Code Museums as Functional Tests for Static Analyses

Daniel Speicher, Jan Nonnen, Andri Bremm
University of Bonn, Computer Science III, Bonn, Germany
{dsp, nonnen, bremm}@cs.uni-bonn.de

Growing your software guided by tests [2] has the benefit of thoroughly tested implementations of the right functionality. If you are developing static analyses your "test data" consist of *code* that has a few lines to a few classes. How can tests based on this "data" be kept expressive and maintainable? After exploring a variety of different other approaches we decided to embed the expectations into the data instead of embedding the data into test cases. Figure 1 presents an overview of our tool set. The functional tests are within Java code, the tested static analyses are implemented as logic programs.[1]

1 Code Museums

To keep test data and expectations collocated and to leverage the assistance of the compiler to check the syntactical correctness, we collect the test data in compilable projects and add the expectations as annotations. We call these projects "museums". As the purpose of this code is not to be executed there is a subtle change in the meaning of annotations (1.1). These annotations serve as descriptions of the code (1.2), as functional tests of the static analyses (1.3), and as the basis for education and cultivation of developers and static analyses (1.4).

1.1 Labels in Life and in Museums

Labels added to things used in life are mainly used to guide the handling of the thing.[2] Once we put a thing into a museum the purpose of the thing changes: It is meant to be viewed and not to be used. A label next to the thing therefore does not guide the usage but describes it. Similar to labels on things in life annotations on code can describe how the code is executed or at least compiled[3]. The annotations which we add to our code in a museum are not meant to change the compilation or execution but to describe the code.

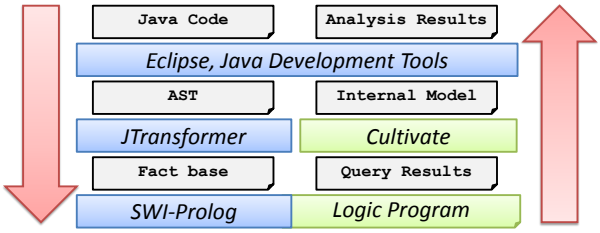

Figure 1: Data flow through the layers of our tools. Static analyses are implemented as logic programs in Cultivate.

1.2 Description of the Code

Figure 2 shows an example of a class in a code museum. Several annotations are added to the class and its methods. The annotation `@HasMetricValues` lists the values of three cohesion metrics[4] for this class. `@HasQualities` lists certain qualities of the methods. `@Expressed` lists concepts that the code expresses[5] In our example all method names start with "get" and may therefore be considered as "getter method". Still, a concept that is expressed in the code does not need to be fulfilled, therefore `@Fulfilled` lists the concepts that the method fulfills. The method `getCount` is clearly a getter method, while the method `getNextCount` is not.

1.3 Functional tests

The code together with the annotations is used as a functional test of the logic program. The logic program contains implementations of metrics and qualities. A test fails if the implementation does not produce the metric value or the quality that is annotated. Similarly the logic program contains a definition of when a program element is meant to express a certain concept and when it actually fulfills it. These definitions are tested by comparing them with the annotations of expressed and fulfilled concepts.

[1] http://sewiki.iai.uni-bonn.de/cultivate

[2] A label on a door may decide about whether women or man use it, or if we use the door to enter or to leave a building; a label on a bottle may decide whether we are willing to drink the content or will be even preventing any contact with our skin.

[3] E.g. `@Inject` identifying fields that are automatically initialized; `@SuppressWarning` suppressing compiler warnings.

[4] LCOM1 is the number of method pairs that do not access a common field. LCOM5 is the ratio of pairs of field and methods for which the method does not access the field. TCC is the ratio of pairs of public methods that do not (directly or via further methods) access a common field. [1]

[5] E.g. by following a naming convention, placing a type in a certain package, following a structure that is a known design pattern or implementation pattern.

1.4 Education and Cultivation

The annotations explain the code not only to the logic program but as well to developers or students. With annotations design pattern, implementation pattern or programming styles can be described.[6] Our example demonstrates an aspect of the calculation of cohesion metrics: Accesses to fields via trivial getter methods like `getCount()` should be considered equivalent to a direct field access[7]. Our example illustrates that basic concepts like "getter method" are not always as clear as say may seem. Judgment of design quality involves the understanding of similar concepts that appear to be fuzzy, once explored in detail. Examples are a good basis for such clarification. As functional tests for our logic program, a new failing example triggers the evolution of more refined judgment implemented in the logic program.

