TTC'15 Case Study: Transformation of Java Annotations

Filip Křikava

INRIA Lille

France

filip.krikava@inria.fr

Martin Monperrus

University Lille 1 / INRIA Lille

France

martin.monperrus@univ-lille1.fr

Java 5 introduced annotations as a systematic mean to syntactic meta-data to various elements of Java source code. They are extensively used by various frameworks to conveniently extend behavior of Java programs that would otherwise have to be done manually or synthesized from an external resource. These frameworks often use reflection or instrument Java classes through aspect-oriented techniques or a direct bytecode modification. However, class level instrumentation might not always be available or desirable.

In this case study we focus on source level transformation. Concretely, the task is to inlines behavior specified by an annotation library that encapsulates common programming concerns such as logging, caching and retry on a failure. The objective is to explore how current transformation tools are suitable for programming language transformations.

1 Introduction

Java 5 has been extended with annotations that provide a convenient way to supply meta-data to various element of Java source code such as classes and methods. Since then, annotations have been extensively used by a number of Java libraries, frameworks and tools. One of the advantage of using annotations is that they attach syntactic meta-data directly to the relevant element of Java source code rather than decoupling them to external resources or setting up manually through an API.

Among other things, annotations can be used to express cross cutting concerns such as logging or caching that would otherwise cluttered the source code as they are notably difficult to abstract using regular programming techniques. Essentially, an annotation acts as marker for a transformation tool which applies the expressed concern into the corresponding Java element. Majority of these tools rely on Java reflection API or some sort of class instrumentation using aspect-oriented programming (AOP) techniques or direct byte code manipulation to process the these annotations. While this is usually the preferred way, in some cases the class level instrumentation might not available or desirable. The problem is that it might (i) lead to leaky abstraction since the transformation is transparent to a developer, (ii) introduce some performance penalties, or (iii) bring some extra non-trivial dependencies.

In this case study we focus on the situation when class level instrumentation is not possible and develop a solution which is using source level transformation. Concretely, we present a transformation task whose objective is to inline behavior specified by a sample annotation library that covers

common programming concerns such as logging, caching and retry on a failure. The objective is to explore how current transformation tools are suitable for programming language transformations.

In the next section we present the sample annotation library and describe the tasks of the case study in more detail. After that, Section 3 overviews criteria that will be used to evaluate submitted solutions to this case study.

All supporting material, as well as this document, is available on the case study github page `https://github.com/fikovnik/ttc15-tranj-case`. The contestants are invited to submit an issue shall there be any ambiguity about the transformation tasks that shall be carried in this case study.

@Martin: cite some AOP / annotations paper?

2   Case Description

As the annotation library, we use a simplified version of `jcabi-aspects`[1] which is a collection Java annotations together with AOP-based processing tool which allow developers to conveniently apply some common cross-cutting concerns into Java application. From all the `jcabi-aspects` annotations, we have selected the following:

@RetryOnFailure for an automated retry of failed method execution.

@Cacheable for a simple data caching of parameterless methods return values.

@Loggable for an automated logging of method calls including method entry, exit and any exception that is caught in the method body.

The core transformation involves traversing a source Java class and extending the annotated methods with the behavior specified by the above annotations. The details of each annotation including the specification of its arguments are provided in its associated javadoc[2].

In the following we provide an overview of the transformation tasks processing one annotation at a time. The transformation is described using an example rather than with a formal definition. While we believe that it should be enough, shall there be any ambiguity, we invite the reader to raise an issue on the case study github page. In the last task, we look on the case of combining these annotations together and optimizing the resulting Java source code.

The complete code for these transformation tasks is provide on the github page in the `ttc15.tranj.examples` package.

2.1   Running Example

To better illustrate the transformation tasks of this case study, we use the following example. In order to keep the case study concise and clear, we use only a single example and omit the use of

---

[1]`http://aspects.jcabi.com/index.html`
[2]`https://github.com/fikovnik/ttc15-tranj-case/tree/master/src/ttc15-tranj/src/main/java/ttc15/tranj/annotation`

any external library.

Let us consider a class that can download a content of an URL. A possible Java implementation is shown in Listing 1.

```java
public class URLDownload {
    private final URL url;

    public URLDownload(String url) throws MalformedURLException {
        this.url = new URL(url);
    }

    public byte[] get() throws IOException {
        try (InputStream input = url.openStream()) {

            ByteArrayOutputStream buffer = new ByteArrayOutputStream();
            byte[] chunk = new byte[4*1024];
            int n;

            while ((n = input.read(chunk)) > 0 ) {
                buffer.write(chunk, 0, n);
            }

            return buffer.toByteArray();
        }
    }
}
```

Listing 1: Basic version of `URLDownload` class.

