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 attach syntactic meta-data to various elements of Java source code. Since then, annotations have been extensively used by a number of libraries, frameworks and tools to conveniently extend behavior of Java programs that would otherwise have to be done manually or synthesized from an external resource. The annotations are usually processed through reflection and the extended behavior is injected into Java classes using aspect-oriented techniques or a direct byte code modification. However, in some cases, class-level instrumentation might not always be available or desirable and the transformation happens at the source code level.

In this case study we focus on such source-level transformation. Concretely, the task is to inject 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 are the contemporary transformation tools 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 them 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 and 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.

Cite??

In this case study we focus on the situation when class level instrumentation is not possible and develop a solution for annotation processing based on source level transformation. Concretely,

To appear in EPTCS.

the transformation task to inject behavior specified by an annotation library that covers common programming concerns such as logging, caching and retry on a failure. The objective is to explore how the 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[1]. 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` [2], a collection of Java annotations together with AOP-based processing tool which allows 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.

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

In the following text 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 sufficient to understand the nature of the transformation tasks to be carried out, shall there be any ambiguity, the readers are invited to raise an issue on the case study github page. In the last task, we look on the case of combining these annotations together. Finally, we propose an extension for an optimization the resulting Java source code when these annotations are used together. 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. Let us consider a class that is used to download a content of an URL. A possible[2] Java implementation is shown in Listing 1.

[1] `https://github.com/fikovnik/ttc15-tranj-case`

[2] In order to keep the case study concise and clear we omit the use of any external library.

```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 ready, we would like to extend it with the following features:

– *Failure handling.* The method should become more robust and accommodate for some of the inevitable network delays by retrying the download in the 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, while `SocketTimeoutException` should trigger a retry (preferably after a delay).

– *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 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 a transformation tool automatically synthesize code similar to the shown in Section B. Concretely, the only change to the original code should be the following annotations:

```java
@RetryOnFailure(attempts = 3, delay = 1000, retry = { SocketTimeoutException.class },
  ↪  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 to extend a given method with a simple failure handling strategy that retries failed invocations according to given options. Annotating the `URLDownload.get()` method with

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

should produce the following code:

```
1  public byte[] get() throws IOException {
2    // added retry counter
3    int __retryCount = 0;
4
5    // added loop
6    while (true) {
7      try {
8        // the content of the original method
9        ...
10     } catch (UnknownHostException e) {
11       // added escalation cases
12       throw e;
13     } catch (SocketTimeoutException e) {
14       // added retry cases
15       __retryCount += 1;
16
17       if (__retryCount > 3) {
18         throw e;
19       } else {
20         // added delay
21         try {
22           Thread.sleep(1000);
23         } catch (InterruptedException e1) {
24           throw e;
25         }
26       }
27     }
28   }
29 }
```

The original method code–*i.e.* what is between the lines 9–20 in Listing 1, is wrapped by another try-catch block (line 7) which is itself in a while loop (line 6). The number of attempts (3) is projected in the condition `if (__retryCount > 3)` (line 17) and the delay is translated in a current thread sleep on lines 21–25. Finally, we make a distinction in the exceptions that on which the call should be retried (`SocketTimeoutException`) and the ones that should be escalated (`UnknownHostException`) in the generated catch clauses in line 10 and 13 respectively.

2.3 Task 2: Caching

The second transformation tasks should handle the `@Cacheable` annotation. The objective is to extend a given method with a simple caching strategy that keeps a result of a method invocation for a given period of time. Annotating the `URLDownload.get()` method with `@Cacheable(lifetime = 1000)` should produce the following code[3]:

[3] Too keep things simple, we do not concern ourselves with invalidating the cache and thus freeing occupied memory.

```
1  public class URLDownload {
2    // added bookkeeping fields
3    private long __getCacheLastAccessed = 0;
4    private byte[] __getCacheContent = null;
5
6    ...
7
8    public byte[] get() throws IOException {
9      // added condition
10     if (System.currentTimeMillis() - __getCacheLastAccessed < 1000 && __getCacheContent
       ↪  != null) {
11       return __getCacheContent;
12     }
13
14     try (InputStream input = url.openStream()) {
15       ByteArrayOutputStream buffer = new ByteArrayOutputStream();
16       byte[] chunk = new byte[4 * 1024];
17       int n;
18
19       while ((n = input.read(chunk)) > 0) {
20         buffer.write(chunk, 0, n);
21       }
22
23       // added
24       __getCacheContent = buffer.toByteArray();
25       __getCacheLastAccessed = System.currentTimeMillis();
26
27       // added
28       return __getCacheContent;
29     }
30   }
31 }
```

The `__getCacheLastAccessed` and `__getCacheContent` defined on line 3 and 4 are cache bookkeeping fields used for storing the information about the last access time and method result respectively. The condition on line 10 checks the validity of the cache based on the annotation `lifetime`. Finally, the method exit point is replaced with the stored result on lines 24–25. Clearly, this should be done for all method exit points in the case the annotated method have multiple ones.

