DESIGN PATTERNS WITH ASPECTJ, GENERICS, AND REFLECTIVE PROGRAMMING

Adam Przybylek
Department of Business Informatics, University of Gdansk, Piaskowa 9, 81-824 Sopot, Poland
adam@univ.gda.pl

Keywords: design patterns, AOP, generics, reflective programming.

Abstract: Over the past decade, there has been a lot of interest towards aspect-oriented programming (AOP). Hannemann and Kiczales developed AspectJ implementations of the Gang-of-Four (GoF) design patterns. Their study was continued by Hachani, Bardou, Borella, and others. However, no one has tried to improve the implementations by using generics or reflective programming. This research faces up to this issue. As a result, highly reusable implementations of Decorator, Proxy, and Prototype are presented.

1 INTRODUCTION

Capturing design knowledge in a way that makes it possible for others to reuse it is the basic idea behind design patterns (Noda & Kishi 2001). The solutions proposed in the original design pattern literature (Gamma et al. 1995) are shaped by techniques as well as language deficiencies from the object oriented (OO) paradigm. With the rise of the aspect-oriented (AO) paradigm, new programming abstractions have been occured that suggest it is time for these solutions to be revised.

First attempts to reshape design pattern solutions based on aspect-oriented programming (AOP) were initiated by Hannemann & Kiczales (2002) and then continued by Hachani & Bardou (2002), Borella (2003), Monteiro & Fernandes (2004), and Denier, Albin-Amiot & Cointe (2005). H&K developed AspectJ implementations of the Gang-of-Four (GoF) patterns. For 12 out of all 23 patterns, they found reusable implementations. However, AspectJ didn't support generics (generic types), when they were doing their research. With the advent of this technique, a new support for more reusable implementations have been occured. Generic types let programmers define a type without specifying all the other types it uses. The unspecified types are supplied as parameters at the point of use.

In this research the existing AO implementations were examined according to applying generics and reflective programming. It was found that generics bring advantages for the Decorator and the Proxy patterns, of which only the implementation of the former exhibits significant improvement. In addition, a highly reusable implementation of the Prototype pattern was achieved by combining AOP with reflective programming.

The modeling language used in this study is the extension to UML that was proposed by Przybyłek (2008). The entire source code with examples is available at the author's homepage (Przybyłek 2010).

2 MOTIVATIONS AND GOALS

For 11 out of 23 GoF patterns reusable implementations aren't known. Only the solutions of these patterns are considered reusable. As a consequence the programmer still has to implement the patterns for each application he is constructing (Borella 2003). However, to the best of the author's knowledge, there is no evaluation of the impact of generics or reflection in AspectJ on the GoF patterns. Generics were added to AspectJ in 2005 and since that time, implementations of design patterns have had the potential to become more reusable. Hence, the solutions that have been presented so far should be revisited and reconsidered. The aim of this paper is to present the results of exploring the existing AO implementations according to apply generics and reflection.

3 THE DECORATOR PATTERN

The intent of the Decorator pattern is to perform additional actions on individual objects (Borella 2003; Gamma et al. 1995). The additional actions and the decorated objects are selected at runtime. An alternative for this pattern is inheritance. However, the Decorator pattern has several advantages over subclassing. First, additional actions can be added and removed at runtime and per object. Second, combining independent extensions is easy by composing decorator objects. With subclassing every combination of extensions results in a new subclass and this can cause an explosion of subclasses.

There are many variations of the Decorator pattern, but in this paper the one used is that defined by Borella (2003). The first AO approach to this pattern was presented by Hannemann & Kiczales (2002). Their solution has three limitations (Borella 2003). Firstly, it does not allow for dynamic attaching and dynamic detaching of decorators. Secondly, it is not possible at runtime to define an order of activation among decorators. Thirdly, after inserting the decorator aspect in the system, all instances of the intended class are decorated.

Other solutions were proposed by Monteiro & Fernandes (2004) and Hachani & Bardou (2002) but these do not add anything new and so need not be considered. A significant contribution to implementing the Decorator pattern was made by Borella (2003). He uses a per-target instantiation model to decorate only a subset of the instances of a class. His solution enables decorators to be composed dynamically. The order of activation of each decorator is given at runtime.

The Borella solution has two main imperfections. Firstly, the around advice and the wrap method are not generic, and depend on the type of the decorated object. Secondly, the decorator classes could directly implement the Decorator interface. Introducing the Decorator interface via the parent declaration unnecessarily complicates the solution. Fig. 1 presents a new solution to the Decorator pattern.

