Implementing Patterns with Annotations

Gabriele Kahlout
Free University of Bozen
g@dp4j.com

ABSTRACT

There are few CASE tools that support design patterns implementations. I propose a tool that **injects patterns implementation** boilerplate code, **validates** it, and **refactors** to it. The tool aims to be highly usable through intention-revealing **annotations** as the primary **declarative** interface, and by integration with the familiar IDE editor for feedback.

Keywords

Design Patterns, Refactoring, annotations processing

1. INTRODUCTION

Many developers implement patterns by writing the implementation code anew, rather than using a tool to generate the code[1], or on a library to provide the implementation code[2,3]. This could be attributed to the lack of usable tools, although descriptions and proof-of-concept implementations of such tools abound[4,5,6,7,8] in the research community.

They were also published before annotations[9] support in Java. Meffert [10] proposes annotations as documentation artifacts, and effective-annotations[11] and jpatterns[12] are implementations. Guice[13] provide a @Singleton annotation for dependency injections.

Hartmann et al. [14], described a pattern enforcing compiler (PEC) which allows classes to be marked as having a given pattern through Java syntax (such as *implements Singleton*).

I propose dp4j[15] with which developers declare with annotations the patterns they intend to implement, or claim the code implements. At its core the tool is an annotations processor for the Java Compiler that guided by annotations validates implementations and generates boilerplate code[16] injected in the AST tree, or to the source file.

This paper discusses the functions of the tool, usability enhancements through IDE integration, and motivating examples applied to production open-source code.

2. DECLARE WITH ANNOTATION PROCESSORS

Re-use pattern implementation boilerplate code to focus, instead, on domain code.

2.1 Motivation

Re-using implementation code through pattern libraries is limited in utility because implementations often prescribe a structure on the source code; libraries can only synthesize the content of elements in such structure by hiding some code behind calls to their routines. Likewise, framework interfaces help more in documenting the implementation than implementing it.

A special compiler, such as PEC, that gives special meaning to language constructs declaring implementations is not a portable solution, and embedding such support for patterns in the standard compiler might not be desirable for all users of the compiler.

---

[1] Emerson Murphy-Hill and Andrew P. Black, Why Don't People Use Refactoring Tools? in In Proceedings of the 1st Workshop on Refactoring Tools (Berlin, Germany, 2007).

[2] Loki is a C++ library that implements Abstract Factory, Command, Factory Method, Singleton, and Visitor pattern making extensive use of C++ template metaprogramming. http://loki-lib.sourceforge.net/

[3] Perfectjpattern is a Java library implementing design patterns. http://sourceforge.net/projects/perfectjpattern

[4] J Rajesh and D Janakiram, JIAD: A Tool to Infer Design Patterns in Refactoring. in Proc. of PPDP (Verona, Italy, 2004)

[5] Cinneide, M.O., Automated refactoring to introduce design patterns. in Software Engineering, 2000. Proceedings of the 2000 International Conference on, (Limerick , Ireland, 2000)

[6] Jeon, S., Lee, J-S. and Bae, D-H. An Automated Refactoring approach to design pattern-based program. Transformations in Java Programs. (Korea Advanced Institute of Science and Technology, 2002)

[7] Tokuda, L. and Batory, D. Automated Software Evolution via Design Pattern Transformations. 1995.

[8] Carmen Zannier and Frank Maurer, Tool support for complex refactoring to design patterns.XP'03 Proceedings of the 4th international conference on Extreme programming and agile processes in software engineering, (Heidelberg, 2003), Springer-Verlag Berlin, 123-130.

[9] http://download.oracle.com/javase/tutorial/java/javaOO/annotations.html

[10] Meffert, K. Supporting design patterns with annotations, (Potsdam, 2006), Technical University, Ilmenau, 8 pp.-445. Sabo, M. and Poruban, J. Preserving Design Patterns using Source Code Annotations, (Kosice, Slovak Republic, 2009), Technical University of Kosice, 53-56.

