Refactoring Design Patterns from Object-Oriented to Aspect-Oriented in Eclipse

Daniele Contarino
University of Catania
Viale A. Doria 6, 95125 Catania, Italy
daniele.contarino@studium.unict.it

***Abstract*—Nowadays, Aspect-Oriented programming is used for industrial software systems. However, many useful Object-Oriented systems have been developed and are still in use and a manual porting of big projects from Object-Oriented (OO) paradigm to Aspect-Oriented (AO) one, making the most of Design Patterns (DP), may prove a costly and time-consuming process. This paper proposes a practical approach for automatically converting OO code to AO code, specifically for the implementation of DPs. The solution comprises two essential steps: recognition of the DP in OO code and conversion into AO code. Then, two examples of DP Singleton and Composite implementations on the Eclipse IDE with Eclipse Java Development Tools (JDT) are given.**

***Index Terms*—Design Patterns, Aspect-Oriented programming, Recognition, Code replace, Separation of concerns**

I. Introduction

Design Patterns are an essential resource for developing software systems of any size. The Design Patterns (DPs) are the result of several years of fieldwork by senior software engineers. They allow us to save time when designing an application. However, many DPs have some crosscutting code (e.g. DP Observer) or inject code in a different concern (e.g. DP Singleton). Aspect-Oriented programming [1] solves the crosscutting code problem by moving such a code in separate modules or *aspects*. A concern is a specific concept or goal; the concerns can be domain or core concerns. Usually, in OO programming the core concerns are represented by main classes, then there are other non-domain or system concerns. In OO systems the latter concerns are crosscutting, while in AO systems they are aspects, as e.g. logging systems, user session checker, etc.

Design Patterns in AO systems [2]–[5] improve code reusability, reduce the software complexity, by extracting and separating the business logic code and the DP feature or *non-domain concerns* [6], [7]. Furthermore, the AO versions of DPs are more reusable, since a single aspect can be used for all the application classes needing it, and reduce the lines of code, both for the business logic and the DP portions. This allows the developer to focus more on models and less on the interactions between the classes. Moreover, the same aspect can be reused without changes on other software systems.

Unfortunately, carrying out a refactoring from the OO to AO paradigm for medium or large software systems, for having the benefits of the latter one, can be a very long and expensive job. It could also produce human-related mistakes. Therefore, several approaches have been proposed to assist refactoring systems, from detecting the portions of code that need to be improved [8]–[11], to carry out some transformation on the code [12], [13]. Moreover, extensive research work has been performed for identifying design patterns [14], [15], which in turn can help understand and improve the code.

This paper proposes to assist finding the portions of code that can be refactored and to transfom such found portions. This allows us to save time and reduce errors, requiring human intervention only for custom DPs. Given a software project, our idea would be to have a tool able to:

- Recognise the used DPs;
- Identify the classes the DP implements;
- Create the aspects related with the found DP;
- Delete the DP features from the OO version of the code and make appropriate changes to transform the code.

Therefore, a tool has been implemented with the said goals. The chosen development environment is Eclipse; this choice was motivated both for its widespread use in the enterprise environments and its internal flexible support. This allows building plugins that can easily perform parsing, checking and changing of the existing application code.

The use of aspects, hence the substantial replacement of crosscutting code, helps us to separate the domain concerns (business logic) from non-domain concerns (DP logic). However, some metadata, as Java annotations used to tag the class with the role for a specific DP will be used, as a kind of link for the two parts.

II. Background concepts and tools

A. Design Patterns

Design patterns are as the developer "utility knives". They represent a set of best practices conceived by experienced OO software developers; and are solutions for general problems that happen during the software development. DPs describe a given scenario. For example, Singleton DP lets us use a single object in all the application; the developers know this technique with the name Singleton, independently from the language and platform used. Learning DPs helps the unexperienced developers to solve severals problems that otherwise would require the help of a senior software engineer.

Each DP is described by:

1) name;
2) goal;
3) occurring problem;
4) solution offered;
5) effects.

In 1994 one of the most widespread used DP collection was published by four authors (also called "Gang of Four"). It describes 23 patterns divided in three categories: Creational, Structural and Behavioral patterns [16].

