CPS2002 — Code Analysis

Assignment Part 2

Juan Scerri

B.Sc. (Hons)(Melit.) Computing Science and Mathematics (Second Year)

December 30, 2022

Contents

List of Figures

Listings

1   Plagiarism Declaration

Plagiarism is defined as *"the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines"* (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I, the undersigned, declare that the report submitted is my work, except where acknowledged and referenced. I understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

Juan Scerri   CPS2002   December 30, 2022

Student's full name   **Study-unit code**   **Date of submission**

Title of submitted work: CPS2002 Code Analysis

Student's signature

2   Selected Open–Source Project

The selected open–source project which will be analysed in this report is the `modelmapper` project. It is a simple Java library which allows for the conversion of a class into another (see listing 1).

```java
public class PersonEntity {
    public Id id;
    public String name;
    public String surname;
    public int age;

    // getter and setters
}

public class Person {
    public Id id;
    public String name;
    public String surname;
    public int age;

    // getter and setters
}

Person person = (new ModelMapper()).map(personEntity, Person.class);
```

Listing 1: Using the `modelmapper` library

At the time of writing the project has 721 commits, 220 open issues and 11 open pull requests. The project was clone from GitHub and since the project uses Maven, the test suite was ran with the command `mvn clean test`, (see figure 1).

Furthermore, the test suite was run from within IntelliJ to get code coverage metrics (see figure 2).

Figure 1: Running the test suite from the terminal

Figure 2: Code coverage metrics generated by IntelliJ

Additionally, object-oriented data about the project was extracted using CodeMR.

Analysis of modelmapper
General Information

Total lines of code: 5352

Number of classes: 176

Number of packages: 10

Number of external packages: 30

Number of external classes: 182

Number of problematic classes: 12

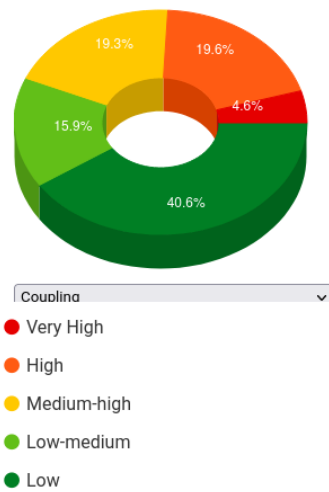**Number of highly problematic classes: 3**

Figure 3: HTML report generate by CodeMR for the `modelmapper` module

3   Project Analysis

3.1   Object–Oriented Metrics
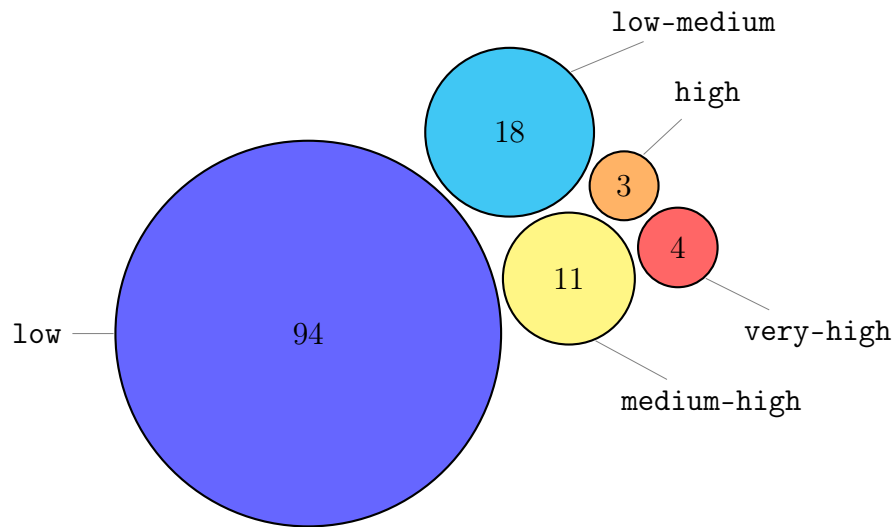
3.1.1   Complexity

Figure 4: Number of classes which have `low` to `very-high` complexity

Note: The percentage values provided by CodeMR where not used. This is because when converting from percentages to quantities the values where not identical to the ones reported by CodeMR.