Now, when we have the basic functionality, we would like to extend it with following features:

– *Failure handling.* The method should become more robust and accommodate for some of the inevitable network delays by retrying the download in case an exception occurs. It should however only retry the call in the case when it makes sense–*i.e.* when it is possible to recover. For example, in the case of `UnknownHostException` exception, it should fail immediately.

– *Caching.* The URL should not be consulted every single time the method is called, but instead it should keep the content for certain amount of time in memory.

– *Logging.* Each call to the method should be logged. The logging message should present the state of the current instance (`this.toString()`) as well as how long the invocation took.

Instead of coding this manually, producing code such as the one shown in the listing in Section A, we would like to use annotations to declaratively specify the above concerns and have the a transformation tool automatically synthesize code similar to the shown in Section B. Concretely, the only change to the original code should the annotating the method with the following annotations:

```java
@RetryOnFailure(attempts = 3, delay = 1000, escalate = { UnknownHostException.class })
@Cacheable(lifetime = 1000)
@Loggable
public byte[] get() throws IOException { ... }
```

2.2   Task 1: Retrying on a Failure

The first transformation tasks should handle the `@RetryOnFailure` annotation. The objective is extend a given method with a simple failure handling strategy that retries failed method invocations. Annotating the `URLDownload.get()` method with

```
1 @RetryOnFailure(attempts = 3, delay = 1000, types = { IOException.class }, escalate = {
  ↪   UnknownHostException.class })
```

should produce the following code:

```java
1 public byte[] get() throws IOException {
2   int __retryCount = 0;
3
4   while (true) {
5     try {
6       // the content of the original method
7     } catch (UnknownHostException e) {
8       throw e;
9     } catch (IOException e) {
10      __retryCount += 1;
11
12      if (__retryCount > 3) {
13        throw e;
14      } else {
15        try {
16          Thread.sleep(1000);
17        } catch (InterruptedException e1) {
18          throw e;
19        }
20      }
21    }
22  }
23 }
```

The original method code–*i.e.* what is between the lines 9–20 in Listing 1, is wrapped by another try-catch block which is itself in a while loop. The number of attempts (3) is projected in the condition `if (__retryCount > 3)`, the delay is translated in a current thread sleep on lines 15–19. Finally, we make a distinction in the exceptions that on which the call should be retried (`IOException`) and the ones that should be escalated (`UnknownHostException`) in the generated catch clauses in line 9 and 7 respectively.

2.3   Task 2: Caching

The second transformation tasks should handle the `@RetryOnFailure` annotation. The objective is extend a given method with a simple failure handling strategy that retries failed method invocations. Annotating the `URLDownload.get()` method with

```
1 @Cacheable(lifetime = 1000)
```

should produce the following code:

```
1  public class URLDownload {
2    private long __getCacheLastAccessed = 0;
3    private byte[] __getCacheContent = null;
4
5    ...
6
7    public byte[] get() throws IOException {
8      if (System.currentTimeMillis() - __getCacheLastAccessed < 1000 && __getCacheContent
         ↪   != null) {
9        return __getCacheContent;
10     }
11
12     try (InputStream input = url.openStream()) {
13       ByteArrayOutputStream buffer = new ByteArrayOutputStream();
14       byte[] chunk = new byte[4 * 1024];
15       int n;
16
17       while ((n = input.read(chunk)) > 0) {
18         buffer.write(chunk, 0, n);
19       }
20
21       __getCacheContent = buffer.toByteArray();
22       __getCacheLastAccessed = System.currentTimeMillis();
23
24       return __getCacheContent;
25     }
26   }
27 }
```

The __getCacheLastAccessed and __getCacheContent defined on line 2 and 3 are cache book-keeping fields used for storing the information about the last access time and method result respectively. The condition on line 8 checks the validity of the cache based on the annotation lifetime. Finally, the method exit point (**return**) is replaced with the stored result on lines 22–25. This should be done for all method exit points.

2.4 Task 3: Logging

The last annotation transformation adds logging. Any method annotated with @Loggable should have conditionally (depending on the annotation options) logged (i) its entry point, (ii) all its exit points, and (iii) all exceptions.