[11] http://code.google.com/p/effective-annotations/

[12] http://www.jpatterns.org

[13] http://code.google.com/p/google-guice/wiki/Scopes

[14] Hartmann, S. and Stumptner, M. APCCM '05 Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling. A pattern-enforcing compiler (PEC) for Java: using the compiler. (Darlinghurst, Australia, 2005), Australian Computer Society, Inc. 69-78.

[15] http://dp4j.com

[16] http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

2.2 Solution: Inject Implementations with Annotation Processors

Annotations were introduced with JDK5 and can be added as modifiers to class, methods and fields declarations. They are as portable as other constructs.

During compilation annotation processors inject in the AST, or source files, the compiler boilerplate implementation code for the annotated elements (classes, methods and fields). The Java Compiler then generates the bytecode of the resulting AST.

It's argued that intuitively named annotations (e.g. in jpatterns, annotations reflect the UML structure diagrams of the GoF patterns[17]) are easier to learn than library interface counterparts.

2.3 Motivating Example: Singleton Pattern

The bytecode for Code Snippet 1 and Code Snippet 2 is equivalent, since dp4j injects an annotated singleton instance, and a getter in the AST of Code Snippet 1.

```
@com.dp4j.Singleton
public class ContentProducer extends
MeaningsDatabase{

    private ContentProducer() {
        super();
        checkContentStatus();
...
```

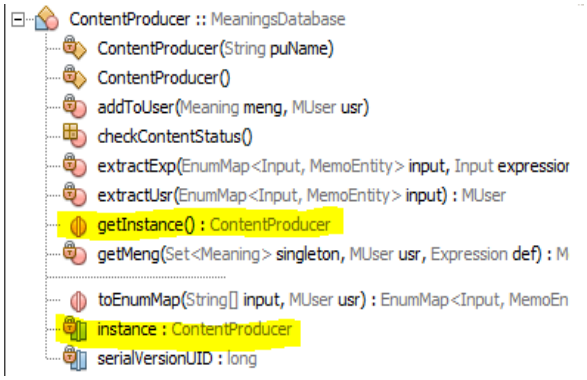
Code Snippet 1. Shows example usage of @Singleton annotation on a class from MemorizEasy source code[18]

```
import org.jpatterns.gof.*;
@SingletonPattern
public class ContentProducer extends
MeaningsDatabase{

    @SingletonPattern.Singleton
    private static final ContentProducer
instance = new ContentProducer();

    @SingletonPattern.SingletonMethod
    public static ContentProducer
getInstance() {
        return instance;
    }

    private ContentProducer() {
        super();
        checkContentStatus();
...
```

Code Snippet 2. Shows a Singleton implementation completely specified in the source file.

2.4 IDE Integration

Injecting implementation code directly in the AST trades transparency for increased domain code visibility. Through IDE integration some transparency can be gained by displaying the signature of injected members in an overview representation of the class, as shown below.



Screenshot 1. Shows NetBeans IDE[19] class members navigator. The highlighted members were injected in the AST.

Clicking on the member node in the illustrated class representation redirects to the member declaration in the source file; for injected members the generated code could be shown.

3. VALIDATE IMPLEMENTATIONS WITH ANNOTATION PROCESSORS

Validate patterns implementations against the patterns specification to maintain an understanding of the code as it evolves.

3.1 Motivation

As a program evolves (and hacks are coded) the implementation may depart from that of the documented pattern introducing subtle bugs, theoretically warranted against by the pattern. It requires a great deal of time and effort to identify changes to the code that violate a pattern implementation and update the documentation, especially by the authors of such changes.

3.2 Solutions: Code Analysis with Annotation Processors

Although there may be several different implementations for a pattern, there's a specification subset common to all[20] (e.g. a Builder must provide a method or field for retrieving the built product instance, a Template Method must invoke an abstract method, there must be at most one Singleton instance at any one time, etc.). Such logic is embedded in pattern detection tools[21].