2.4 Task 3: Logging

The last annotation transformation introduce 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 that are caught.

Annotating the `URLDownload.get()` method with `@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 3 defines an instance of SLF4J logger [1]. Lines 9–11 and 25-27 insert the entry and exit point logging respectively. The entry point logging message additionally includes the state of the object–*i.e.* the value of all its declared (non-inherited) fields.

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` transformation, a log message should be added right after every catch clause. 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 (SocketTimeoutException 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 }
```

2.5 Task 4: Annotation composition

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

```
@RetryOnFailure(attempts = 3, delay = 1000, escalate = { UnknownHostException.class })
@Cacheable(lifetime = 1000)
@Loggable
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*)

An optional 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

The submitted solutions will be evaluated on a synthetic data set as well as on a real source code. The transformation is therefore expected to be packed in the way that it is runnable from a command line shell. The program should take two arguments, a path to a `valid` Java 7 source file input and a path to where the result transformation should be written.

```
$ ./my-solution.sh URLDownload.java SynthesizedURLDownload.java
```

A part of the evaluation process is done automatically and therefore the ability of scripting the transformation is important. It also demonstrates the usability of the solution. The evaluation test cases will be made available to the contestants after the submission period to reevaluate theirs and others solutions. For this case study we have chosen the following evaluation criteria that are evaluated automatically:

– *Correctness.* The transformation tasks are scored individually on the $0-1$ scale that corresponds to the percentage of successful transformations–*i.e.* the number of correctly inserted code versus the total number of expected insertion points. As a prerequisite, the resulting file must be a valid Java 7 file that compiles and whose original behavior is not altered.

– *Performance.* All submissions will be also assessed on their performance–*i.e.* how long did it take to transform the individual test cases.

Next to the automatic evaluation, we are also interested in an assessment of the usability of a given transformation tool to the problem being addressed. Usability of a programming language, a library or a tool is difficult to assess as it tends to be subjective since it largely depends on the

preferences and background of its users. In this case study we will a combination of source code complexity, abstraction level and development effort. For each task defined in the previous section, we ask the solution authors to provide a measures of

– The number of source lines of code (SLOC) of all manually written code excluding comments and empty lines.
– The time in person-hours spent in developing the solution including design, implementation, testing and debugging.

Interpreting SLOC metrics is always problematic and the issue of what is the right level of "verbosity" in a language is complex and should not be reduced naively to just counting SLOC. On the one hand, usability is not achieved only by having fewer lines of code, but instead, by having more expressive and concise code, which is beneficial to writers as well as to readers. On the other hand, code bloat resulting from code duplication and from lack of constructs that enable the building of more concise but expressive statements, is not desirable.

Next to this, we also ask about an assessment about the *abstraction level* that is provided by the transformation tool to solve this case study. It is a degree of the expressiveness of the solution ranging from high to low. The objective is to determine how closely the solution expresses the problem being addressed. A high abstraction level has the solution expresses in the terms of problem-level concepts while a low-level abstraction level is using simple implementation-level concepts.

References

[1] ASF: *Simple Logging Facade for Java*. Available at http://www.slf4j.org/.
[2] Yegor Bugayenko: *Useful AOP aspects, incl. JSR-303*. Available at http://aspects.jcabi.com/index.html.

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.SocketTimeoutException;
8  import java.net.URL;
9  import java.net.UnknownHostException;
10
11 import org.slf4j.Logger;
12 import org.slf4j.LoggerFactory;
13
14 public class ManualURLDownload {
15   private static final long CACHE_TIMEOUT = 1000;
16   private static final int MAX_RETRY = 3;
```