B. Aspect-Oriented programming

The Aspect-Oriented Programming is a programming paradigm that allows crosscutting concerns to be separated into modules called aspects [1]. We use in our project the AspectJ technology with Java [17]. In AspectJ, an aspect consists of

- Join Point: is a point inside the application execution (e.g. method invocation, method return, method execution, object creation, exception, fields access). In this point it usually interferes a cross-cutting concern or rather the aspect.
- Pointcut: is the tool that selects the Join Point based by the context and the rule specified as the definition.
- Advice: like a method, it executes a code when a join point is selected by a pointcut.

AspectJ is an extension for Java SDK. AspectJ is a preprocessor that integrates the cross-cutting concerns into Java code. It converts the Pointcuts and Advices into Object-Oriented code before that the Java code is compiled in bytecode.

C. Eclipse JDT

Eclipse Java Development Tools (JDT) is a tool suite for developing Java applications that can manipulate Java source code. For this reason, JDT is a fundamental package of Eclipse IDE, and is used by several Eclipse plugins.

JDT package is made of three items: Java Models, Search Engine and Abstract Syntax Tree (AST). Our developed tool is a plug-in that uses the Eclipse JDT for recognizing and editing the Java Eclipse projects and the Eclipse UI library for integrating with the IDE (context menu, message box, etc.).

Eclipse UI gave us the project in a Java Model representation. We then can create the handles that can be attached to the context menu in the Package Explorer of Eclipse. The handles catch the selected project (as IJavaProject) and pass it to our (individual) DP converter.

Within the Eclipse JDT library the following main classes have been used.

- IWorkspace represents Eclipse workspace;
- IJavaProject represents a single Java project;
- IPackageFragmentRoot represents a collection of resources (e.g. a folder or a library);
- IPackageFragment represents a sub-package inside a project;
- ICompilationUnit represents a file with Java source code;
- IType represents a class;
- IMethod represents a method;
- IField represents an attribute.

III. DP Recognition and conversion process

A. Recognition

The recognition of design patterns implemented inside a Java software system is carried out by observing the OO features of a DP, hence looking for their typical "signature" in a software system. One of the most used techniques consists of analysing the classes and finding relationships such as *implements* or *extends* [15], [18], [19]. In this way we can determine the class hierarchy and check whether it matches that of a DP. Every DP has a typical structure and we propose to search for three features. Our recognition algorithm can make use of the following.

- Match with known specific classes or interfaces. This technique is used for DP Observer search, i.e. one of the relevant application class has to implement the *Observer* interface and another one has to inherit from the *Observable* class, which is included in the JDK standard library. Such a class and an interface are well known and can be identified from the analysis of the application code.
- Match an interface with the classes that implement it. Starting from a given application, we get all interfaces. We observe where they are implemented and the behavior of the classes that implement the same interface. In case of DP Composite it will be useful to get the list of classes that implement a given interface. We will check if a specific class contains a list of attributes having as a type the same interface (a list of leafs). Other examples where this feature is used include: DP Factory Method and DP Abstract Factory: we search an interface that have a method whose return type is another interface.
- Match some attributes and/or methods. Some DPs are characterized by the presence of some specific attributes or properties concerning the methods (or the constructors). E.g. DP Singleton has a private constructor, a private static attribute of the same type of the class and a static method that returns an object of the same type of the class. A similar recognition can be done for Flyweight, Proxy, Object Pool (with a single Creator, like Singleton).

The recognition process can be optimized if, during the process, each class will collect all the useful features for identification.

More specifically, class SearchPattern has been used to perform searches. It takes as input a IJavaElement instance, (which is an Eclipse supertype for IJavaProject, IMethod, IPackageFragment, IPackageFragmentRoot, IType) on which a search has to be performed, and an int representing the matching mode (e.g. we could be searching for the initialisation of a reference or the use of a reference, in the method body). Class SearchRequestor holds found items, and interface IJavaSearchScope define where the search should be performed.

B. Conversion

To convert OO design patterns into AO versions, we proceed in two stages:

- Deleting the OO features related to the design patterns and/or reshaping of classes, attributes and methods;
- Creating the aspects.