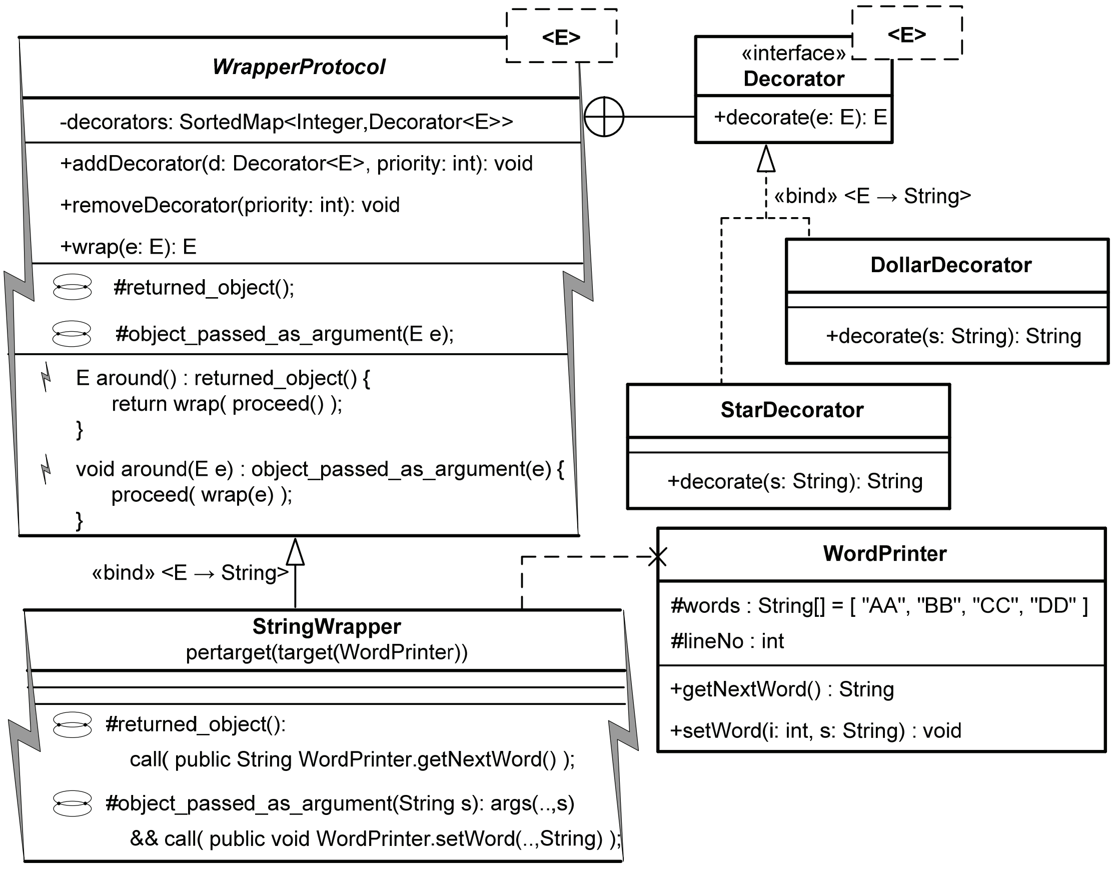
WrapperProtocol<E> describes the generic structure and behaviour that is determined by the pattern and it does not define any application specific behaviour. This solution reduces the non-reusable parts of the implementation. The around advices are responsible for intercepting objects which should be decorated and passing them to the wrap(E e) method as argument. This method iterates through all the registered decorators and gives them its argument to decorate (Listing 1). After decoration, the argument is returned. A concrete decoration process is implemented in the StarDecorator and DollarDecorator classes.

```
public E wrap(E e) {
  for ( Decorator<E> dec: decorators.values() ) e=dec.decorate(e);
  return e;
}
```

Listing 1: The wrap method.

```
public class testDecorator {
  public static void main(String[] args) {
    WordPrinter w = new WordPrinter();                      //1
    StringWrapper wrapper = StringWrapper.aspectOf(w); //2
    wrapper.addDecorator( new StarDecorator(), 2 );     //3
    wrapper.addDecorator( new DollarDecorator(), 1 );  //4
    w.setWord(0,"XXX");                                     //5
    System.out.println( w.getNextWord() );             //6
  }
}
```

Listing 2: A use of the Decorator pattern.



