The Decorator pattern revisited: an aspect-oriented solution.

Adam Przybyłek

Gdańsk University, Department of Business Informatics, Piaskowa 9, 81-824 Sopot, Poland
adam.przybylek@univ.gda.pl

Abstract A critical challenge for software developers is to utilize design experience and reuse existing software when building new systems. Design patterns have proved effective in helping to address these issues. However, the solutions proposed by the literature on design patterns are determined by techniques and language constructs established by object-oriented programming (OOP). With the rise of the aspect-oriented (AO) paradigm, new programming abstractions have been presented that suggest that it is time for these solutions to be revisited. In this paper parametric polymorphism and AO constructs are applied to develop a reusable implementation of the Decorator pattern that can be (un)plugged into existing software.

1 Introduction

Developing high quality software requires knowledge usually learned only by experience [1, 5]. Experience acquired in projects that have worked in the past allows a designer to avoid the pitfalls of development [8]. Capturing design knowledge in a way that makes it possible for others to reuse it is the basic idea behind design patterns [10]. The most useful 23 patterns were catalogued by Gamma et al. in [5] what is known as the “Gang of Four” book.

A design pattern is a general solution that addresses a recurring problem encountered in object-oriented (OO) software development [5]. It constitutes a set of guidelines that describe how to accomplish a certain task in a specific design situation [11]. A design pattern also identifies classes that play a role in the solution to a problem and describes their collaborations and responsibilities.

The use of design patterns promotes the construction of software that is adaptable, extensible, robust, reusable and less prone to programmer error [1, 2, 8]. Robustness is the ability of software to be modified locally without redesigning the overall structure and reflects ease of maintenance. Software reuse saves development time and reduces the costs of implementation.

Proceedings of BIR’2008 the 7th International Conference on Perspectives in Business Informatics Research, Gdansk, Poland, September 25-26, 2008

Although design patterns are an established strategy, the emergence of aspect-oriented (AO) languages has given new insight into their implementation. Several studies have shown that AO programming (AOP) is a promising technique for improving software quality [8]. These improvements are manifest in terms of better code locality, reusability, composability, and (un)pluggability [6].

In recent years design patterns have been used to illustrate the advantage of AOP over the traditional OO programming (OOP) [14]. Hannemann and Kiczales showed modularity improvements in 17 of 23 cases when implementing the design patterns described in [5]. The present article goes a step further and shows how AO solutions can take advantage of parametric polymorphism (also known as generics). Its main contribution is to provide a reusable implementation of the Decorator pattern that can be introduced to existing software in a non-intrusive way. The intent of the Decorator pattern (also known as the Wrapper pattern) is to perform additional actions on individual objects [3, 5]. The additional actions and the decorated objects should be selected at runtime. There are many variations of this pattern, but in this paper the one used is that defined by Borella [3]. All the implementations presented have been standardized to simplify the review.

The remainder of this paper is structured as follows. Section 2 defines some necessary terminology and outlines a graphical notation used in the rest sections. The motivation and contribution of the research are discussed in Section 3. Existing AO solutions of the Decorator pattern are reviewed in Section 4. A novel solution and implementation of this pattern are introduced in Section 5. Summary is given in the last section.

2 Background

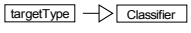
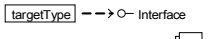
In recent years a growing interest in AO software development (AOSD) has been reported. AOSD aims to increase software quality by introducing a new unit of modularity, called aspect. Aspects allow a crosscutting concern to be implemented in a modular and well-localized way [12]. A concern is called a crosscutting concern if it cannot be modularly represented within traditional decomposition units such as classes or functions [13].

The most well-known and the most mature AO language today is AspectJ. AspectJ grew out of the research work undertaken by Gregor Kiczales [7] at Xerox PARC (Palo Alto Research Center). It adds six main concepts to Java that enable crosscutting concerns to be implemented in a modular manner. These are: an aspect, a joinpoint, a pointcut, an advice, an introduction, and a parent declaration. An **aspect** serves as container for the rest of the constructs. A **joinpoint** is a well-defined location in the program flow where the implementation of a crosscutting concern can be plugged in. Typical examples of joinpoints include a throw of an exception, a call to a method and an object instantiation. A **pointcut** is a language construct designed to identify and select joinpoints within the program flow. It can also extract contextual information such

as the target object and method arguments. An **advice** is a method-like construct used to define an additional behaviour that has to be inserted at each joinpoint picked out by the associated pointcut. The body of an advice is the implementation of a crosscutting functionality. Depending on the type of advice, whether “before”, “after” or “around,” the body of an advice is executed before, after or in place of the selected joinpoints. An around advice may not execute the captured call at all, may execute it with the original context, may change the context (such as the parameters) or may even execute it in multiple. An **introduction** is used to crosscut the static-type structure of a given class. It allows a programmer to add attributes and methods to the class without having to modify it explicitly. Existing interfaces can also be enriched by this mechanism. A **parent declaration** may change the classes' super-classes and interfaces by inserting new generalization and realization relationships into the class structure.