After having recognized all the involved different classes of a project in Eclipse IDE, one or more ".aj" files implementing the DP are generated. They customize some different fields by using the class names. We use the Eclipse JDT to realize these stages. It allows us to write aspect code as simple text. Moreover, if we want to change an existing code we will use a more powerful tool that is the *Abstract Syntax Tree* (AST). This one parses the Java code in a XML tree and vice versa. This allows us to operate directly at the rows where we need the code changes.

IV. Proposed toolkit

We created a toolkit that provides us with support for recognizing and converting DPs. The main steps are as follows:

- Analysis of the existing code and search classes that implements a given DP;
- Creation of the object that will perform the conversion;
- Creation of the aspects;
- Changing the analysed code.

The toolkit declares the previous steps (abstract methods) and imposes them as a guide to the various refactoring. It applies the appropriate changes to the code.

A. Implemented Classes

The following describes the main classes implementing our toolbox. Class `AspectPrototype` supports the generation of an aspect, and holds the name of the aspects as well as the constituting portions, such as pointcuts, advices, etc., and the name of the file and package holding the code. The main methods for building such portions of code are: `addAttribute()`, `addPointcut()`, `addAdvice()`, `addMethod()`, and the last three take as input some strings representing name, parameters, and body.

Class `AspectDesignerPattern` is abstract and defines the methods that have to be implemented to recognise and convert a DP. Such a class holds the name of the classes, package, project, which have to be processed; moreover, it has methods searching features on the code, deleting portions of the found code, and creating portions of the new code. The main steps performed are described in more detail below.

- Method *createAspect()* creates a custom aspect for the found class; uses *AspectPrototype* class having the internal elements expected for an aspect (pointcuts, advice, attributes, etc.) and writes the code in the *.aj* file into the indicated package.
- Method *replaceCode()* deletes and/or modify all the portions of code characterizing a given DP in OO paradigm.

Class `SingletonModifier` is a specialisation for the namesake DP and recognises and convert a Singleton DP from OO to AO version. It holds an instance of AspectPrototype to generate the correspondent aspect; an instance of IMethod represents the constructor and another instance representing the getInstance() method typical of a Singleton. An instance of IField is also representative of the static attribute implemented in a Singleton class. The main methods implement the code for finding an implementation of Singleton within a class, to identify constructor and the static method, and to search and remove the invocation of static methods in classes using the Singleton.

Class `SingletonHandler` gives the possibility to connect the Eclipse environment with the proposed toolkit. It is an implementation of IObjectActionDelegate interface, having method run(), which will be called upon the click on the selected project. Such a class holds an instance representing the Eclipse shell, so that messages can be shown. The main additional method is responsible to call the methods provided by class `SingletonModifier`, in order to recognise and convert portions of code.

Class `SingletonSearchEngine` is a subclass of class `AbstractSearchEngine` and is used to find the calls to static methods.

Class `CompositeModifier` recognises and converts DP Composite from OO version to AO version. It mainly holds an instance of `AspectPrototype` in order to generate and write an appropriate aspect, realising the AO version, a reference to class playing as Component, a list of references for the classes playing as Child (Leaf) and the Composite. The main methods implement the logic to search for application classes implementing Composite DP.

V. Recognition rules

A. Singleton DP

The Singleton DP is a successful DP and developers are using it regularly. Now, our application will be concerned with the recognition and rewriting of the code, so that the AO version will be produced. In the recognition process, our algorithm consider three features that distinguish the DP Singleton, as follows.

- It exists a single private constructor;
- it exists a public static method that returns some object having the same type as the hosting class;
- it exists a private static attribute of the same type of the hosting class.

If a class satisfies the previous conditions, this will be represented as an instance of *SingletonModifier*, which implements the functionality of the abstract class *AspectDesignerPattern*. Code in Listing 1 shows the main steps of the identification. Then, once the class has been identified, it is constructed the relative aspect and replaced its code. In the end, the old parts of a Singleton class are deleted, i.e. the attributes and methods typical of the DP, and then the aj file is written.

