Assignment 4 – Software Design and Modelling

Applying Design by Contract to an Existing Project

Filippo Casari Claudio Maggioni

Contents

1 Project Selection

The aim of this project is to apply Design by Contract principle to a part of or a complete application by not altering the application behaviour. No restrictions are placed on the size or type of project, other than being hosted on GitHub or otherwise on a public website.

We first consider the **vavr-io/vavr** GitHub project, a Java library containing some data structures useful for functional programming (e.g. "maybe" and "either" types). However, due to peculiar design choices in the library codebase allowing for concise client code, the interfaces provided have an insufficient number of query methods to effectively writed contracts, and therefore we discard the project.

We instead choose the *Apache Commons Lang* library (**apache/commons-lang** on GitHub), an Apache Software Foundation[1] sponsored support library for Java. We choose this project mainly for the support role it fulfills, as many classes implement data structures and collections that are independent from each other and implementing an easy to understand interface. Additionally, the library is written with classical Java software design principles allowing for a richer set of query methods to potentially use in contracts.

1.1 The Apache Commons Lang Project

According to GitHub, the project has 179 contributors as of October 30th, 2022.

[1] `https://apache.org`

Files	Language	Blank	Comment	Code
464	Java	16792	63093	89748
30	Text	1852	0	11972
26	XML	408	530	3958
1	Maven	29	39	967
4	Markdown	40	0	271
1	HTML	13	16	236
5	YAML	37	96	152
1	Velocity Template Language	23	31	90
1	Groovy	12	22	81
1	CSV	1	0	16
1	Properties	3	19	3
1	Bourne Shell	0	2	2
536	SUM	19210	63848	107496

Table 1: *cloc* output for the Apache Commons Lang project repository at revision `770e72d2f78361b14f3fe27caea41e5977d3c638`.

All the source and test classes are contained within in the package *org.apache.commons.lang3* or in a sub-package of that package. For the sake of brevity, this prefix is omitted from now on when mentioning packages and classes in the project.

We choose to analyze the code at the *git* revision `770e72d2f78361b14f3fe27caea41e5977d3c638` of the library.

According to GitHub, the project has more than 2,400 stars, more than 1,500 forks, 222 open issues on the Apache Commons JIRA instance[2]. We analyze some line of code metrics of the project by using the *cloc* tool, which returns what shown in Table 1.

1.2 Scope of Contract Implementation

As the Apache Commons Lang project ostensibly provides utility classes and collections, the unit of functionality is often a single class. We therefore decide to select 4 classes on which to design contracts. For sake of brevity from here onward the prefix *org.apache.commons.lang3* is omitted from each class name. The classes we choose are.

math.IEEE754rUtils a utility class able to efficiently sort floating point arrays;

math.Fraction a class representing a fraction with an integer numerator and denominator;

CharRange a memory efficient implementation of a set of `char`s within a contiguous range;

util.FluentBitSet a memory efficient implementation of ordered list of bits supporting bitwise operations and fast random access.

We choose these classes as we feel they are a significant sample of the API the Apache Commons Lang offers, as it is mainly composed by utility classes and implementations of data structures that client developers may use to support their client code.

As the Commons Lang project has a rather complicated build system targeting different Java versions and dividing sources and tests in different compilation modules, we decide to create a new project copying the sources of the aforementioned classes and the relevant tests. However, we have to make minor edits on the tests sources. This is because all test classes extend a generic setup class *AbstractLangTest*, which performs some global tear down operations that are irrelevant in the scope of this

[2] `https://issues.apache.org/jira/projects/LANG/`, as of 2022-12-06 (ISO 8601 date)

Files	Language	Blank	Comment	Code
8	Java	706	1541	3342
15	XML	0	0	1067
6	Text	6	0	149
1	Maven	4	0	54
1	Markdown	3	0	7
31	SUM	719	1541	4619

Table 2: *cloc* output for the *main* (not annotated) branch of the contracts project.

Files	Language	Blank	Comment	Code
13	Java	793	1667	3736
15	XML	0	0	1058
6	Text	6	0	149
1	Maven	4	0	65
1	Markdown	5	0	34
36	SUM	808	1667	5042

Table 3: *cloc* output for the *annotated* branch of the contracts project.

assignment. Therefore, in our copy we remove the `extends` clause from all the tests. Additionally, part of both the source and test code performs some validation operations which depend on other utility classes in Commons Lang. We therefore include Commons Lang as a dependency to our project to allow this code to be compiled.

The resulting project can then be found in the branch `main` of the repository

usi-si-teaching/msde/2022-2023/software-design-and-modeling/assignment-4-dbc/group-1

on *gitlab.com*.

We then run the *cloc* tool again on these sources to provide a baseline for further analysis of the annotated code. The line of code metrics can be found in Table 2.

2 Approach to Contract Design

Writing contracts for the classes we choose is quite easy, as documentation, tests and pre-existing exception-based argument checking paints a clear picture of how each public interface behaves. Therefore our approach is relatively straightforward.

