CPS2002 — Code Analysis

Assignment Part 2

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

December 31, 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
3.1.3 Cohesion 11
3.2 Test Suite Suitability 11
3.3 Maintainability 12

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 `modelmapper` 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
7 A dependency graph of all the classes in the `org.modelmapper.interal` package generated by CodeMR 10
8 Number of classes which have `low` to `high` **lack of cohesion** 11
9 Further code coverage metrics generated by IntelliJ 12

Listings

1 Using the `modelmapper` library 4
2 An internal method in the class `TypeMapImpl` 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 31, 2022
Student's full name	**Study-unit code**	**Date of submission**

Title of submitted work: CPS2002 Code Analysis

Student's signature

J. Scerri

2 Selected Open–Source Project

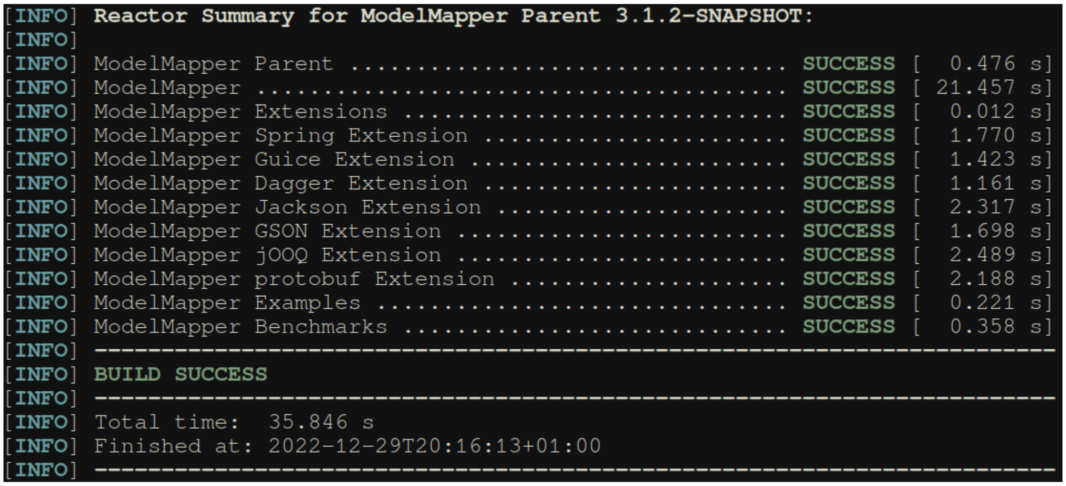
The selected open–source project which will be analysed in this report is the `modelmapper` project. It is a simple Java library which allows for the conversion of a class into another (see listing 1).

```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

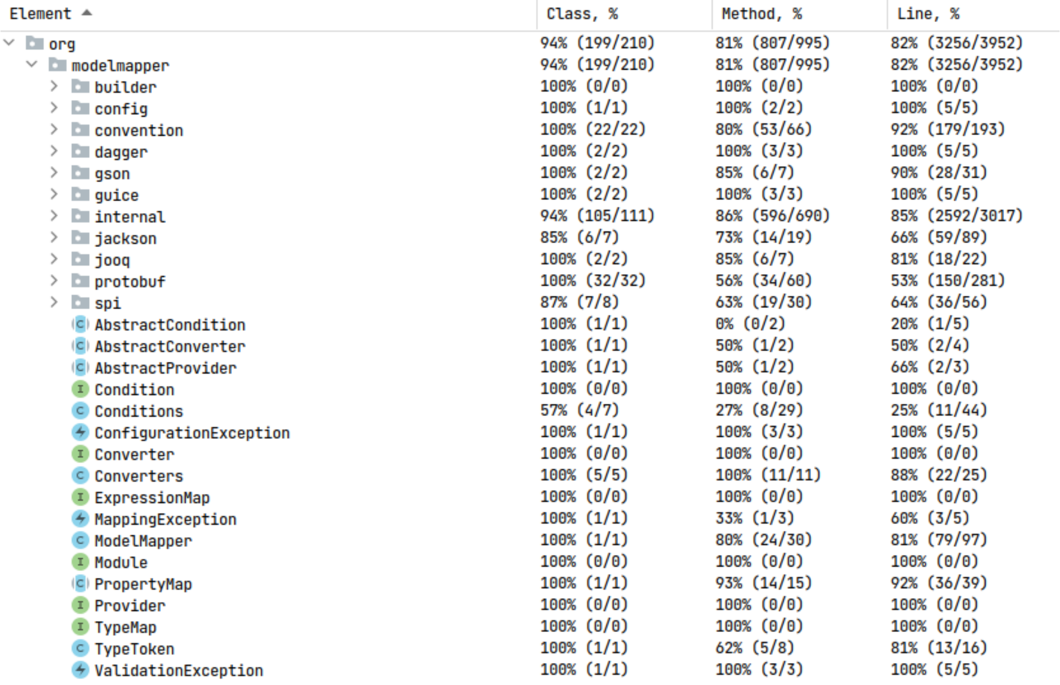
At the time of writing the project has 721 commits, 220 open issues and 11 open pull requests. The project was clone from GitHub and since the project uses Maven, the test suite was ran with the command `mvn clean test`, (see figure 1).

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