Since during the annotations processing phase processors access the program code, they can analyze it and report violations

---

[17] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. AddisonWesley Professional, 1994.)

[18] www.memorizesy.com

[19] www.netbeans.org

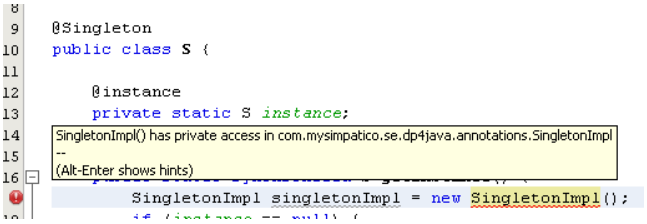
[20] Dong, J., Alencar, P. and Cowan, D. Formal Specification and Verification of Design Patterns. Idea Group Inc., 2007.

[21] Von Detten, M. and Platenius, M. Improving Dynamic Design Pattern Detection in Reclipse with Set Objects. Eindhoven University of Technology, The Netherlands, 2009.

through the compiler reporting mechanism. Doing so, and when annotations are used to document[22] implementations, the processor also validates documentation accuracy.

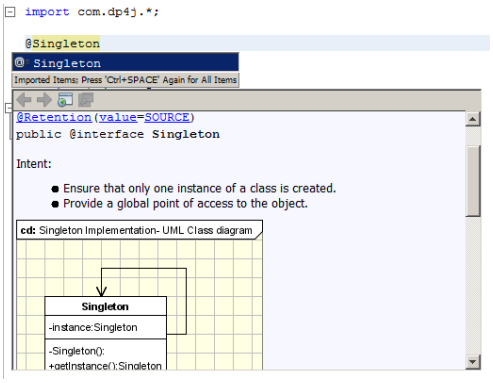
3.3 IDE Integration

IDEs can report errors from the annotations processor, as illustrated below in NetBeans, to provide contextual feedback.



Screenshot 2: IDE reports that SingletonImpl may not be instantiated since its default constructor is private.

The IDE feature of displaying the JavaDoc of annotations provides immediate access to the annotated pattern documentation, and is a built-in alternative to the tool described by Budinsky et al.[23].



Screenshot 3. Shows @Singleton JavaDoc in an IDE pop-up.

3.4 Motivating Example: Singleton Pattern

For the following snippets dp4j verifies (also processes effective-annotations and jpattern's annotations) that ContentProducer:
- has only private constructors (including the default);
- has exactly one static instance variable of type ContentProducer; This must be either private or final (or both); If private a public static method must return that instance; If not final it must initialize it there, only when null.

[22] http://download.oracle.com/javase/tutorial/java/javaOO/annotations.html

[23] Budinsky, F., Finnie, M. and Yu, P. Automatic Code Generation from Design Patterns. Addison-Wesley Publishing Company, 1995. www.research.ibm.com/designpatterns/pubs/codegen.pdf

4. REFACTORING WITH ANNOTATION PROCESSORS

Identify opportunities for refactoring to a pattern and enable them.

4.1 Motivation

Refactoring to Patterns[24] suggests that using patterns to improve an existing design is better than using patterns early in a new design. However inherent to refactoring is the risk of inadvertently changing the program external behavior, while incremental code changes can be demotivating since they are short-lived artifacts mitigating risks.

4.2 Solution: Just In Time Implementation

By injecting the pattern implementation only in the AST, as opposed to committing to the implementation in the source code, it becames easier to refactor away from that implementation and towards a different implementation since there's less code to handle. JITI is like re-doing it all over, but it's the tool that does it.