After conversion of a Singleton class `MyClass`, all calls like

```
MyClass uniqueObject = MyClass.getInstance();
```

will be replaced by:

```
MyClass uniqueObject = new MyClass();
```

The client will see `MyClass` as a usual class (not a Singleton DP), however all the instantiations return the same instance of `MyClass`.

```
public final class SingletonModifier extends
     AspectDesignerPattern {

 private static IMethod getOnlyPrivateConstructor(
    IMethod[] methods) throws JavaModelException {

  int numOfPrivateConstructors = 0;
  int privateConstructorsIndex = -1;
  for (int i=0; i < methods.length; i++)
   if (methods[i].isConstructor() &&
      Flags.isPrivate(methods[i].getFlags())) {
    numOfPrivateConstructors++;
    privateConstructorsIndex = i;
   }
  if (numOfPrivateConstructors != 1) return null;
  return methods[privateConstructorsIndex];
 }

 private static IMethod getInstanceMethod(
   IMethod[] methods, String className) throws
        JavaModelException {

  for (int i=0; i < methods.length; i++)
   if (methods[i].getReturnType().
     compareTo("Q"+className+";") == 0 &&
     Flags.isPublic(methods[i].getFlags()) &&
     Flags.isStatic(methods[i].getFlags()))
    return methods[i];
  return null;
 }

 private static IField getInstanceAttribute(
  IField[] fields, String className) throws
       JavaModelException {

  for (int i=0; i < fields.length; i++)
   if (fields[i].getTypeSignature().
     compareTo("Q"+className+";") == 0 &&
     Flags.isStatic(fields[i].getFlags()))
    return fields[i];
  return null;
 }
}
```

Listing 1. Methods inspecting the characteristics for Singleton DP

B. Composite DP

The Composite DP allows developers to use a set of items as a single object through the implementation of a common interface. Every object can be a node (the element have a pointer to another Composite object) or a leaf (object that implements the concrete operation). For example: the I/O disk operation over folder or file can be processed in the same way (e.g. move, copy and rename). The folder contains one or more folders or files. The client (e.g. the explorer GUI) will handle the folder (Composite) and the files (Leaf) likewise (as Component).

As performed for the Singleton DP, our algorithm consider three feature that distinguish the Composite DP. For recognition of DP, we:

- get all interfaces of a project;
- for each interface, search all classes that implements it;
- obtained the class list of each interface, search: (i) the Composite class as a class that holds a list of type Component interface as an attribute, and (ii) the Leaf class, as a class implementing the Component interface.

As for *Singleton* recognition, if a group of classes implement a specific interface that satisfies the previous conditions, the interface and all classes recognized will be stored in an instance of *CompositeModifier*.

Code in Listing 2 shows the implementation of the above.

```
public final class CompositeModifier extends
     AspectDesignerPattern {

 public static AspectDesignerPattern[] getDesignerPattern(
      IJavaProject prj) {
  ArrayList<IType> leaves = new ArrayList<IType>();
  ArrayList<IType> composites = new ArrayList<IType>();
  Vector<Composite> compositeDP = new Vector<Composite>();
  Hashtable<IType,IField> c=new Hashtable<IType,IField>();
  if (prj == null) return null;
  ArrayList<IType> listInterface = new ArrayList<IType>();
  IPackageFragment[] pks = prj.getPackageFragments();
  for (IPackageFragment pack : pks)
   for (ICompilationUnit jf : pack.getCompilationUnits())
    for (IType classStub : jf.getTypes())
     if (Flags.isInterface(classStub.getFlags()))
      listInterface.add(classStub);
  AspectSearchEngine se = new AspectSearchEngine(prj);
  for (IType intf : listInterface) {
   se.search(intf, IJavaSearchConstants.IMPLEMENTORS,
    SearchPattern.R_ERASURE_MATCH |
    SearchPattern.R_CASE_SENSITIVE);
   Reference[] matches = se.getReferences();
   if (matches.length == 0) continue;
   Vector<IType> listType = new
    Vector<IType>(Arrays.asList(se.getUsingType(matches)));
   Iterator<IType> listTypeIterator = listType.iterator();
   while (listTypeIterator.hasNext()) {
    IType classStub = listTypeIterator.next();
    boolean subjFound = false;
    IField[] attributes = classStub.getFields();
    for (IField a : attributes)
     if (Composite.isDynamicList(a.getTypeSignature(),
        intf.getElementName())) {
      composites.add(classStub);
      listTypeIterator.remove();
      c.put(classStub, a);
      subjFound = true;
      break;
     }
    if (!subjFound) {
     leaves.add(classStub);
     listTypeIterator.remove();
    }
   }
   if (leaves.size() > 0)
    compositeDP.add(new Composite(prj, intf,
     composites, leaves, c));
  }
  AspectDesignerPattern[] adp = new
   AspectDesignerPattern[compositeDP.size()];
  compositeDP.toArray(adp);
  return adp;
 }
}
```

Listing 2. Methods inspecting the characteristics for Composite DP

After conversion, the list of Component instances hold inside the Composite class will be removed.