```
17
18    private final Logger logger = LoggerFactory.getLogger(ManualURLDownload.class);
19
20    private long lastAccessed = 0;
21    private byte[] cachedContent = null;
22    private final URL url;
23
24    public ManualURLDownload(String url) throws MalformedURLException {
25      this.url = new URL(url);
26    }
27
28    public byte[] get() throws IOException {
29      long entryTime = System.currentTimeMillis();
30
31      if (logger.isTraceEnabled()) {
32        logger.trace(String.format("get() [url='%s']: entry", url));
33      }
34
35      if (System.currentTimeMillis() - lastAccessed < CACHE_TIMEOUT && cachedContent !=
         ↪  null) {
36        if (logger.isTraceEnabled()) {
37          logger.trace(String.format("get(): exit cached [%d ms]",
             ↪  System.currentTimeMillis() - entryTime));
38        }
39        return cachedContent;
40      }
41
42      int retryCount = 0;
43
44      while (true) {
45        try (InputStream input = url.openStream()) {
46
47          ByteArrayOutputStream buffer = new ByteArrayOutputStream();
48          byte[] chunk = new byte[4*1024];
49          int n;
50
51          while ((n = input.read(chunk)) > 0 ) {
52            buffer.write(chunk, 0, n);
53          }
54
55          cachedContent = buffer.toByteArray();
56          lastAccessed = System.currentTimeMillis();
57
58          if (logger.isTraceEnabled()) {
59            logger.trace(String.format("get(): exit [%d ms]", System.currentTimeMillis() -
               ↪  entryTime));
60          }
61          return cachedContent;
62        } catch (UnknownHostException e) {
63            logger.error("get(): exception -> forced exit", e);
64          throw e;
65        } catch (SocketTimeoutException e) {
66          if (logger.isWarnEnabled()) {
67            logger.warn("get(): exception -> retrying", e);
68          }
69
70          retryCount += 1;
71
```

```
72          if (retryCount > MAX_RETRY) {
73            if (logger.isTraceEnabled()) {
74              logger.trace("get(): max retry count reached");
75            }
76            throw e;
77          } else {
78            try {
79              Thread.sleep(1000);
80            } catch (InterruptedException e1) {
81              if (logger.isWarnEnabled()) {
82                logger.warn("get(): interrupted while waiting to retry");
83              }
84              throw e;
85            }
86            if (logger.isTraceEnabled()) {
87              logger.trace("get(): retrying");
88            }
89          }
90        }
91      }
92    }
93  }
```

B Source Code of Synthesized `URLDownloader`

```
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.SocketTimeoutException;
8  import java.net.URL;
9  import java.net.UnknownHostException;
10
11 import org.slf4j.Logger;
12 import org.slf4j.LoggerFactory;
13
14 public class SynthesizedURLDownload {
15   private final Logger __logger = LoggerFactory.getLogger(SynthesizedURLDownload.class);
16   private long __getCacheLastAccessed = 0;
17   private byte[] __getCachedContent = null;
18
19   private final URL url;
20
21   public SynthesizedURLDownload(String url) throws MalformedURLException {
22     this.url = new URL(url);
23   }
24
25   public byte[] get() throws IOException {
26     long __entryTime = System.currentTimeMillis();
27     if (__logger.isTraceEnabled()) {
28       __logger.trace(String.format("get() [url='%s']: entry", url));
29     }
30
```

```
31      if (System.currentTimeMillis() - __getCacheLastAccessed < 1000 && __getCachedContent
         ↪  != null) {
32        if (__logger.isTraceEnabled()) {
33          __logger.trace(String.format("get(): exit [%d ms]", System.currentTimeMillis() -
             ↪  __entryTime));
34        }
35        return __getCachedContent;
36      }
37
38      int __retryCount = 0;
39      while (true) {
40        try (InputStream input = url.openStream()) {
41
42          ByteArrayOutputStream buffer = new ByteArrayOutputStream();
43          byte[] chunk = new byte[4*1024];
44          int n;
45
46          while ((n = input.read(chunk)) > 0 ) {
47            buffer.write(chunk, 0, n);
48          }
49
50          __getCachedContent = buffer.toByteArray();
51          __getCacheLastAccessed = System.currentTimeMillis();
52
53          if (__logger.isTraceEnabled()) {
54            __logger.trace(String.format("get(): exit [%d ms]", System.currentTimeMillis()
               ↪  - __entryTime));
55          }
56          return __getCachedContent;
57        } catch (UnknownHostException e) {
58          __logger.error("get(): exception", e);
59          if (__logger.isTraceEnabled()) {
60            __logger.trace(String.format("get(): exit [%d ms]", System.currentTimeMillis()
               ↪  - __entryTime));
61          }
62          throw e;
63        } catch (SocketTimeoutException e) {
64          __logger.error("get(): exception", e);
65
66          __retryCount += 1;
67
68          if (__retryCount > 3) {
69            if (__logger.isTraceEnabled()) {
70              __logger.trace(String.format("get(): exit [%d ms]",
                 ↪  System.currentTimeMillis() - __entryTime));
71            }
72            throw e;
73          } else {
74            try {
75              Thread.sleep(1000);
76            } catch (InterruptedException e1) {
77              __logger.error("get(): exception", e);
78              if (__logger.isTraceEnabled()) {
79                __logger.trace(String.format("get(): exit [%d ms]",
                   ↪  System.currentTimeMillis() - __entryTime));
80              }
81              throw e;
82            }
```

```
83                  }
84              }
85          }
86      }
87  }
```