CodeMR defines code complexity in the following way.

Definition. *A class is said to be <u>complex</u> if it is difficult to understand and describes the interactions between a number of entities. Higher levels of complexity increase the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes.*

CodeMR reports that only 7 classes have a `high` or `very-high` complexity. However, this does not mean that the project is not complex.

Apart from the complexity brought on by **branching** as mentioned in McCabe's Cyclomatic Complexity, there are other factors. Specifically, **indirection** caused by dependency injection or function calls also adds complexity as the programmer has to jump from one segment of code to another to understand.

Figure 5: A list of problematic classes identified by CodeMR

```java
@Override
public <P> TypeMap<S, D> include(TypeSafeSourceGetter<S, P> sourceGetter,
    Class<P> propertyType) {
  @SuppressWarnings("unchecked")
  TypeMapImpl<? super S, ? super D> childTypeMap = (TypeMapImpl<? super S
    , ? super D>)
      configuration.typeMapStore.get(propertyType, destinationType, name)
    ;
  Assert.notNull(childTypeMap, "Cannot find child TypeMap");

  List<Accessor> accessors = PropertyReferenceCollector.collect(this,
    sourceGetter);
  for (Mapping mapping : childTypeMap.getMappings()) {
    InternalMapping internalMapping = (InternalMapping) mapping;
    addMapping(internalMapping.createMergedCopy(accessors, Collections.<
    PropertyInfo>emptyList()));
  }
  return this;
}
```

Listing 2: An internal method in the class `TypeMapImpl` which was reported by CodeMR as having `very-high` complexity

Furthermore, additional barriers to understanding this particular project are: extensive use of **unchecked code**, **reflection** and **generics** (see listing 2). Unfortunately, Java has limited capabilities when it comes to reflection and generics requiring the use of unchecked code.

This naturally makes the project for any observer highly complex. However, after acquainting one's self with the terminology used and the less complex classes in the project it will overall be easier to understand the more complicated pieces of code. So, from that perspective overall the project has a `medium` complexity given that it is solving quite a complicated problem.
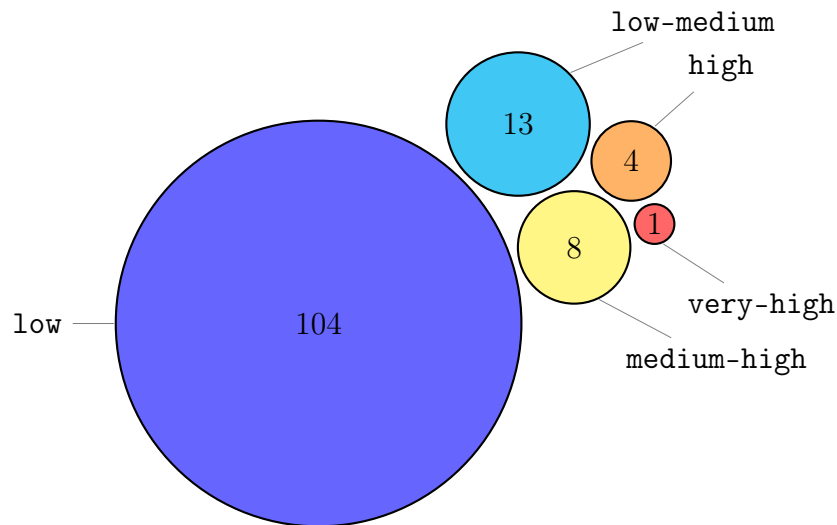
3.1.2 Coupling

Figure 6: Number of classes which have `low` to `very-high` coupling

Definition. *Two classes **A** and **B** are <u>coupled</u> if:*

- ***A** has an attribute that refers to (is of type) **B**.*

- ***A** calls on services of an object **B**.*

- ***A** has a method that references **B** (via return type or parameter).*

- ***A** has a local variable which type is class **B**.*

- ***A** is a subclass of (or implements) class **B**.*

Furthermore, tightly coupled systems tend to exhibit the following characteristics:

- *A change in a class usually forces a ripple effect of changes in other classes.*

- *Requires more effort and/or times due to the increased dependency.*

- *Might be harder to reuse a class because dependency classes must be included.*

3.2   Test Suite Suitability

3.3   Maintainability