While AOSD has gained widespread industry support at the programming level, less attention has been given to the design level. In a previous paper [13] the present author introduced AoUML to fill this gap. AoUML is defined as an extension to the UML meta-model with specific elements in support of AOM (Table 1).

Table 1. AoUML notation.

Notation	Element
	aspect
	pointcut
	advice
	introduction
	parent declaration
	
	precedence

3 Motivation and goals

Although the design patterns introduced by the Gang of Four are a widely recognized approach for building high quality software, they have some limitations. Usually only the solutions of these patterns are considered reusable, whereas the implementations are not [3, 4]. As a consequence the programmer still has to implement the patterns for each application he is constructing [3]. Moreover, some of the programs written according to design patterns are complex and the overall structure of the programs is not easy to understand [1, 4].

This paper advocates the use of AOP and generics to address these problems. The first attempt to use AOP to improve solutions of design patterns was initiated

by Hannemann and Kiczales [6]. The studies were then continued by Borella [3], Monteiro [9], and Denier [4]. However, the solutions referred to do not consider generics added to AspectJ in 2005. Since that time, implementations of design patterns have had the potential to become more flexible and reusable. This paper reviews the Decorator pattern by showing the reusable implementation that can be (un)plugged into existing software.

4 Related Works

The first AO approach to the Decorator pattern (Figure 1) was presented by Hannemann and Kiczales [6]. In the *StarDecorator* and *DolarDecorator* aspects, concrete decorations are implemented. These aspects intercept requests to the *WordPrinter::setWord(int,String)* method and perform decorations on the method's *String* argument before the method starts.

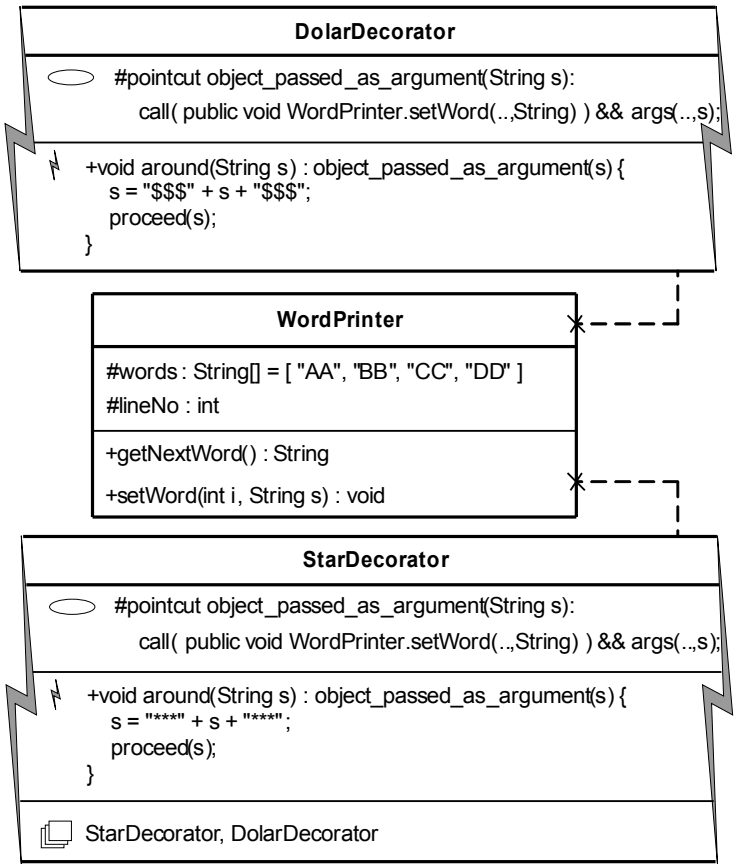


Fig. 1. The Hannemann-Kiczales solution

The above solution has three limitations [3, 14]. 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 *WordPrinter* class (Listing 1) are decorated.

```
public class WordPrinter {
  protected String[] words = {"AA", "BB", "CC", "DD"};
  protected int lineNo = 0;

  public String getNextWord() {
    return word[lineNo++ % word.length];
  }
  public void setWord(int i, String s) {
    word[i % word.length] = s;
  }
}
```

Listing 1. The WordPrinter class

Another solution was proposed by Monteiro et al. [9], but this does not add anything new and so need not be considered. A significant contribution to the task of implementing the Decorator pattern was made by Borella [3], whose solution is shown in Figure 2.