```
[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] ------------------------------------------------------------------------
```

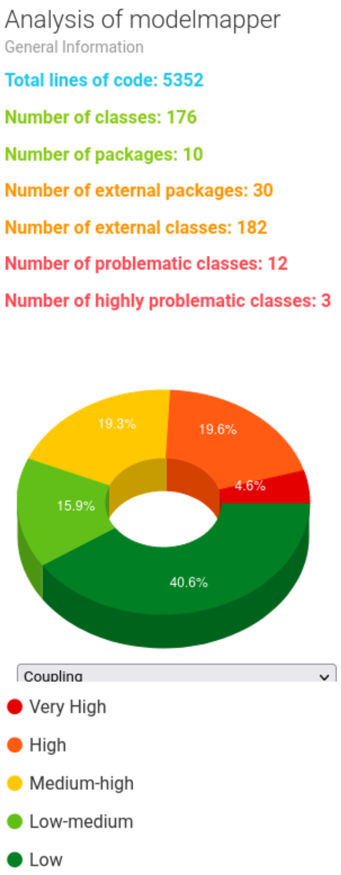
Figure 1: Running the test suite from the terminal



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

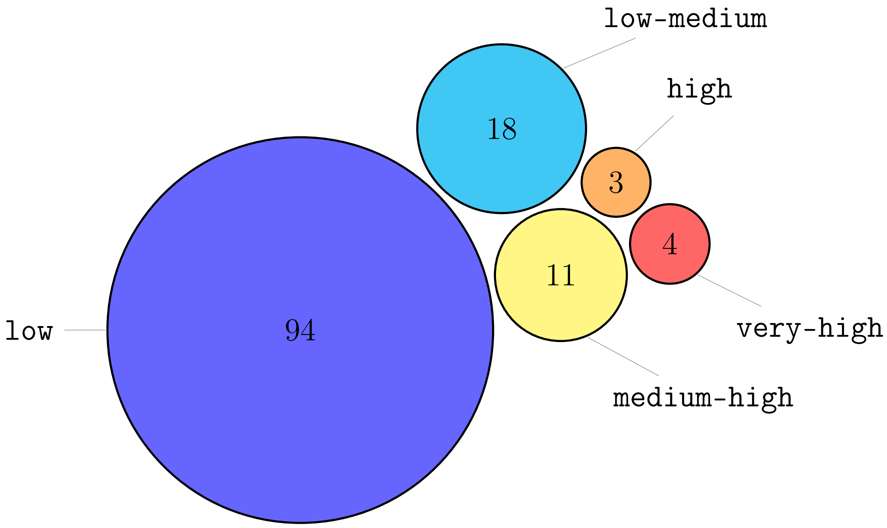


Figure 4: Number of classes which have `low` to `very-high` complexity

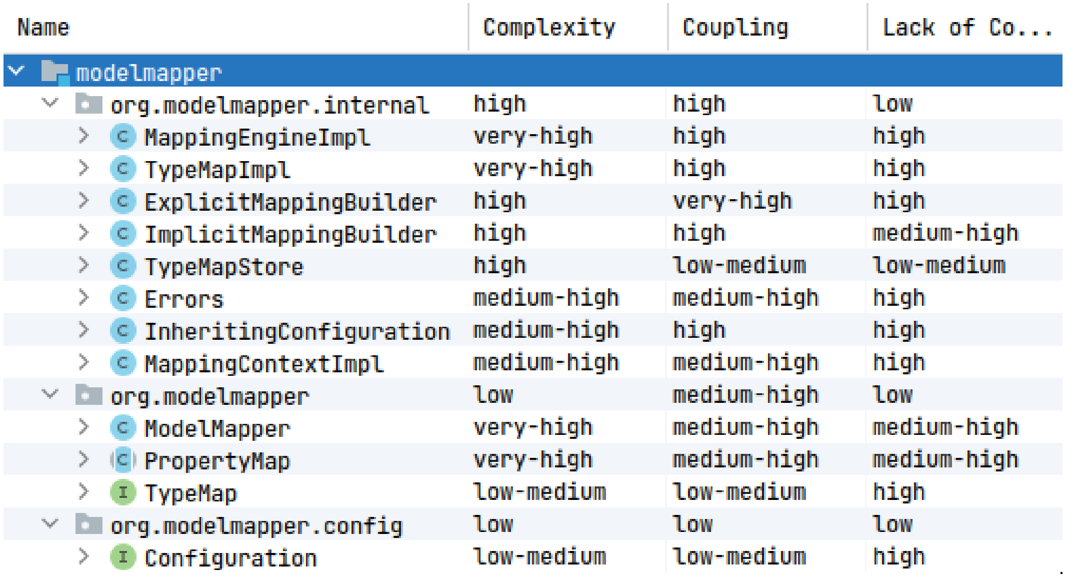
Note: The percentage values provided by CodeMR where not used. This is because when converting from percentages to quantities the values where not identical to the ones reported by CodeMR.

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

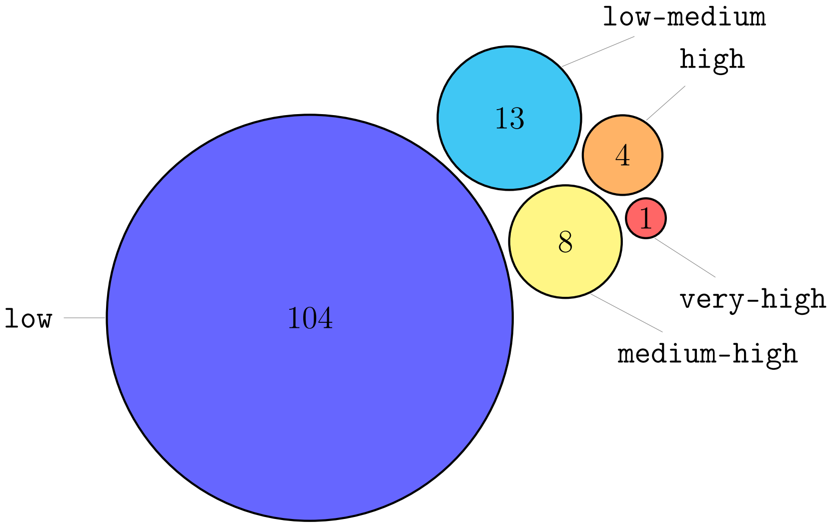


Figure 6: Number of classes which have `low` to `very-high` coupling

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 time due to the increased dependency.*
- *Might be harder to reuse a class because dependent classes must be included.*

As can be seen in figure 5, there are 5 problematic classes (also mentioned in figure 6) which have been marked as having high coupling.