Annotating the URLDownload.get() method with

```
1  @Loggable
```

should produce the following code:

```
1  public class URLDownload {
2    // added logger
3    private final org.slf4j.Logger __logger =
       ↪   org.slf4j.LoggerFactory.getLogger(URLDownload.class);
4
5    ...
6
7    public byte[] get() throws IOException {
8      // added entry point logging
9      long __entryTime = System.currentTimeMillis();
```

```
10      if (__logger.isTraceEnabled()) {
11        __logger.trace(String.format("get() [url='%s']: entry", url));
12      }
13
14      try (InputStream input = url.openStream()) {
15
16        ByteArrayOutputStream buffer = new ByteArrayOutputStream();
17        byte[] chunk = new byte[4 * 1024];
18        int n;
19
20        while ((n = input.read(chunk)) > 0) {
21          buffer.write(chunk, 0, n);
22        }
23
24        // added exit point logging
25        if (__logger.isTraceEnabled()) {
26          __logger.trace(String.format("get(): exit in %d ms", System.currentTimeMillis() -
             ↪   __entryTime));
27        }
28        return buffer.toByteArray();
29      }
30    }
31 }
```

Line 2 defines an instance of SLF4J logger[3]. Lines 7–10 and 22-24 insert the entry and exit point logging.

Because, there is no exception handling code–*i.e.* no catch clause, only entry and exit points are traced. If we would however apply the logging annotation on the method shown as output of @RetryOnFailure, the result should look like:

```
1 while (true) {
2   try {
3     ...
4   } catch (UnknownHostException e) {
5     // added exception handling
6     __logger.error("get(): exception", e);
7
8     throw e;
9   } catch (IOException e) {
10    // added exception handling
11    __logger.error("get(): exception", e);
12
13    __retryCount += 1;
14
15    if (__retryCount > 3) {
16      throw e;
17    } else {
18      try {
19        Thread.sleep(1000);
20      } catch (InterruptedException e1) {
21        // added exception handling
22        __logger.error("get(): exception", e);
23
24        throw e;
25      }
26    }
27  }
28 }
```

---

[3]http://www.slf4j.org/

2.5   Task 4: Annotation composition

It should be possible to have all the extension present together on one method:

```
1 @RetryOnFailure(attempts = 3, delay = 1000, escalate = { UnknownHostException.class })
2 @Cacheable(lifetime = 1000)
3 @Loggable
4 public byte[] get() throws IOException { ... }
```

The code transformation should be performed in order–*i.e.*, first applying `@RetryOnFailure`, then `@Cacheable` and finally adding `@Loggable`. The expected result is shown in Section B.

2.6   Extension 1: Optimization (*optional*)

A possible extension to this case study is to have the annotations being aware of one another. In this case, the order of the annotations does not matter and they will be always applied in the order specified in the task 4. For the caching and retry on failure, nothing will change, but the logging will recognize the generated code and adjust the logging messages appropriately. The expected result–*i.e.* placement and message content is shown in the Section A.