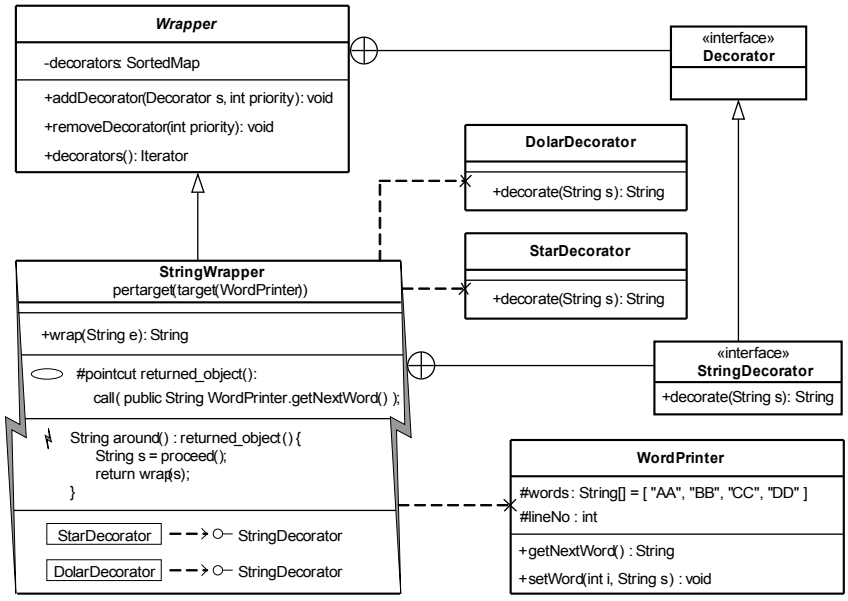


Fig. 2. The Borella solution

Borella uses a per-target instantiation model to decorate only a subset of the instances of a class. In order to decorate a specific object (Listing 2) 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).

```
WordPrinter w = new WordPrinter(); //1
StringWrapper wrapper = StringWrapper.aspectOf(w); //2
wrapper.addDecorator( new StarDecorator(), 2 );  //3
wrapper.addDecorator( new DolarDecorator(), 1 ); //4
w.setWord(0,"XXX"); //5
System.out.println( w.getNextWord() ); //6
```

Listing 2. A use of the Decorator pattern

The *addDecorator* and *removeDecorator* methods enable decorators to be composed dynamically. The order of activation of each decorator is given by the parameter *priority*. The decorators are stored in the *decorators* attribute (Listing 3).

```
public abstract class Wrapper {
  protected interface Decorator{};

  private SortedMap decorators = new TreeMap();

  protected Iterator decorators() {
    return decorators.values().iterator();
  }
  public void addDecorator(Decorator d, int priority) {
    decorators.put(new Integer(priority), d);
  }
  public void removeDecorator(int priority) {
    decorators.remove(new Integer(priority));
  }
}
```

Listing 3. The Wrapper class

The *StringWrapper::wrap(String)* method iterates through all the registered decorators and passes them the object to decorate (Listing 4).

```
public String wrap(String s) {
  Iterator i = decorators();
  while (i.hasNext()) {
    s = ( (StringDecorator) i.next() ).decorate(s);
  }
  return s;
}
```

Listing 4. The StringWrapper::wrap(String) method

An example of the use of *StringWrapper* is presented in Listing 2. Without decorators, the *WordPrinter::getNextWord()* method would return "XXX". However, the *wrapper* object has registered (lines 3 and 4) the *StarDecorator* and *DolarDecorator* instances, which wrap the returned object with "***" and "$$$" respectively. As a result, the "*** $$$ XXX $$$ ***" string is printed on the screen (line 6).

5 Proposed approach

The Borella solution has tree main imperfections. (1) The around advice and the wrap method depend on a concrete wrapper and so are not reusable. (2) The *StarDecorator* and *DolarDecorator* classes are defined when new requirements on decoration are known. They could, therefore, directly implement the *StringDecorator* interface. Introducing the *StringDecorator* type via the parent declaration unnecessarily complicates the solution. (3) The *Decorator* interface, which is nested within the *Wrapper* class, is not needed at all. Since the *SortedMap* operates on an item typed by *Object*, the *addDecorator* method could also declare the first argument as *Object* instead of *Decorator*. Figure 3 presents a significant improvement to the Borella solution.