<E>
WrapperProtocol
-decorators: SortedMap<Integer,Decorator<E>>
+addDecorator(d: Decorator<E>, priority: int): void
+removeDecorator(priority: int): void
+wrap(e: E): E
#returned_object();
#object_passed_as_argument(E e);
E around() : returned_object() {
return wrap(proceed());
}
void around(E e) : object_passed_as_argument(e) {
proceed(wrap(e));
}
<E>
«interface»
Decorator
+decorate(e: E): E
«bind» <E → String>
DollarDecorator
+decorate(s: String): String
StarDecorator
+decorate(s: String): String
«bind» <E → String>
StringWrapper
pertarget(target(WordPrinter))
#returned_object():
call(public String WordPrinter.getNextWord());
#object_passed_as_argument(String s): args(..,s)
&& call(public void WordPrinter.setWord(..,String));
WordPrinter
#words : String[] = ["AA", "BB", "CC", "DD"]
#lineNo : int
+getNextWord() : String
+setWord(i: int, s: String) : void

Figure 1: The Decorator pattern.

The joinpoints at which the object to decorate should be captured are specified by the returned_object and object_passed_as_argument pointcuts. Thus it is possible to decorate the object, which is passed as an argument or returned as a result of a method call. The definitions of both pointcuts are empty, so at least one of them should be overridden in a subaspect.

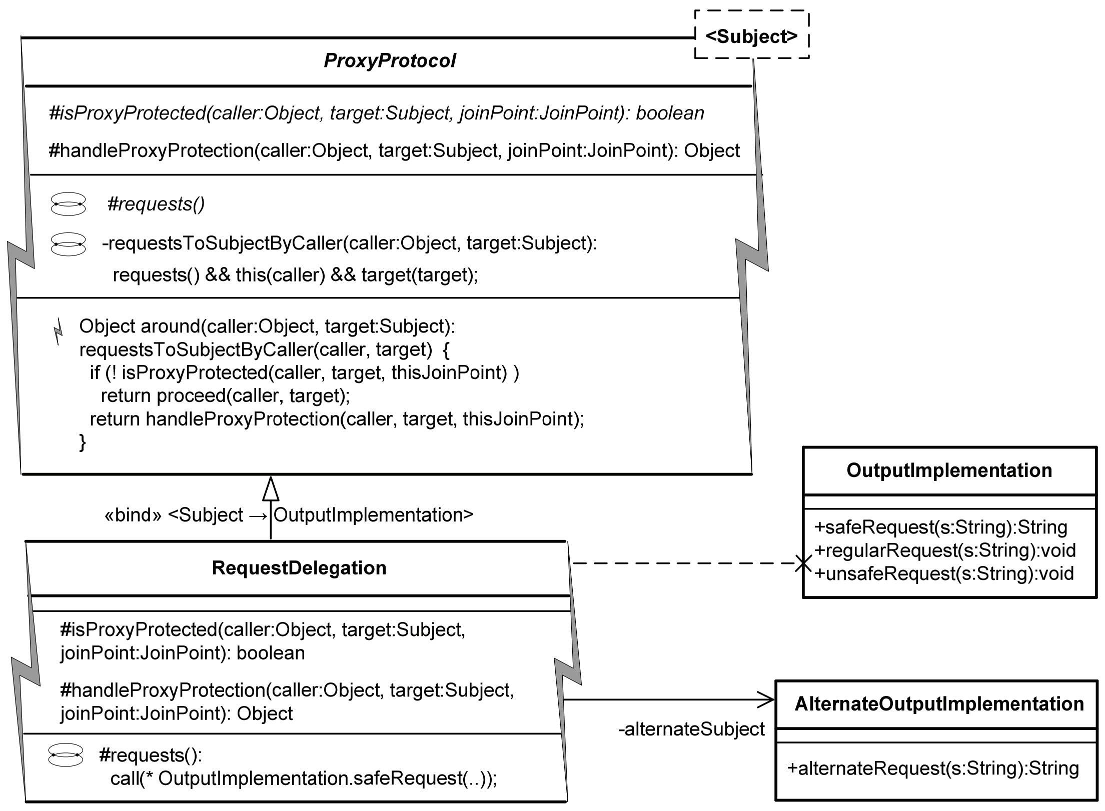
The concrete subaspect, which knows what type of object is captured and in which context, has to be derived from WrapperProtocol<E> by giving a bound type to the E parameter. One of such subaspects is StringWrapper that binds the E parameter with String. StringWrapper intercepts requests to the getNextWord() method and performs a decoration on the object returned by this method. An example of the use of StringWrapper is shown in Listing 2.