To define a method contract, we first look at the aforementioned resources to understand the behaviour of class. If needed, we the precondition clause, considering both the existing sources in the method body and additional constraints on parameter values specified in the method documentation (though in our experience with these classes we can say the Commons Lang developers are quite thorough in their in-code assertions). We then define a postcondition clause when needed, taking count of the class's behaviour and the domain of return values.

To check if a contract is potentially defined in a wrong way we run the relevant test suite. With the exception of assertions relating to precondition violations (an issue discussed in Section 4), the unit tests are as effective with the original code as they are with the asserted code. This allows to check that our changes do not alter the intended behaviour of each class and additionally gives *jSicko* many opportunities to evaluate preconditions and postconditions with test data.

All the contracts we implement do not rely on any change on the original source code other than *jSicko* clause definitions and adding `@Pure` annotations to query methods.

3 Contract Implementation

In order to implement design by contract, we use version 1.0.0 of the **jSicko** contract checking library written by Dr. Andrea Mocci. The library allows definitions of preconditions and postconditions in `default` methods of interfaces which contracted class reference. We define a total of 18 preconditions and 38 postconditions. We define contracts for both command methods and query methods, using the latter ones to define a more detailed domain of the values returned and to describe inter-dependencies between queries. With the exception of *math.IEEE754rUtilsContracts*, the contracts we define are not complete as they do not span all the side effects caused by the command methods and all the inter-dependencies between the query methods.

The resulting code after contract implementation can then be found in the `annotated` branch of the repository

usi-si-teaching/msde/2022-2023/software-design-and-modeling/assignment-4-dbc/group-1

on *gitlab.com*. We now specify the methods our contracts apply to and in which interfaces the matching contracts can be found.

The interface *math.IEEE754rUtilsContracts* implements a complete set of contracts for the methods of class
math.IEEE754rUtils, namely:

- `static double min(double[])` and overloads;
- `static double max(double[])` and overloads.

The interface *CharRangeContracts* implements the contract of method `boolean contains(CharRange)` of class *CharRange*.

The interface *math.IEEE754rUtilsContracts* implements contracts for some methods of class *math.Fraction*, namely:

- `static Fraction getFraction(int, int)` and
  `static Fraction getFraction(int, int, int)`;
- `static Fraction getReducedFraction(int, int)`;
- `public int getDenominator()`;
- `public Fraction negate()`;
- `public Fraction abs()`;
- `public Fraction reduce()`.

Finally, the interface *util.FluentBitSet* implements contracts for some methods in the *util.FluentBitSet* class, namely:

- `FluentBitSet and(FluentBitSet)` and overloads;
- `FluentBitSet andNot(FluentBitSet)` and overloads;
- `FluentBitSet or(FluentBitSet)` and overloads;
- `FluentBitSet clear()` and overloads;
- `boolean get(int)` and overloads;
- `boolean isEmpty()`;
- `FluentBitSet flip(int)` and overloads;
- `FluentBitSet xor(FluentBitSet)`;

- `FluentBitSet set(int);`
- `int nextSetBit(int)` and `int previousSetBit(int);`
- `int nextClearBit(int)` and `int previousClearBit(int)` .

After implementing the aforementioned contracts we run the *cloc* tool again to understand the impact of contract code on source code metrics. Results are shown in Table 3.

The following sections go in further detail on the implementation of some contracts, providing additional insight on the strategies we use for defining them.

3.1 Bitwise Operations and the *FluentBitSet*

We worked on the class *util.FluentBitSet*, which is an alternative implementation of class *java.util.BitSet* providing some additional operations.

The original *BitSet* class implements a bit vector that expands as necessary. Bits are represented as `boolean` values, while non-negative integers serve as indexes for accessing them. The class allows to inspect, set, or clear certain indexed bits. Various bitwise logical operations can be applied to a *BitSet* to change its contents without changing the contents of another *BitSet*.

The *util.FluentBitSet* class varies from *BitSet* as it allows its command methods to return `this` to allow more succinct client code. This style of command methods is called "fluent", hence the name of the class.

3.1.1 The *xor* Command Method

```
@Ensures("xor_fbs")
public FluentBitSet xor(final BitSet set)
```

Listing 1: Postcondition declaration for the *xor* command method in the *util.FluentBitSet* class.

```
@Pure
default boolean xor_fbs(final FluentBitSet returns, final BitSet set) {
    int card1 = old(this).cardinality();
    int card2 = set.cardinality();
    final boolean bool;
    if (card1 + card2 > this.length()) {
        bool = (card1 + card2 - this.length()) >= this.cardinality();
        return bool && test_length(returns, set::length);
    } else {
        return (card1 + card2) >= this.cardinality();
    }
}
```

Listing 2: Matching postcondition clause for the *xor* command method in the *util.FluentBitSet* class.

In order to write clauses for this method we use the `int cardinality()` query method. The cardinality of a *FluentBitSet* is the total number of bits set to 1. We additionally use the `int length()` query method, returning the position of the most significant bit set to 1 in the set.