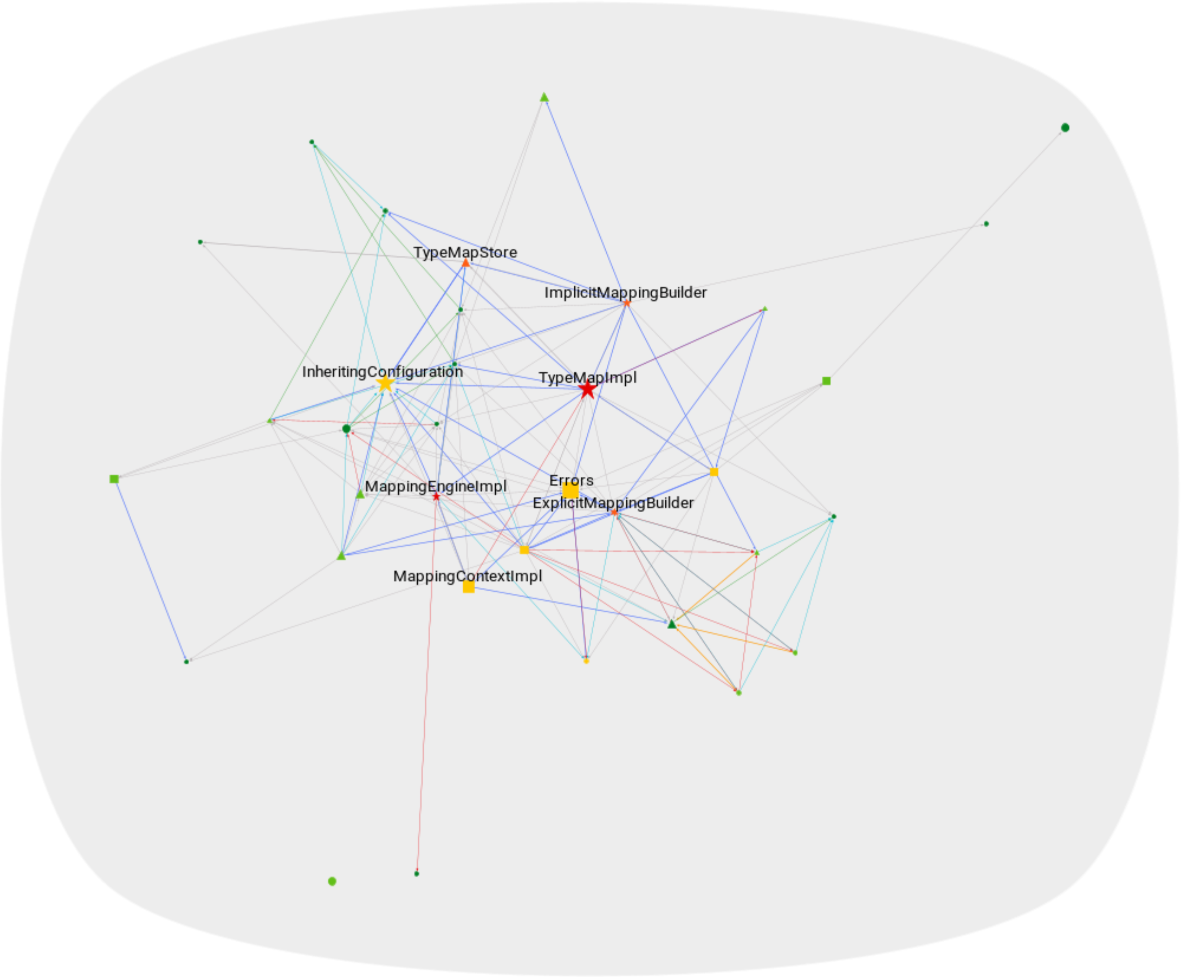


Figure 7: A dependency graph of all the classes in the `org.modelmapper.interal` package generated by CodeMR

Clearly, these classes seem to be doing most of the heavy lifting, in fact most of the complexity is also present in these classes. So as consequence high coupling is expected. This is clear in the dependency visualisation in figure 7.

As a suggestion, to reduce coupling (and even complexity) a more structured approach to dependency management should be taken. If necessary even having code duplication to allow for decoupling would be better as it isolates all the dependencies required by a specific class. Essentially, the dependency graph should look like a tree.

3.1.3 Cohesion

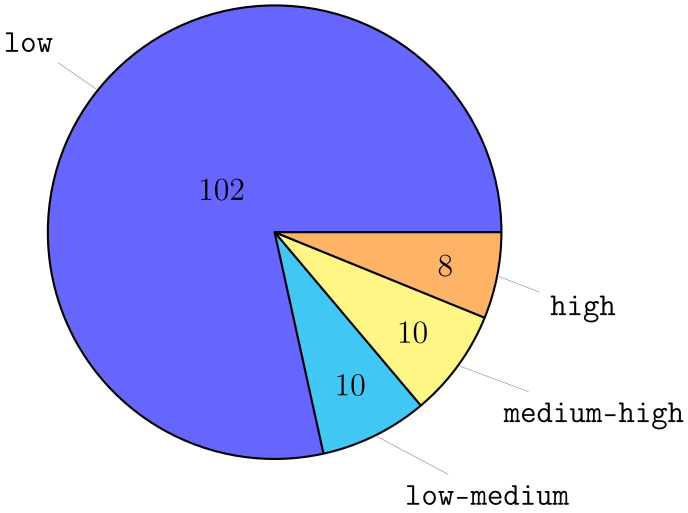


Figure 8: Number of classes which have `low` to `high` **lack of cohesion**

Definition. *Cohesion is a measure of how well the methods of a class are related to each other. High cohesion (low lack of cohesion) tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.*