In order to decorate a specific object, the instance of StringWrapper that is associated with this object is retrieved (line 2). Zero or more decorators are then attached to this instance (lines 3 and 4). Without any decorator, the getNextWord() method would return "AA". However, the wrapper object has registered (lines 3 and 4) the StarDecorator and DollarDecorator instances, which wrap the returned object with "***" and "$$$" respectively. As a result, the "*** $$$ AA $$$ ***" string is printed on the screen (line 6).

4 THE PROXY PATTERN

The proxy pattern allows the developer to provide a surrogate object in place of the actual object in case access to the real object needs to be delegated or controlled. The following are some of the more common uses for proxies (Grand 2002):

- Represent an object that is complex or time consuming to create with a simpler one.
- Create the illusion that an object on a different machine is an ordinary local object.
- Control access to a service-providing object based on a security policy.
- Create the illusion that a service object exists before it actually does.



<Subject>
ProxyProtocol
#isProxyProtected(caller:Object, target:Subject, joinPoint:JoinPoint): boolean
#handleProxyProtection(caller:Object, target:Subject, joinPoint:JoinPoint): Object
#requests()
-requestsToSubjectByCaller(caller:Object, target:Subject):
requests() && this(caller) && target(target);
«bind» <Subject → OutputImplementation>
RequestDelegation
#isProxyProtected(caller:Object, target:Subject, joinPoint:JoinPoint): boolean
#handleProxyProtection(caller:Object, target:Subject, joinPoint:JoinPoint): Object
#requests():
call(* OutputImplementation.safeRequest(..));
OutputImplementation
+safeRequest(s:String):String
+regularRequest(s:String):void
+unsafeRequest(s:String):void
-alternateSubject
AlternateOutputImplementation
+alternateRequest(s:String):String

```
Object around(caller:Object, target:Subject):
requestsToSubjectByCaller(caller, target)  {
  if (! isProxyProtected(caller, target, thisJoinPoint) )
    return proceed(caller, target);
  return handleProxyProtection(caller, target, thisJoinPoint);
}
```

Figure 2: The Proxy pattern.

The structure of this pattern, that uses generics, is shown on Fig. 2. ProxyProtocol<Subject> is a reusable part of the implementation. Subject is a parameter. The client binds this parameter with the type of the object to be "proxied". The requestsToSubjectByCaller pointcut intercepts calls to the subject. If a call is proxy protected, the handleProxyProtection method is called instead of the original method. The isProxyProtected method checks whether the request should be handled by the proxy or not. By default it returns true. The handleProxyProtection method provides an alternative return value if a call is proxy protected. The default implementation returns null. Concrete subaspects are forced to implement the requests pointcut. This pointcut defines which requests need to be handled by the proxy.

5 THE PROTOTYPE PATTERN

The Prototype pattern allows an object to create a copy of itself without knowing its direct class. This pattern can avoid expensive “creation from scratch”. The most important requirement for objects to be used as prototypes is that they have a method, typically called copy, that returns a new object that is a copy of the original object. How to implement the copy operation for the prototypical objects is another important implementation issue. There are two basic strategies for implementing the copy operation (Grand 2002):

– Shallow copying means that the attributes of the cloned object contain the same values as the attributes of the original object and that all object references indicate the same objects.
– Deep copying means that the attributes of the cloned object contain the same values as the attributes of the original object, except that attributes that refer to objects refer to copies of the objects referred to by the original object. In other words, deep copying also copies the objects that the object being cloned refers to. Implementing deep copying can be tricky. You will need to be careful about handling any circular references.

Shallow copying is easier to implement because all classes inherit a clone method from the Object class that does just that. However, unless an object's class implements the Cloneable interface, the clone method will throw an exception and will refuse to work. This paper presents the first strategy with some modification (Figure 3).