4.3 Motivating Example: Singleton Pattern

Considering the code in Code Snippet 1, the previous discussion implied that dp4j default behavior is to in-line initialize Singleton instances. Consider a refactoring that lazily initializes[25] the Singleton. The following snippet illustrates the refactoring. Crossed code is removed, while code in bold is added.

```
import org.jpatterns.gof.*;
@SingletonPattern
public class ContentProducer extends
MeaningsDatabase {

@SingletonPattern.Singleton
private static ~~final~~ ContentProducer
instance ~~= new ContentProducer();~~

    private ContentProducer() {
        super();
        checkContentStatus();
    }

@SingletonPattern.SingletonMethod
public static synchronized ContentProducer
getInstance() {
    if(instance == null){
        instance = new ContentProducer();
    }
    return instance;
}
...
```

Code Snippet 3. Shows Code Snippet 2 refactored to lazy initialization Singleton.

[24] Kerievsky, J. Refactoring to Patterns. Addison Wesley, 2004. http://www.industriallogic.com/xp/refactoring

[25] Stencel Krzysztof and Tari Zahir. Implementation Variants of the Singleton Design Pattern. Springer Berlin, Heidelberg, 2008.

While the refactoring is simple, it's again boilerplate code the developer can be abstracted from with dp4j `@Singleton lazy` attribute, similar to Lombok's `@Getter lazy`[26].
A developer refactoring from Code Snippet 1 would only need to add the lazy element to perform the refactoring, and remove it to refactor back.

```
@Singleton(lazy=true)
public class ContentProducer extends
MeaningsDatabase {
....
```

Code Snippet 4. Shows code Snippet 1 refactored to lazy initialization Singleton.

5. FORM TEMPLATE METHOD WITH ANNOTATION PROCESSORS

Identify Template Method candidates and refactor to them.

5.1 Motivation

Form Template Method Refactoring is very useful since it enables sharing code that couldn't otherwise be shared refactoring with pull-up method refactoring[27] alone. CASE support for such refactoring makes it more accessible and common-practice.

5.2 Motivating Example: Template Method

Snippets below show the code before and after the refactoring.

```
public class LabelServlet extends
HttpServlet {

@Override
protected void doGet(HttpServletRequest
request, HttpServletResponse response)
throws ServletException, IOException {
        final String referrer =
request.getHeader("referer");
        final String operation =
request.getParameter(Constants.operation);
        final HttpSession session =
request.getSession();
        final MUser usr = (MUser)
session.getAttribute(Constants.usr);
      authorize(response, usr);
   if (labelName != null &&
!operation.contains(Constants.newLabel)) {
   label = Database.getLabel(labelName,
labelOwner, con);
   }else{
   label = new Label(labelName, labelOwner);
   }
   labelHandlers.get(operation).execute(labe
l, usr, con, request,   response);
        final String redirectAddress =
processRequest(request, operation, usr,
referrer);
      response.sendRedirect(forwardAddress);
....

@Override
public class MengServlet extends HttpServlet
{

protected void doGet(HttpServletRequest
request, HttpServletResponse response)
throws ServletException, IOException {
        final String referrer =
request.getHeader("referer");
        final String operation =
request.getParameter(Constants.operation);
        final HttpSession session =
request.getSession();
        final MUser usr = (MUser)
session.getAttribute(Constants.usr);

        authorize(response, usr);
       if (operation == null &&
isMultipart) {
            importWords(request, usr, con);
        } else if (operation != null &&
operation.equals(Constants.insert)) {
            final String labelName =
request.getParameter(Constants.labelName);
            insertWords(request, response,
usr, con, labelName);
        final String redirectAddress =
processRequest(request, operation, usr,
referrer);
        response.sendRedirect(forwardAddress
);
    }
...
```

Code Snippet 5. Shows a method implemented by 2 subclasses of HttpServlet in MemorizEasy source code where the only difference in the overridden methods is shown in bold.

```
public abstract class
LabelMengSuperClassServlet extends
HttpServlet{

@Override
public final void doGet(HttpServletRequest
request, HttpServletResponse response)
throws ServletException, IOException {
    final String referrer =
request.getHeader("referer");
    final String operation =
request.getParameter(Constants.operation);
    final HttpSession session =
request.getSession();
    final MUser usr = (MUser)
session.getAttribute(Constants.usr);
    authorize(response, usr);
    servletWork(operation, usr, session,
request);
    final String redirectAddress =
processRequest(request, operation,
usr,            referrer);
    response.sendRedirect(forwardAddress);
}
```