As can be seen in figure 8 there no classes which have a very high lack of cohesion. Nevertheless, there some classes which have high lack of cohesion. Again looking at figure 5, they are essentially, the same classes described in the prior section.

But upon closer inspection of the actual classes reveals that these, where possible, are following a form of cohesions called Procedural Cohesion or they expose their methods to their consumers.

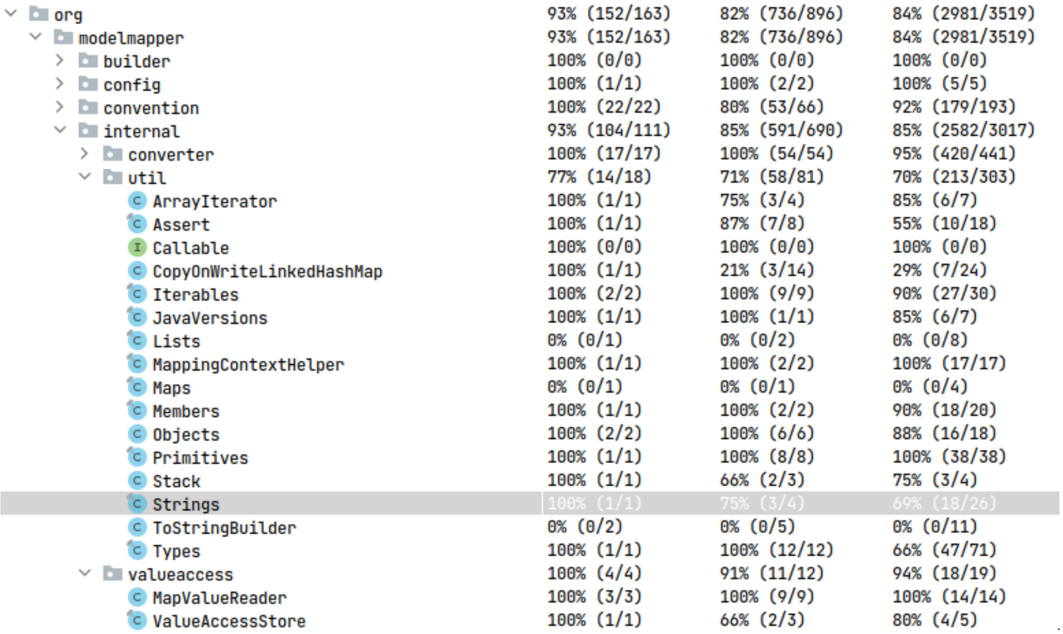
Procedural Cohesion is a type of cohesion where methods are grouped together because they form part of a chain of execution.

Cohesion can be improved by trying to break the code into methods which can be reused in multiple places in the class. However, of course this is not always possible.

3.2 Test Suite Suitability

The overall line coverage of the project was 82%. This is a significant amount of the code and hence the project is being sufficiently tested with regards to all possible code paths.