2 Experiences

While working with students we encourage a development style that is guided by tests. Indeed we did invest much time in the creation of different test frameworks. There had been successful steps like mastering plugin tests (start up eclipse, programmatically create Java projects, trigger compilation, execute the fact creation and the analyses), performance improvements by precompiling the fact bases, and finally a set of Hamcrest[8] matchers that allowed to write JUnit tests for the analyses fluently. Nevertheless because of the separation of "test data" and test expectations the creation of functional test cases never felt easy. The test cases per metric or smell barely covered all subtleties.[9] Once we introduced a first prototypical implementation of museums to our students the number of test cases created increased[10] Besides the pure number of test cases the most important benefit was the possibility to discuss the cases.

[6]There are annotations expressing that a program element plays a role in a design pattern or that there is a relation to another program element. References in the definition follow the rules of Java identifiers but ignoring modifiers that reduce visibility. Java does not allow to annotate statements (besides local variable declarations). Therefore we allow to add these annotations with a line comment next to the statement. The first token in a line comment is taken as identifier of the statement so that references to lines can be made. There are annotations that express that a program element has a "bad smell" (i.e. can be improved by refactoring) or that it does not.

[7]This explains the results of the cohesion metrics. `getCount()` is ignored. The two other methods access `logger` and are thus connected an LCOM1=0. Only `count` is not accessed by both remaining methods, so that LCOM5=1/4. Finally there is only one public method remaining, so that trivially TCC=1.

[8]http://code.google.com/p/hamcrest/

[9]Number of test cases between 1 and 5.

[10]Number of examples/counter examples: Overhaul of the implementation of the class version of the "Law of Demeter" 36/17; Android performance guidelines "avoid internal use of accessor methods" 7/7; "make private fields public if accesses from an inner class" 4/14; "declare constants static final" 2/14; "favor static methods of virtual methods" 30/32.

```
@HasMetricValues({ "lcom1=0", "lcom5=0.25", "tcc=1.0" })
public class CountGenerator {

    private int count = 0;
    private Logger logger;

    @Expressed({ "getter_method" })
    @Fulfilled({ "getter_method", "trivial_getter_method" })
    @HasQualities({ "returns_field", "side_effect_free" })
    public int getCount() {  return count;  }

    @Expressed({ "getter_method" })
    @Fulfilled({ "getter_method", "lazy_intializer" })
    @NotFulfilled({ "trivial_getter_method" })
    @HasQualities({ "returns_field" })
    protected Logger getLogger() {
        if (logger == null) { logger = new Logger(); }
        return logger;
    }

    @Expressed({ "getter_method" })
    @NotFulfilled({ "getter_method", "trivial_getter_method" })
    @HasQualities({ "returns_field" })
    public int getNextCount() {
        count += 10;
        getLogger().log("Next count is: " + count);
        return count;
    }
}
```

Figure 2: Code with annotations describing the code.

3 Limitations and Opportunities

As we integrate our expectations into the test data, these modifications may have an influence on the test result. Still, the distinction between annotated expectations and the main Java code are obvious to test developers, so possible interferences can be identified. The approach offers the opportunity to integrate the implementation smoothly into the design. To "test", whether a certain smell applies where it is expected, an analysis is implemented that verifies for each annotation of an expected smell that the implementation of the smell heuristic (another analysis) delivers the annotated element. Similarly consistency checks of the annotations, e.g. verifying the presence of the referenced analysis can be added.

4 Conclusion

Museums are collections of annotated code as "test cases" for static analyses that can be even used for educational purposes or as the basis to discuss judgments about code quality in the developer team. Developers may consult these collections of code examples to explore the coding culture and to refine their judgment. That is, well maintained museums offer developers the same service as museums offer to citizens.

References

[1] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[2] S. Freeman and N. Pryce. *Growing object-oriented software, guided by tests.* Addison-Wesley Professional, 2009.