Adapter Paper

Kardelen Hatun Christoph Bockisch 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

This paper provides a sample of a LaTeX document which conforms to the formatting guidelines for ACM SIG Proceedings. It complements the document *Author's Guide to Preparing ACM SIG Proceedings Using LaTeX2$_\epsilon$ and BibTeX*. This source file has been written with the intention of being compiled under LaTeX2$_\epsilon$ and BibTeX.

The developers have tried to include every imaginable sort of "bells and whistles", such as a subtitle, footnotes on title, subtitle and authors, as well as in the text, and every optional component (e.g. Acknowledgments, Additional Authors, Appendices), not to mention examples of equations, theorems, tables and figures.

To make best use of this sample document, run it through LaTeX and BibTeX, and compare this source code with the printed output produced by the dvi file.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION / COPIED FROM SLE DOCSYM PAPER

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 [?] and [?]. Generative programming [?] 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 [?].)

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 chall enges. (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) 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.

2.1 Adaptee Selection

The object adapter pattern relies on either getting the adaptee object as a parameter at construction or setting the adaptee object after construction. This approach requires acquiring adaptee objects and explicitly initializing the object adapters. 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. There are two ways to pass on the adaptee objects to an adapter declaration. First is referencing a collection instance which contains a set of objects to be adapted. This method is straightforward but limited in terms of object selection options. The second method uses a new pointcut mechanism that we have designed 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, 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.

2.2 Adapter Declarations

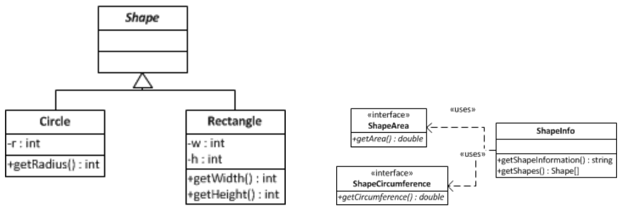
Adapter declarations are contained by aspects but like inter-type declarations they are not aspect members. In Figure 1 the basic syntax of adapter declarations are shown. The header of an adapter declaration consists of an identifier (Figure 1: $\langle identifier \rangle$), the list of interfaces the adapter implements (Figure 1: $\langle interface \rangle$+) and an adaptee reference which contains the adaptee objects. In the body of an adapter declaration implementation of the interface methods is provided. In the body of an adapter declaration, the keyword adaptee corresponds to the object that's being adapted.

⟨*concrete-adapter-decl*⟩ ::= '`declare adapter:`'
    ⟨*identifier*⟩ '`[`' ⟨*interface*⟩+ '`]`' '`adapts`' ⟨*adaptee-ref*⟩
    '`{`' ⟨*adapter-body*⟩ '`}`'

⟨*adaptee-ref*⟩ ::= ⟨*ip-ref*⟩
  | ⟨*obj-collection*⟩

Figure 1: Grammar definition for adapter declarations

Let us give a concrete example to illustrate how adapter declarations can be used. In Figure 2a 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 2b). However none of the classes in the shapes hierarchy implements these interfaces, hence they should be adapted.



(a) The shapes hierarchy

(b) ShapeInfo class that requires two unsupported interfaces

Figure 2: Incompatible interfaces: Shape and ShapeInfo

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 named CircleAdapter and which implements ShapeArea and ShapeCircumference interfaces (Line 1). CircleAdapter adapts the objects selected by the circles instance pointcut. In the body of the adapter the method implementations are shown. The first method getArea returns the area of the adaptee which is of type Circle, by calling adaptee's getRadius method (Lines 3 – 6). The getCircuference method from ShapeCircumference interface is similarly implemented (Lines 7 – 10).

2.2.1 *Adapter Hierarchies*

Similar to Java classes, adapter declarations can also form inheritance hierarchies. The extended grammar that support adapter inheritance is given in Figure 3. According to this grammar extension three types of adapter declarations are possible.

Abstract Adapter Declaration An abstract adapter declaration (⟨abstract-adapter-decl⟩ rule) takes the ab-

```
declare adapter: CircleAdapter[ShapeArea, ShapeCircumference] ↙
      adapts circles
{
  public double getArea()
  {
    return Math.pow(adaptee.getRadius(),2)∗Math.PI;
  }
  public double getCircumference()
  {
    return 2∗adaptee.getRadius()∗Math.PI;
  }
}
```

Listing 1: The adapter declaration for Circle objects

⟨*adapter-decl*⟩ ::= ⟨*abstract-adapter-decl*⟩
  | ⟨*concrete-adapter-decl*⟩
  | ⟨*adapter-extend-decl*⟩

⟨*abstract-adapter-decl*⟩ ::= '`declare adapter:`' 'abstract' ⟨*identifier*⟩ '`[`' ⟨*interface*⟩+ '`]`' ('`adapts`' ⟨*adaptee-ref*⟩ | ⟨*type-ref*⟩)? '`{`' ⟨*adapter-body*⟩? '`}`'