3   Evaluation Criteria

3.1   Correctness and Completeness

TBW

3.2   Quality Attributes

- *Usability*
- *Visibility and Expressibility*
- *Complexity*
- *Performance*

TBW

A   Source Code of `URLDownloader` Coded Manually

```
1 package ttc15.tranj.examples;
2
3 import java.io.ByteArrayOutputStream;
4 import java.io.IOException;
5 import java.io.InputStream;
6 import java.net.MalformedURLException;
7 import java.net.URL;
```

```java
8    import java.net.UnknownHostException;

9
10   import org.slf4j.Logger;
11   import org.slf4j.LoggerFactory;

12
13   public class ManualURLDownload {
14     private static final long CACHE_TIMEOUT = 1000;
15     private static final int MAX_RETRY = 3;

16
17     private final Logger logger = LoggerFactory.getLogger(ManualURLDownload.class);

18
19     private long lastAccessed = 0;
20     private byte[] cachedContent = null;
21     private final URL url;

22
23     public ManualURLDownload(String url) throws MalformedURLException {
24       this.url = new URL(url);
25     }

26
27     public byte[] get() throws IOException {
28       long entryTime = System.currentTimeMillis();

29
30       if (logger.isTraceEnabled()) {
31         logger.trace(String.format("get() [url='%s']: entry", url));
32       }

33
34       if (System.currentTimeMillis() - lastAccessed < CACHE_TIMEOUT && cachedContent !=
          ↪  null) {
35         if (logger.isTraceEnabled()) {
36           logger.trace(String.format("get(): exit cached [%d ms]",
              ↪  System.currentTimeMillis() - entryTime));
37         }
38         return cachedContent;
39       }

40
41       int retryCount = 0;

42
43       while (true) {
44         try (InputStream input = url.openStream()) {

45
46           ByteArrayOutputStream buffer = new ByteArrayOutputStream();
47           byte[] chunk = new byte[4*1024];
48           int n;

49
50           while ((n = input.read(chunk)) > 0 ) {
51             buffer.write(chunk, 0, n);
52           }

53
54           cachedContent = buffer.toByteArray();
55           lastAccessed = System.currentTimeMillis();

56
57           if (logger.isTraceEnabled()) {
58             logger.trace(String.format("get(): exit [%d ms]", System.currentTimeMillis() -
                ↪  entryTime));
59           }
60           return cachedContent;
61         } catch (UnknownHostException e) {
62             logger.error("get(): exception -> forced exit", e);
```

```
63        throw e;
64      } catch (IOException e) {
65        if (logger.isWarnEnabled()) {
66          logger.warn("get(): exception -> retrying", e);
67        }
68
69        retryCount += 1;
70
71        if (retryCount > MAX_RETRY) {
72          if (logger.isTraceEnabled()) {
73            logger.trace("get(): max retry count reached");
74          }
75          throw e;
76        } else {
77          try {
78            Thread.sleep(1000);
79          } catch (InterruptedException e1) {
80            if (logger.isWarnEnabled()) {
81              logger.warn("get(): interrupted while waiting to retry");
82            }
83            throw e;
84          }
85          if (logger.isTraceEnabled()) {
86            logger.trace("get(): retrying");
87          }
88        }
89      }
90    }
91  }
92 }
```

B Source Code of Synthesized `URLDownloader`

```java
1   package ttc15.tranj.examples;
2
3   import java.io.ByteArrayOutputStream;
4   import java.io.IOException;
5   import java.io.InputStream;
6   import java.net.MalformedURLException;
7   import java.net.URL;
8   import java.net.UnknownHostException;
9
10  import org.slf4j.Logger;
11  import org.slf4j.LoggerFactory;
12
13  public class SynthesizedURLDownload {
14    private final Logger __logger = LoggerFactory.getLogger(SynthesizedURLDownload.class);
15    private long __getCacheLastAccessed = 0;
16    private byte[] __getCachedContent = null;
17
18    private final URL url;
19
20    public SynthesizedURLDownload(String url) throws MalformedURLException {
21      this.url = new URL(url);
22    }
23
```

```java
public byte[] get() throws IOException {
  long __entryTime = System.currentTimeMillis();
  if (__logger.isTraceEnabled()) {
    __logger.trace(String.format("get() [url='%s']: entry", url));
  }

  if (System.currentTimeMillis() - __getCacheLastAccessed < 1000 && __getCachedContent
  ↪  != null) {
    if (__logger.isTraceEnabled()) {
      __logger.trace(String.format("get(): exit [%d ms]", System.currentTimeMillis() -
      ↪  __entryTime));
    }
    return __getCachedContent;
  }

  int __retryCount = 0;
  while (true) {
    try (InputStream input = url.openStream()) {

      ByteArrayOutputStream buffer = new ByteArrayOutputStream();
      byte[] chunk = new byte[4*1024];
      int n;

      while ((n = input.read(chunk)) > 0 ) {
        buffer.write(chunk, 0, n);
      }

      __getCachedContent = buffer.toByteArray();
      __getCacheLastAccessed = System.currentTimeMillis();

      if (__logger.isTraceEnabled()) {
        __logger.trace(String.format("get(): exit [%d ms]", System.currentTimeMillis()
        ↪  - __entryTime));
      }
      return __getCachedContent;
    } catch (UnknownHostException e) {
      __logger.error("get(): exception", e);
      if (__logger.isTraceEnabled()) {
        __logger.trace(String.format("get(): exit [%d ms]", System.currentTimeMillis()
        ↪  - __entryTime));
      }
      throw e;
    } catch (IOException e) {
      __logger.error("get(): exception", e);

      __retryCount += 1;

      if (__retryCount > 3) {
        if (__logger.isTraceEnabled()) {
          __logger.trace(String.format("get(): exit [%d ms]",
          ↪  System.currentTimeMillis() - __entryTime));
        }
        throw e;
      } else {
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e1) {
          __logger.error("get(): exception", e);
```

```
77              if (__logger.isTraceEnabled()) {
78                __logger.trace(String.format("get(): exit [%d ms]",
                  ↪  System.currentTimeMillis() - __entryTime));
79              }
80              throw e;
81            }
82          }
83        }
84      }
85    }
86  }
```