CPS2002 — Code Analysis

Assignment Part 2

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

January 30, 2023

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	January 30, 2023
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 had 721 commits, 220 open issues and 11 open pull requests. The project was cloned from GitHub and since the project uses Maven, the test suite was run 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.

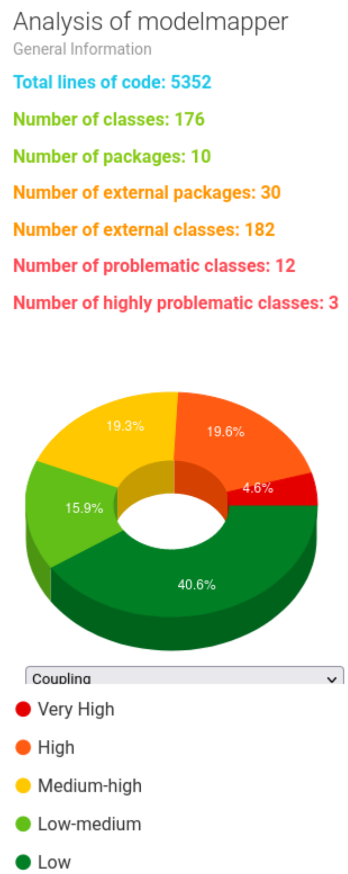

Figure 3: HTML report generate by CodeMR for the `modelmapper` module

3 Project Analysis

3.1 Object–Oriented Metrics

3.1.1 Complexity

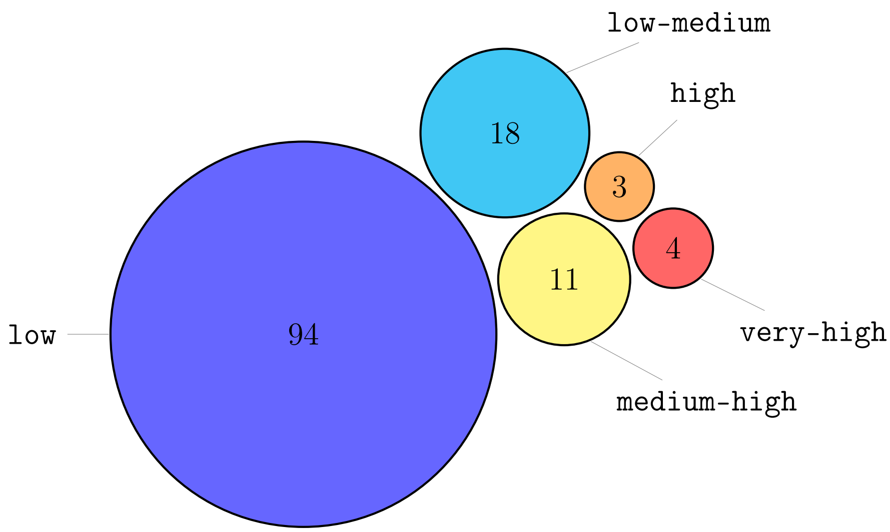

Figure 4: Number of classes which have `low` to `very-high` complexity

Note: The percentage values provided by CodeMR were 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 add complexity. This is because a 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: The 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. Additionally, looking at the User Manual on https://modelmapper.org will help in the understanding process.

Even though this system according to Lehman's Laws is very close to an `S`–type system. Given the lack of implementation–focused specification it makes the project more complicated then it has to be. This is because individual developer choices can heavily effect the complexity of the code.

So, from this perspective the project has a **medium** complexity given that it is solving quite a complicated problem. However, it might pose resitance to new contributors due to this complexity. This can be easily remedied with more friendly and implementation–focused specification.