Nevertheless, this is **not** a guarantee of the project's quality. This is because certain tests can have artificially high code coverage whilst only really testing a small subset of the covered area. This is very common when unit tests call high–level methods to test the library/application.



org	93% (152/163)	82% (736/896)	84% (2981/3519)
modelmapper	93% (152/163)	82% (736/896)	84% (2981/3519)
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)
internal	93% (104/111)	85% (591/690)	85% (2582/3017)
converter	100% (17/17)	100% (54/54)	95% (420/441)
util	77% (14/18)	71% (58/81)	70% (213/303)
ArrayIterator	100% (1/1)	75% (3/4)	85% (6/7)
Assert	100% (1/1)	87% (7/8)	55% (10/18)
Callable	100% (0/0)	100% (0/0)	100% (0/0)
CopyOnWriteLinkedHashMap	100% (1/1)	21% (3/14)	29% (7/24)
Iterables	100% (2/2)	100% (9/9)	90% (27/30)
JavaVersions	100% (1/1)	100% (1/1)	85% (6/7)
Lists	0% (0/1)	0% (0/2)	0% (0/8)
MappingContextHelper	100% (1/1)	100% (2/2)	100% (17/17)
Maps	0% (0/1)	0% (0/1)	0% (0/4)
Members	100% (1/1)	100% (2/2)	90% (18/20)
Objects	100% (2/2)	100% (6/6)	88% (16/18)
Primitives	100% (1/1)	100% (8/8)	100% (38/38)
Stack	100% (1/1)	66% (2/3)	75% (3/4)
Strings	100% (1/1)	75% (3/4)	69% (18/26)
ToStringBuilder	0% (0/2)	0% (0/5)	0% (0/11)
Types	100% (1/1)	100% (12/12)	66% (47/71)
valueaccess	100% (4/4)	91% (11/12)	94% (18/19)
MapValueReader	100% (3/3)	100% (9/9)	100% (14/14)
ValueAccessStore	100% (1/1)	66% (2/3)	80% (4/5)

Figure 9: Further code coverage metrics generated by IntelliJ

So overall the project has enough unit tests. However, there are some gaps. Going over the code the following observations can be made:

- Getters are often ignore and not tested. Presumably, because they are very simple.
- Catch blocks also seem to be minimally tested.
- Some classes are not covered at all for example `BridgeClassLoaderFactory`, `StrongTypeConditionalConverter` etc.
- Some classes also seem to partially covered as a side effect of some other test for example `CopyOnWriteLinkedHashMap`.

Now if we take into consideration the data generated by CodeMR on where the greatest

3.3 Maintainability