

CPS2002 — Code Analysis

Assignment Part 2

Juan Scerri

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

December 30, 2022

Contents

1	Plagiarism Declaration	3
2	Selected Open–Source Project	4
3	Project Analysis	7
3.1	Object–Oriented Metrics	7
3.1.1	Complexity	7
3.1.2	Coupling	9
3.2	Test Suite Suitability	10
3.3	Maintainability	10

List of Figures

1	Running the test suite from the terminal	5
2	Code coverage metrics generated by IntelliJ	5
3	HTML report generate by CodeMR for the <code>modelmapper</code> module	6
4	Number of classes which have low to very-high complexity	7
5	A list of problematic classes identified by CodeMR	8
6	Number of classes which have low to very-high coupling	9

Listings

1	Using the <code>modelmapper</code> library	4
2	An internal method in the class <code>TypeMapImpl</code> which was reported by CodeMR as having very-high complexity	8

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

A handwritten signature in black ink, appearing to read 'J. Scerri', is written over a horizontal line.

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).

```
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).

```

[INFO] Reactor Summary for ModelMapper Parent 3.1.2-SNAPSHOT:
[INFO]
[INFO] ModelMapper Parent ..... SUCCESS [ 0.476 s]
[INFO] ModelMapper ..... SUCCESS [ 21.457 s]
[INFO] ModelMapper Extensions ..... SUCCESS [ 0.012 s]
[INFO] ModelMapper Spring Extension ..... SUCCESS [ 1.770 s]
[INFO] ModelMapper Guice Extension ..... SUCCESS [ 1.423 s]
[INFO] ModelMapper Dagger Extension ..... SUCCESS [ 1.161 s]
[INFO] ModelMapper Jackson Extension ..... SUCCESS [ 2.317 s]
[INFO] ModelMapper GSON Extension ..... SUCCESS [ 1.698 s]
[INFO] ModelMapper jOOQ Extension ..... SUCCESS [ 2.489 s]
[INFO] ModelMapper protobuf Extension ..... SUCCESS [ 2.188 s]
[INFO] ModelMapper Examples ..... SUCCESS [ 0.221 s]
[INFO] ModelMapper Benchmarks ..... SUCCESS [ 0.358 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 35.846 s
[INFO] Finished at: 2022-12-29T20:16:13+01:00
[INFO] -----

```

Figure 1: Running the test suite from the terminal

Element ^	Class, %	Method, %	Line, %
org	94% (199/210)	81% (807/995)	82% (3256/3952)
modelmapper	94% (199/210)	81% (807/995)	82% (3256/3952)
> builder	100% (0/0)	100% (0/0)	100% (0/0)
> config	100% (1/1)	100% (2/2)	100% (5/5)
> convention	100% (22/22)	80% (53/66)	92% (179/193)
> dagger	100% (2/2)	100% (3/3)	100% (5/5)
> gson	100% (2/2)	85% (6/7)	90% (28/31)
> guice	100% (2/2)	100% (3/3)	100% (5/5)
> internal	94% (105/111)	86% (596/690)	85% (2592/3017)
> jackson	85% (6/7)	73% (14/19)	66% (59/89)
> jooq	100% (2/2)	85% (6/7)	81% (18/22)
> protobuf	100% (32/32)	56% (34/60)	53% (150/281)
> spi	87% (7/8)	63% (19/30)	64% (36/56)
AbstractCondition	100% (1/1)	0% (0/2)	20% (1/5)
AbstractConverter	100% (1/1)	50% (1/2)	50% (2/4)
AbstractProvider	100% (1/1)	50% (1/2)	66% (2/3)
Condition	100% (0/0)	100% (0/0)	100% (0/0)
Conditions	57% (4/7)	27% (8/29)	25% (11/44)
ConfigurationException	100% (1/1)	100% (3/3)	100% (5/5)
Converter	100% (0/0)	100% (0/0)	100% (0/0)
Converters	100% (5/5)	100% (11/11)	88% (22/25)
ExpressionMap	100% (0/0)	100% (0/0)	100% (0/0)
MappingException	100% (1/1)	33% (1/3)	60% (3/5)
ModelMapper	100% (1/1)	80% (24/30)	81% (79/97)
Module	100% (0/0)	100% (0/0)	100% (0/0)
PropertyMap	100% (1/1)	93% (14/15)	92% (36/39)
Provider	100% (0/0)	100% (0/0)	100% (0/0)
TypeMap	100% (0/0)	100% (0/0)	100% (0/0)
TypeToken	100% (1/1)	62% (5/8)	81% (13/16)
ValidationException	100% (1/1)	100% (3/3)	100% (5/5)

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

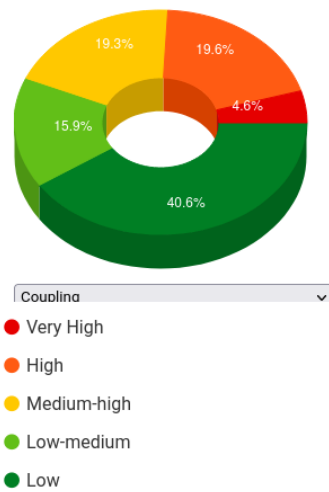


Figure 3: HTML report generate by CodeMR for the modelmapper module

3 Project Analysis

3.1 Object-Oriented Metrics

3.1.1 Complexity

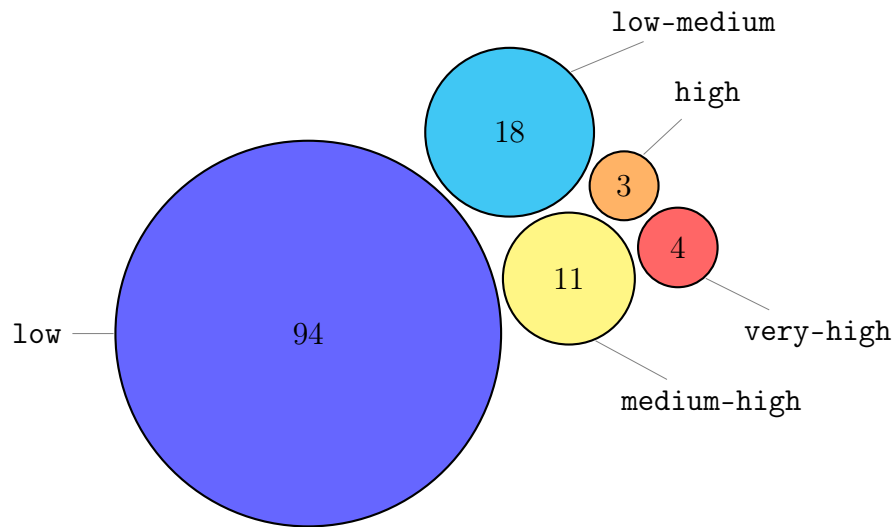


Figure 4: Number of classes which have low to very-high complexity

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

CodeMR defines code complexity in the following way.

Definition. A class is said to be *complex* 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.

Name	Complexity	Coupling	Lack of Co...
▼ modelmapper			
▼ org.modelmapper.internal	high	high	low
> MappingEngineImpl	very-high	high	high
> TypeMapImpl	very-high	high	high
> ExplicitMappingBuilder	high	very-high	high
> ImplicitMappingBuilder	high	high	medium-high
> TypeMapStore	high	low-medium	low-medium
> Errors	medium-high	medium-high	high
> InheritingConfiguration	medium-high	high	high
> MappingContextImpl	medium-high	medium-high	high
▼ org.modelmapper	low	medium-high	low
> ModelMapper	very-high	medium-high	medium-high
> PropertyMap	very-high	medium-high	medium-high
> TypeMap	low-medium	low-medium	high
▼ org.modelmapper.config	low	low	low
> Configuration	low-medium	low-medium	high

Figure 5: A list of problematic classes identified by CodeMR