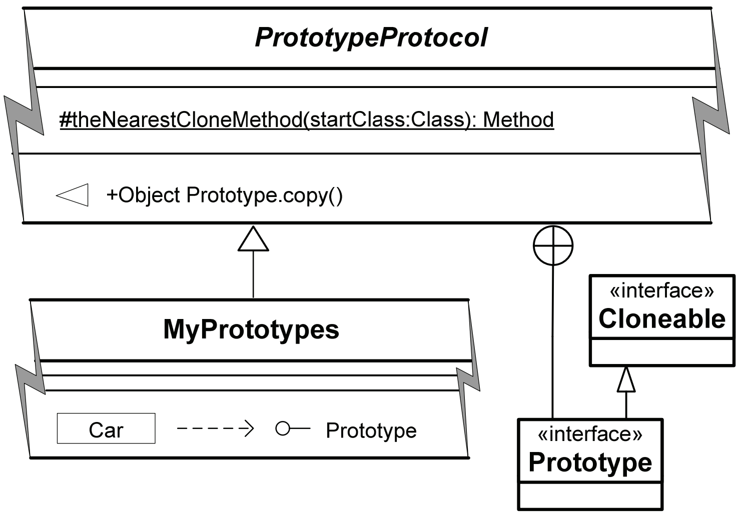


Figure 3: The Prototype pattern.

```
privileged abstract aspect PrototypeProtocol {

  public interface Prototype extends Cloneable{}

  public Object Prototype.copy() {
    Object copy = null;
    Method cloneMethod=null;
    try {
      Class thisClass = ((Object) this).getClass();
      cloneMethod = theNearestCloneMethod(thisClass);
      cloneMethod.setAccessible(true);
      copy = cloneMethod.invoke(this, null);
    } catch(Exception ex) {
      System.out.println(ex);
    }
    return copy;
  }

  protected static Method theNearestCloneMethod(Class startClass) {
    Method cloneMethod=null;
    do {
        try {
          cloneMethod = startClass.getDeclaredMethod("clone", null);
        } catch(NoSuchMethodException ex) {
          startClass = startClass.getSuperclass();
        }
    } while( cloneMethod == null );
    return cloneMethod;
  }
}
```

Listing 3: The PrototypeProtocol aspect.

The PrototypeProtocol aspect attaches a default copy() method on all Prototype participants. The implementation of that method is (1) to find the nearest clone() method up the class hierarchy, (2) to invoke it, and (3) to return the result (Listing 3). The searching process starts with the runtime class of the object to which the copy() request was sent. If that class does not define the clone method, then the search is made in its superclass. In the worst case, the search repeats until the Object class is reached. MyPrototypes assigns the Prototype interface to Car.

6 DISCUSSION

This research presents how AO implementations of the Decorator and the Proxy pattern can be improved using generics. Parametrized aspects, which serve as protocols, were created for both patterns. For the Prototype pattern reflective programming was employed to provide a default implementation for cloning. For Decorator not only implementation but also a new solution was developed. The solution is simpler then the one proposed by Borella, whereas the implementation is much more reusable. The only thing a programmer has to implement is two pointcuts. For the Proxy pattern, Hannemann & Kiczales's solution was used, and only the implementation was adapted. In order to use these patterns, concrete sub-aspects have to be created to make all general properties specific. The implemented patterns can be plugged or unplugged depending on the presence or absence of aspects.

7 SUMMARY

Hannemann and Kiczales introduce solutions that make an application independent of design patterns and improve reusability of the application core part. However, the code for a design pattern is still not reusable. This paper presents the results of exploring the existing AO implementations according to applying generics and reflective programming. It was found that only Decorator and Proxy are suitable to use with generics, while Prototype is suitable to use with reflection. In each case, the applied programming techniques enhanced reusability of the design pattern part.

REFERENCES

Borella, J., 2003. Design Patterns Using Aspect-Oriented Programming. *MSc thesis*, IT University of Copenhagen

Denier, S., Albin-Amiot, H., Cointe, P., 2005. Expression and Composition of Design Patterns with Aspects. In: *2nd French Workshop on Aspect-Oriented Software Development (JFDLPA'05)*, Lille , France

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston

Grand, M., 2002. *Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML.* John Wiley & Sons

Hachani, O., Bardou, D., 2002. Using Aspect-Oriented Programming for Design Patterns Implementation. In: *Workshop Reuse in Object-Oriented Information Systems Design*, Montpellier, France

Hannemann, J., Kiczales, G., 2002. Design Pattern Implementation in Java and AspectJ. In: *17th Conference on Object-Oriented Programming Systems, Languages, and Applications*, Seattle

Monteiro, M.P., Fernandes, J.M., 2004. Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns. In: *Desarrollo de Software Orientado a Aspectos (DSOA'04)*, Málaga, Spain

Noda, N., Kishi, T., 2001. Implementing Design Patterns Using Advanced Separation of Concerns. In: *Workshop on Advanced SoC in OO Systems at OOPSLA'01*, Tampa Bay, Florida

Przybyłek, A., 2008. Separation of Crosscutting Concerns at the Design Level: An Extension to the UML Metamodel. In: *3rd International Multiconference on Computer Science and Information Technology (IMCSIT'08)*, Wisła, Poland

Przybyłek, A., 2010. http://przybylek.wzr.pl/AOP/icsoft2010.zip