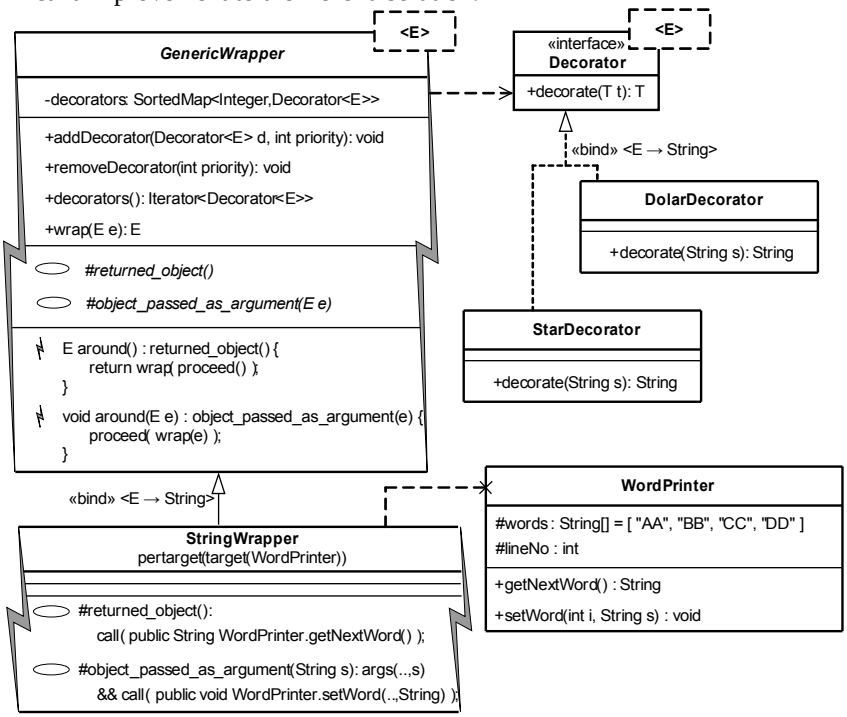


Fig. 3. The Przybyłek solution

The *wrap* method and *returned_object* advice have been generalized and incorporated into *GenericWrapper<E>*. This solution minimizes the non-reusable parts (*StringWrapper*) of the implementation. The *GenericWrapper<E>* is responsible for intercepting objects which should be decorated and passing them to the *wrap* method. The concrete aspect, which knows what type of object and in which context it should be captured, has to be declared as a subaspect of *GenericWrapper<E>* by giving a bound type to the *E* parameter. The joinpoints at which the object to decorate should be captured are specified by the *object_passed_as_argument* and *returned_object* 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 provided by a concrete aspect (e.g. *StringWrapper*) derived from *GenericWrapper<E>*. If there is no need to apply one of these pointcuts, its implementation may be left empty. Since only the implementation of the pointcuts remains in a concrete aspect, the solution is highly reusable.

6 Summary

Some programs written according to traditional solutions of design patterns tend to be too complicated and their overall structure is not easy to understand. This study examines how implementations of the Decorator pattern can be improved using generic types and AOP. The presented examples have shown that these orthogonal techniques nicely complement each other. The solution to the Decorator pattern has become much simpler and highly reusable, not only at the conceptual but also at the implementation level.

References

1. Albin-amiot, H., Guéhéneuc, Y.: Design Patterns: A Round-trip. In: 15th European Conference on Object-Oriented Programming (ECOOP'01), Budapest (2001)
2. Bieman, J.M., Straw, G., Wang, H., Munger, P.W., Alexander, R.T.: Design Patterns and Change Proneness: An Examination of Five Evolving Systems. In: 9th IEEE International Software Metrics Symposium, Sydney (2003)
3. Borella, J.: Design Patterns Using Aspect-Oriented Programming. MSc thesis, IT University of Copenhagen (2003)
4. Denier, S., Albin-Amiot, H., Cointe, P.: Expression and Composition of Design Patterns with Aspects. In: 2nd French Workshop on AOSD, Lille (2005)
5. Gamma, E. et.al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Boston (1995)
6. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: 17th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02), Seattle (2002)

7. Kiczales, G. et.al.: Aspect-Oriented Programming. In: 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä (1997)
8. Kuhlemann, M.: Design Patterns Revisited. School of Computer Science University of Magdeburg (2007)
9. Monteiro, M.P., Fernandes, J.M.: Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns. In: Proceedings of the Desarrollo de Software Orientado a Aspectos (DSOA'04), Málaga (2004)
10. Noda, N., Kishi, T.: Implementing Design Patterns Using Advanced Separation of Concerns. In: OOPSLA 2001 Workshop on Advanced SoC in OO Systems, Florida, 2001
11. Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw-Hill, New York (2005)
12. Przybyłek, A.: Post Object-Oriented Paradigms in Software Development: a Comparative Analysis. In: 2nd International Multiconference on Computer Science and Information Technology, Wisła (2007)
13. Przybyłek, A.: Separation of Crosscutting Concerns at the Design Level: An Extension to the UML Metamodel. In: 3rd International Multiconference on Computer Science and Information Technology, Wisła (2008)
14. Sousa, E., Monteiro, M.P.: Implementing Design Patterns in CaesarJ: an Exploratory Study. In: 6th Workshop on Software-Engineering Properties of Languages and Aspect Technologies (SPLAT'08), Brussels (2008)