VI. Conclusion

After years of research, AOP begins to propose itself as a valid means for developing applications outside the academic context. However, for the last twenty years the development of software systems has undergone exponential growth that does not permit an immediate exchange of technology. Our work provides an approach that identifies parts of the code

that can be refactored from OO to AO version. The already well-formed and strong applications, thanks to twenty years of experience on DPs, give us the opportunity to pass to the aspect version. It maintains the advantages of the latter and is enriched by the AO modularity.

An important step during refactoring is the recognition of the used DPs and the identification of associated classes. We have identified three categories of recognition based on some typical signatures of DP.

Our future work aims at collecting the features of all DPs at once from the application under analysis and recognise one or more DPs on the basis of the presence of all the characteristics required by the DP. Only after recognition of all the DPs the creation of the aspect code and the replacement of the OO code will be performed.

REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of ECOOP*, ser. LNCS, vol. 1241. Springer, 1997, pp. 220–242.

[2] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in *Proceedings of OOPSLA*, vol. 37. ACM, 2002, pp. 161–173.

[3] R. Giunta, G. Pappalardo, and E. Tramontana, "Aodp: Refactoring code to provide advanced aspect-oriented modularization of design patterns," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Riva del Garda, Italy, March 2012, pp. 1243–1250.

[4] ——, "Aspects and annotations for controlling the roles application classes play for design patterns," in *Proceedings of IEEE Asia-Pacific Software Engineering Conference (APSEC)*, Ho Chi Minh, Vietnam, December 2011, pp. 306–314.

[5] ——, "Superimposing roles for design patterns into application classes by means of aspects," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Riva del Garda, Italy, March 2012, pp. 1866–1868.

[6] ——, "Using aspects and annotations to separate application code from design patterns," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Sierre, Switzerland, March 2010, pp. 2183–2189.

[7] A. Calvagna and E. Tramontana, "Delivering dependable reusable components by expressing and enforcing design decisions," in *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, Kyoto, Japan, July 2013, pp. 493–498.

[8] F. Bannò, D. Marletta, G. Pappalardo, and E. Tramontana, "Tackling consistency issues for runtime updating distributed systems," in *Proceedings of IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Atlanta, Georgia, US, 2010, pp. 1–8.

[9] G. Pappalardo and E. Tramontana, "Suggesting extract class refactoring opportunities by measuring strength of method interactions," in *Proceedings of IEEE Asia Pacific Software Engineering Conference (APSEC)*, Bangkok, Thailand, December 2013, pp. 105–110.

[10] E. Tramontana, "Automatically characterising components with concerns and reducing tangling," in *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, Kyoto, Japan, July 2013, pp. 499–504.

[11] M. Mongiovi, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana, "Combining static and dynamic data flow analysis: a hybrid approach for detecting data leaks in Java applications," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Salamanca, Spain, 2015, pp. 1573–1579.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] J. Kerievsky, *Refactoring to patterns*. Addison-Wesley, 2005.

[14] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[15] J. Dong, Y. Sun, and Y. Zhao, "Design pattern detection by template matching," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, 2008, pp. 765–769.

[16] E. Gamma, R. Helm, R. Johnson, and R. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[17] R. Laddad, *AspectJ in Action*. Grennwich, Conn.: Manning Publications Co., 2003.

[18] G. Pappalardo and E. Tramontana, "Automatically discovering design patterns and assessing concern separations for applications," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Dijon, France, April 2006, pp. 1591–1596.

[19] J. Dong, Y. Zhao, and Y. Sun, "A matrix-based approach to recovering design patterns," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 39, no. 6, 2009.