

2-4 01.07.2012 21:15
Is support limited or missing? Already support for the 2nd challenge was limited. It is not clear what AO languages can do at all. Explain the benefits and the limits in more detail.

2-5 01.07.2012 21:15
This is not yet worked out. Therefore present it as ongoing work. "We intend to design ..." (Only when you refer to the adapter declaration. When taking of instance pointcuts it is OK to say "we have...")

2-6 01.07.2012 21:15
How are they used? "It is possible to access all objects currently selected by an instance pointcut and to be notified, when an object is added or removed"

2-7 01.07.2012 21:15
The connection between instance pointcuts and adapters is not clear. Also you don't refer to the challenges from before. Say, how you intend to solve the challenges.

You intend the "adapters" to be the "links". For a declarative definition of the adapters you need to have declarative definitions of objects, which is why you need the instance pointcuts. ...

2-8 01.07.2012 21:15
Does the 3rd challenges still play a role? It's not very clear if you can solve this with adapters and the problem statement is a bit vague compared to the others. Consider to drop this challenge; don't forget to update the remainder of this section in this case.