3.1.2 Coupling

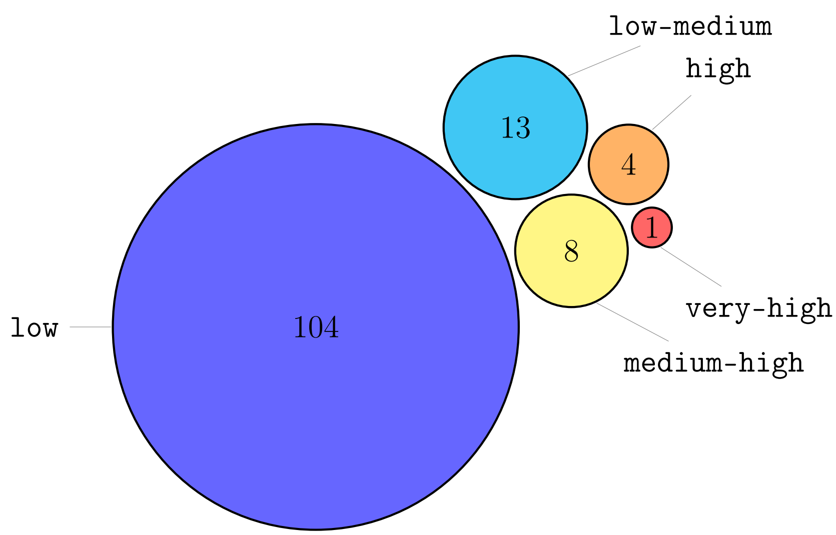

Figure 6: Number of classes which have `low` to `very-high` coupling

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.

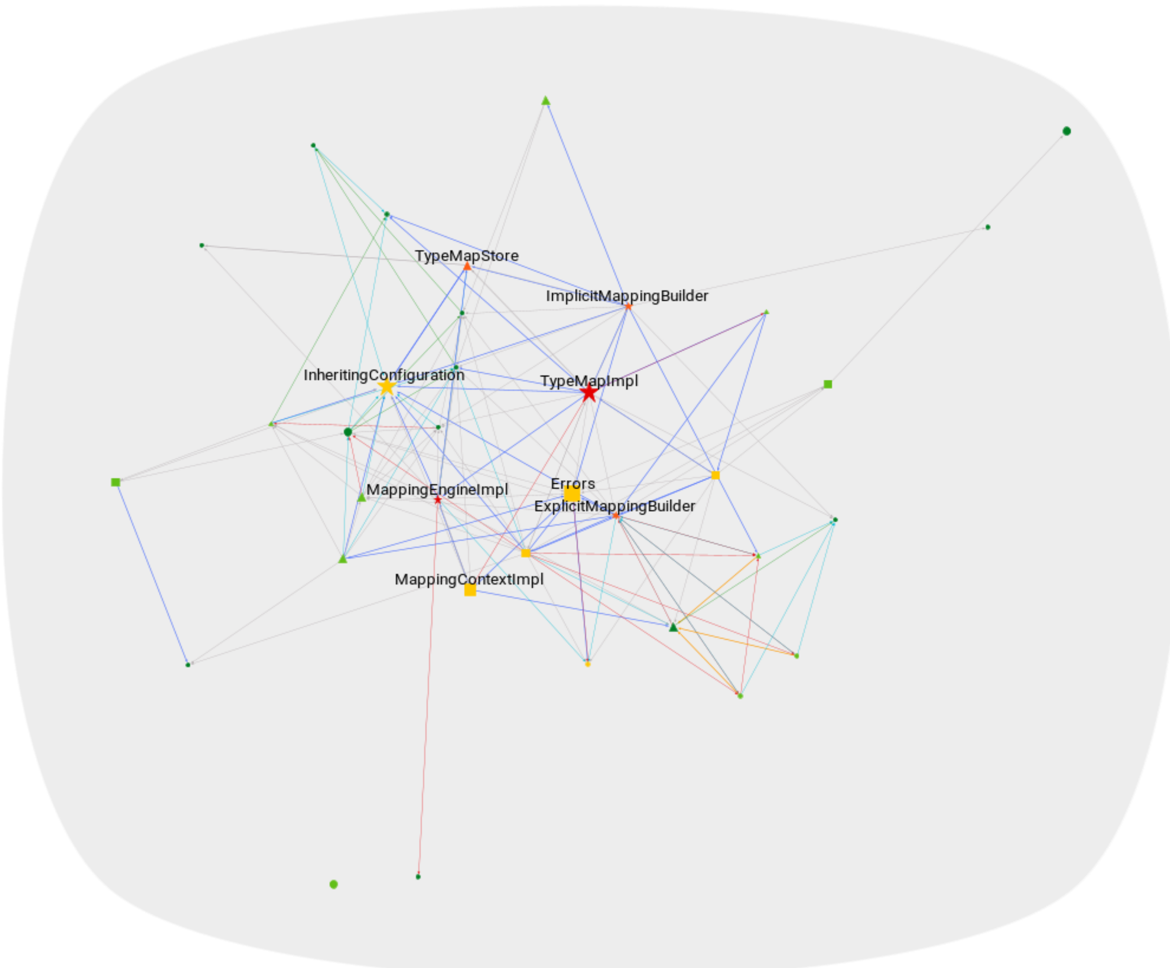

Figure 7: A dependency graph of all the classes in the `org.modelmapper.interal` package generated by CodeMR

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. Also, the type of coupling present between classes is <u>Common Coupling</u>. This is because the classes have instances of each other present as class fields and they share a lot of data and configuration.