```

@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

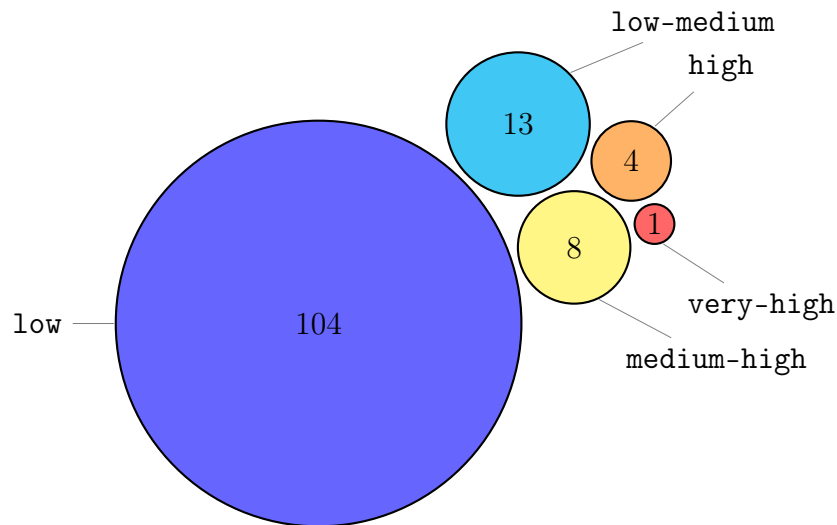


Figure 6: Number of classes which have low to very-high coupling

Definition. Two classes **A** and **B** are coupled 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