⟨*adapter-extend-decl*⟩ ::= '`declare adapter:`' ⟨*identifier*⟩('`[`' ⟨*interface*⟩+ '`]`')? '`extends`' ⟨*adapter-ref*⟩ '`adapts`' ⟨*adaptee-ref*⟩ '`{`' ⟨*adapter-body*⟩ '`}`'

Figure 3: Extended grammar definition for adapter hierarchies

stract keyword before its identifier. Such declarations do not have to provide an advice body or they can provide a partial adapter body, which implements only some methods of the declared interfaces. Different form concrete adapter declarations, the adaptee does not have to be bound. Adaptee reference can either be empty, or an adapter reference (⟨adaptee-ref⟩ rule in Figure 1) or it can be a *type reference*, in order to constrain the bound type for the concrete sub-adapter.

Extended Adapter Declaration An adapter can extend other adapters, either abstract or concrete. An extended adapter can override the interface methods defined by its super-adapter and implement new interfaces. However there are some rules when extending adapters.

- Extending an abstract adapter requires that all of the interface methods that are not implemented in the super-abstract adapter should be implemented by the sub-adapter, given that the sub-adapter is concrete.
- If the abstract super-adapter uses a type-reference in place of the adaptee, then the sub-adapters can only bind adaptees that are subtypes of the declared type-reference.
- If the super-adapter uses an instance pointcut reference as the adaptee, then the sub-adapters can only bind the type-refined subsets of that instance pointcut. **Alternative: the sub-adapters can only bind the subtypes of the instance pointcut's declared type.** This rule also hold for the plain object collections.

The typing constraints still hold if the sub-adapter is abstract, however the first rule does not apply.

```
1  declare adapter: abstract ShapeAdapter[ShapeArea, ↙
         ShapeCircumference, ShapeColor] adapts shapes
2  {
3    public String getColor(){
4          if (this.getArea() > 40)
5            return "RED";
6          else
7            return "BLUE";
8       }
9  }
10 declare adapter: CircleAdapter extends ShapeAdapter adapts ↙
         shapes<Circle> {
11   //implementation of interface methods for a circle
12 }

14 declare adapter: RectangleAdapter extends ShapeAdapter ↙
         adapts shapes<Rectangle>{
15   //implementation of interface methods for a rectangle
16 }
```

Listing 2: An abstract adapter declaration for the Shape hierarchy

Concrete Adapter Declaration Concrete adapter declaration was discussed in subsection 2.2.

Using the shapes example presented in Figure 2, we can illustrate how adapter hierarchies can be utilized. In Listing 2, line 1 an abstract adapter which adapts Shape objects is shown, here the instance pointcut shapes selects all created Shape objects. The `ShapeAdapter` contains an implementation of the `ShapeColor` interface, which will be a common implementation for all the sub-adapters if not overridden. The `getColor` method (lines 3 – 8) calls the `getArea` method (line 4) of the `ShapeArea` interface. When this method is called on the concrete sub-adapters, each sub-adapter will call its own `getArea` implementation. On line 10 we have defined a CircleAdapter which extends the abstract ShapeAdapter. CircleAdapter does not have to declare an interface list since it is a sub-adapter and it inherits the interfaces from its super-adapter. Another important point is the way we refined the shapes instance pointcut with a Java generics type syntax. By writing shapes<Circle> we can select the Circle instances from the instance pointcut set and bind the specialized CircleAdapter to these objects.

2.3 Adapter Compilation

Adapter declarations are compiled to Java classes and Aspects. **Some meta-information is attached to the generated code in order to be used during run-time(?).** The transformation of an adapter declaration to a Java class is straight-forward. For the example given in Listing 2 the Java classes shown in Listing 3 are generated.

3. REFERENCES

[1] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37** (December 2005) 316–344

```
public abstact class ShapeAdapter implements ShapeArea, ShapeCircumference, ShapeColor {
{
  Shape adaptee;
  ShapeAdapter(Shape s)
  {
    adaptee = s;
  }
  public String getColor(){
    if (this.area() > 40)
      return "RED";
    else
      return "BLUE";
  }
}
public class CircleAdapter extends ShapeAdapter{
  Circle adaptee;
  public CircleAdapter(Circle c)
  {
    super(c);
    adaptee = c;
  }
  //implementation of interface methods for a circle
}
public class RectangleAdapter extends ShapeAdapter{
  Rectangle adaptee;
  public RectangleAdapter(Rectangle r)
  {
    super(r);
    adaptee = r;
  }
  //implementation of interface methods for a rectangle
}
```

Listing 3: The generated Java code from the adapter declarations in Listing 2

[2] Fowler, M., Parsons, R.: Domain-specific languages. Addison-Wesley (2010)

[3] Czarnecki, K.: Overview of generative software development. In: Unconventional Programming Paradigms. Volume 3566 of Lecture Notes in Computer Science. Springer Berlin , Heidelberg (2005) 97–97

[4] : Language workbench competition comparison matrix (2011)