[26] http://projectlombok.org/features/GetterLazy.html

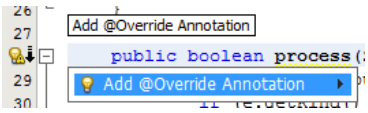
[27] http://www.refactoring.com/catalog/pullUpMethod.html

```
protected abstract void
templateMethod(String operation, MUser usr,
String referrer);
...
public class LabelServlet extends
HttpServlet {

@Override
protected void doGet(HttpServletRequest
request, HttpServletResponse response)
throws ServletException, IOException {
        final String referrer =
request.getHeader("referer");
        final String operation =
request.getParameter(Constants.operation);
        final HttpSession session =
request.getSession();
        final MUser usr = (MUser)
session.getAttribute(Constants.usr);
            authorize(response, usr);

    if
(operation.contains(Constants.newLabel)) {
    label = Database.getLabel(session);
    }else{
    label = new Label(usr);
    }
    labelHandlers.get(operation).execute(labe
l, usr, con, request);

    final String redirectAddress =
processRequest(request, operation, usr,
referrer);
    response.sendRedirect(forwardAddress);

protected void servletWork(String operation,
MUser usr, String referrer){
if (operation.contains(Constants.newLabel))
{
   label = Database.getLabel(session);
   }else{
   label = new Label(usr);
   }
   labelHandlers.get(operation).execute(labe
l, usr, con, request);
}
....

@Override
public class MengServlet extends HttpServlet
{

protected void doGet(HttpServletRequest
request, HttpServletResponse response)
throws ServletException, IOException {
        final String referrer =
request.getHeader("referer");
        final String operation =
request.getParameter(Constants.operation);
        final HttpSession session =
request.getSession();
        final MUser usr = (MUser)
session.getAttribute(Constants.usr);

        authorize(response, usr);
            if (operation == null &&
isMultipart) {
            importWords(request, usr, con);
        } else if (operation != null &&
operation.equals(Constants.insert)) {
            final String labelName =
request.getParameter(Constants.labelName);
            insertWords(request, response,
usr, con, labelName);
        final String redirectAddress =
processRequest(request, operation, usr,
referrer);
        response.sendRedirect(forwardAddress
);
    }

protected void servletWork(String operation,
MUser usr, String referrer){
   if (operation == null && isMultipart) {
      importWords(request, usr, con);
   } else if (operation != null &&
operation.equals(Constants.insert)) {
            final String labelName =
request.getParameter(Constants.labelName);
            insertWords(request, response,
usr, con, labelName);
}
...
```

Code Snippet 6. Shows Code Snippet 5 after applying the refactoring.

5.3 IDE Integration

Using the IDE's standard suggestions interface, shown below, the suggestion could be "Form Template Method", since the steps can be fully automated. NetBeans supports the refactoring steps individually.



Screenshot 4. Shows an annotation hint.

6. FUTURE WORK

The tool could support at least all the GoF patterns. There are also several annotations in FindBugs[28], which could be supported and complemented by dp4j to enable refactoring.
Better integration with IDEs could also be pursued, as discussed.

7. CONCLUSION

This paper discussed the application of annotation processors to inject and validate patterns implementation code, and to automate refactoring to patterns. The contributions include:
claiming novel applications of annotation processors to inject patterns implementations declared through annotations, perform code analysis to validate implementations and automate refactorings; proposing annotations and IDE integration as usable interfaces to such applications; and motiving them through real-life example code.

[28] David H. Hovemeyer and William W. Pugh. FindBugs™. University of Maryland, Maryland, 2009.