The `FluentBitSet xor(FluentBitSet)` is a command method in *FluentBitSet* performing an in-place bitwise exclusive-or operation with another set instance, and returning this instance.

In the contract (shown in Listing 2) we say that when the sum of the cardinalities of the old *FluentBitSet* state and the argument are greater than the returned length, the cardinality of the returned object must be less than the sum of the cardinalities of the old state and the argument minus the length of the new state.

For instance, let us consider:

`old(this).cardinality() = set.cardinality() = 8`, `this.cardinality() = 0`, and `this.length() = 8`

then $(8+8-8)$ is greater or equal to 0, where 0 is the cardinality of the new state. In contrary, if the sum is not greater than the length of the *FluentBitSet* we have:

`old(this).cardinality() = 0`, `set.cardinality() = 8`, `this.cardinality() = 8`, and `this.length() = 8`

then $(8+0-8)$ is greater or equal to 0, where 0 is the cardinality of the result of the *xor* operation.

3.1.2 The *isEmpty* Query Method

```
@Pure
@Ensures("is_empty")
public boolean isEmpty()
```

Listing 3: Postcondition declaration for the *isEmpty* query method in the *util.FluentBitSet* class.

```
@Pure
default boolean is_empty(final boolean returns) {
    return returns == (cardinality() == 0) && returns == (length() == 0);
}
```

Listing 4: Matching postcondition clause for the *isEmpty* query method in the *util.FluentBitSet* class.

We additionally define a postcondition clause for the `boolean isEmpty()` query method, returning `true` when all bits in the sets are 0. This condition trivially is true if and only if its *length* is also 0, and our contract indeed checks this fact.

3.2 Fraction Arithmetic and the *Fraction* Class

Our contracts for the *math.Fraction* class are defined in the interface *math.FractionContracts*. *Fraction* is a class that extends *Number* and implements an arithmetic fraction compose of an integer numerator and denominator. This class is immutable, inter-operates with most methods that accept a *Number* instance. We report below the methods on which we implemented contracts.

3.2.1 The *Negate* Query Method

```
@Pure
@Ensures("check_if_negative")
// [clause omitted]
public Fraction negate()
```

Listing 5: Postcondition declaration for the *negate* query method in *math.Fraction*.

```
@Pure
default boolean check_if_negative(final Fraction returns) {
    if ((this.getNumerator() > 0 && this.getDenominator() > 0) ||
        (this.getNumerator() < 0 && this.getDenominator() < 0)) {
        return returns.getNumerator() < 0;
    }
    return returns.getNumerator() > 0;
}
```

Listing 6: Matching postcondition clause for the *negate* query method in *math.Fraction*.

Here we want to make sure that the returned *Fraction* instance will be positive if the starting fraction was negative, and vice versa. Consequently, we adopt the clause shown in Listing 6 as a postcondition of the method negate (shown in Listing 5). This an instance of the aforementioned inter-dependencies between query method behaviours.

3.2.2 The *getDenominator* Query Method

```
@Ensures("non_zero_den")
public int getDenominator()
```

Listing 7: Postcondition declaration for the *getDenominator* query method in *math.Fraction*.

```
default boolean non_zero_den(final int returns) {
    return returns != 0;
}
```

Listing 8: Matching postcondition clause for the *getDenominator* query method in *math.Fraction*.

Here we introduce a simple postcondition checking a property that any fraction denominator has, namely that it is different from 0. This is an instance of a contract used to restrict the domain of the return value further than its type definition is able to capture.

4 Unit Tests and Code Contracts

The only issue we find in implementing contracts has to do with test checking operations that result in precondition violations. As some preconditions are defensively checked by the original Commons Lang code, some tests check the thrown exception type to be exactly the one thrown in the check. As *jSicko* checks preconditions before method entry, and *jSicko* throws a special exception (namely a child of *AssertionError*) when preconditions are violated, the assertions for the original source code fail as the thrown exception type is does not match the expected one.

To solve this error, we simply modify to all these assertions to allow for *AssertionError*s. For JUnit `assertThrows(...)` assertions we change the expected thrown type to be *Throwable*, as there is no more specific type that is both an *Error* and a *RuntimeException*. These alterations and *jSicko* contract code are the only changes we perform on the source code from Apache Commons Lang.

5 Conclusions

Apache Commons Lang is a Java project using well-established traditional design best practices. This is quite handy as its implementation can be easily understood and the project is well documented. The project also implements a variety of design patterns, making this contract implementation quite diverse as different classes are implemented in different ways beside domain-specific logic. For instance, the utility class *math.IEEE754rUtils* allows us to test *jSicko*'s ability to contract check static methods, and we can say it does that effectively from the experience we gathered.

The process used to write contracts in this assignment can be applied to other projects similar to Apache Commons Lang with respect to documentation availability and the set of Java programming language features use. However, we speculate the approach would need to be changed for projects with inadequate documentation or using newer or more advanced features of the Java language (like proxies), which may be able to break the internal abstractions used by *jSicko*.