Software-Architecture — Annotations, Reflection and Frameworks

Prof. Dr. Axel Böttcher

3. Oktober 2011
Objectives (Lernziele)

- Understand the Java feature „Annotation“
- Implement a simple annotation class
- Know what a framework is
- Implement a simple plugin-mechanism using reflection

Associated practical assignment: configurable toString()-mechanism
Annotations are metadata. They help to get rid of configuration files or naming conventions. Annotations are similar to interfaces:

- they are declared using @ interface instead of interface
- methods must not throw exceptions
- method declarations must not have parameters
- return types may only be of primitive or enum type, or String or Class (including arrays of these)
- return values can be defaulted using the keyword default
- Method „return“- values must be known when an annotation is used, they are referenced via key=value pairs: @Foo(name="bar")
- Method value() is treated specially in that it is not necessary to name the key value when an Annotation is used: @Foo("bar")
  
  binds bar to a key value

Like interfaces, annotations are compiled to class-files.
Example: JUnit’s @Test

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
    /**
     * Default empty exception
     */
    static class None extends Throwable {
        private static final long serialVersionUID = 1L;
        private None() {
        }
    }
    /**
     * Optionally specify <code>expected</code>, a Throwable, to cause a test method to throw an exception of the specified class is thrown by the method.
     */
    Class<?> extends Throwable > expected() default None.class;
    /**
     * Optionally specify <code>timeout</code> in milliseconds to cause a test method to take longer than that number of milliseconds.
     */
    long timeout() default 0L;
}
```
The Reflection Pattern

The equal treatment of program and data: also the program (classes, objects) is treated as data. So the program knows itself and can reflect about itself.

- Dynamic loading of classes at runtime
- Modification of classes at runtime (e.g. private → public)
- Instanciation of classes at runtime

Enables so called *Meta-Programming/Meta-Architectures*. 
Pros and Cons of Reflection

- Almost all modern Frameworks heavily rely on Reflection
- Allows Extremely flexible Software

See Bloch, „Effective Java“, p. 230:
- No more compile-time type checking
- The code required to perform reflective access is clumsy and verbose
- Performance suffers

Use reflection only if you absolutely have to!
Using Reflection, a Java Program is able to load and execute every arbitrary class at runtime.

Important Classes:

1. `java.lang.Class` internal representation of classes
2. `java.lang.reflect.Constructor` access Constructors
3. `java.lang.reflect.Method` access (and call!) methods
4. `java.lang.reflect.Field` access (and manipulate!) fields

When loading a class, the JVM instantiates an object of type `Class` or that class. The Method `public Class getClass()` is already defined within classe `Object`. 
Minimum example of a Unit-Tester

```java
don't edit: import java.lang.reflect.*;

don't edit: public class Tester {
    public static void main(String[] args) throws Exception {
        int tests = 0, successes = 0, failures = 0;
        Class<?> cut = Class.forName(args[0]);
        Method[] methods = cut.getMethods();
        for (Method method : methods) {
            if (method.getAnnotation(org.junit.Test.class) != null) {
                tests++;
                Object ott = cut.getConstructor().newInstance();
                try {
                    method.invoke(ott);
                    successes++;
                } catch (InvocationTargetException e) {
                    failures++;
                }
            }
        }
        System.out.println("Total tests: "+tests + " Successes: "+successes + " Failures: "+failures);
    }
}
```
Class<?> cut = Class.forName(args[0]);
Method[] methods = cut.getMethods();
method.getAnnotation(org.junit.Test.class)
Object ott = cut.getConstructor().newInstance();
Method Invocation

```java
method.invoke(ott);
```

Note the change of roles:
```java
object.method();
```
```java
↓
```
```java
method.invoke(object)
```

Of course, the JUnit-Software is much more complicated. Using Reflection is especially prone to any kind of Exceptions: 
ClassNotFoundException, IllegalArgumentException, 
SecurityException, IllegalAccessException, 
InvocationTargetException und InstantiationException.
Application: Test of private methods

You can use Reflection to manipulate the Modifier:

```java
Method m = testObject.getClass().getMethod("name");
m.setAccessible(true);
m.invoke(....);
```

Alternative (less complicated): Make all methods package-private (no Modifier). Test code in the same package may then access the method.
The Plugin Pattern

Goal: Ability to add new functionality to a system without having to recompile the whole system. The classes to be added are described in a Manifesto. Of course they must implement dedicated interfaces. Realization normally uses Reflection.
Framework

Framework defined interfaces

Framework classes

Plugin Factory

Plugin Classes

Manifest

instanciates

reads

use each other

Plugin
A Framework is a partially complete software (sub-)system that is intended to be instantiated. It defines the architecture for a family of (sub-)systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionalit should be made. In an object-oriented environment a framework consists of abstract and concrete classes. Methods of the own classes are called by the framework:

Das Hollywood-Prinzip: Don’t call us — we call you.