Note: This coupling is not happening at a package level but at a class level. This makes it a less severe form of coupling. However, it still hinders understandability.

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. Ideally, the dependency graph should look like a tree.

3.1.3 Cohesion

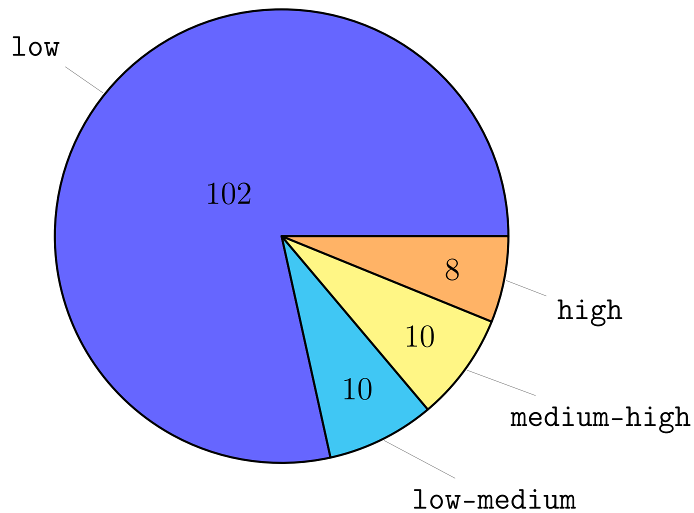

Figure 8: Number of classes which have `low` to `high` **lack of cohesion**

Definition. *<u>Cohesion</u> 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 are no classes which have a very high lack of cohesion. Nevertheless, there are some which have high lack of cohesion. Again looking at figure 5, they are essentially, the same classes described in the prior section.

Closer inspection of the actual classes reveals that these, where possible, are following a form of cohesion called <u>Procedural Cohesion</u> 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

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 be partially covered as a side effect of some other test for example `CopyOnWriteLinkedHashMap`.

Now, if we take into consideration the data generated by CodeMR (see figure 5), there is a large concentration of tests focused on testing the packages with the most complex classes (see figure 9). The package in question is `org.modelmapper.internal` and it is has a coverage of around 85%.

3.3 Maintainability

3.3.1 Development Cycle

2021 : Mar `2.4.0` ⟶ Apr `2.4.1` ⟶ Apr `2.4.2` ⟶ May `2.4.3` ⟶ May `2.4.4` ⟶ Dec `2.4.5`

2022 : Jan `3.0.0` ⟶ Mar `3.1.0` ⟶ Dec `3.1.1`

Figure 10: Project releases in the past two years (2021 and 2022)

The development cycle does not seem to follow any predefined schedule. In fact in the year 2021 there were twice as many releases as in the year 2022. Also, in previous years there was more signigificant activity. In general, releases used to contain a decent amount of work. In these last two years work on the project seems to have slowed down.

3.3.2 Commit Analysis

The following ten commits were analysed:

1. Add error message while skip conflict (`00a45001a73ee65f4351b48f97bb083cc2ae0d6e`)
2. Add unit tests for optional converters (`4b3cc868aacbc797ee2433c77938d62d6e62692d`)
3. Improve performance on implicit mapping (#619) (`2f5b7ccca0e56c7d529e6626b43746e583824910`)
4. Fix logical bug in PropertyInfoRegistry (`9ab5eaacfaf2cd3fb0c079346a6c205317122a49`)
5. Fix configuration ‘collectionsMergeEnabled’, ‘deepCopyEnabled’ (#485) (`b77316ba99ec52b1c497280a31da19ef0c29eedf`)
6. Change collections merge strategy breaks mapping of collection from a… (`6606f3d16b127c1577b7555402954d0aca33ceeb`)
7. Parse date from String to java.util.Date in yyyy-MM-dd format (`8a69be1a65f0abc05f19f2add81e95e36bed696c`)
8. In order to use ModelMapper in jdk11 (build 11+28). (`096183809417d0ff65b2f09c7995913cf877a54f`)

9. Fixes MapValueReader get wrong field type (#371) (`fbc99db5785bdc51b3a896dbf7442e83c7ac95c5`)
10. Updates asm, cglib and objenesis to their latest versions (`d66343ed9eb10d91220baa0a5a2a4be66e212ae3`)

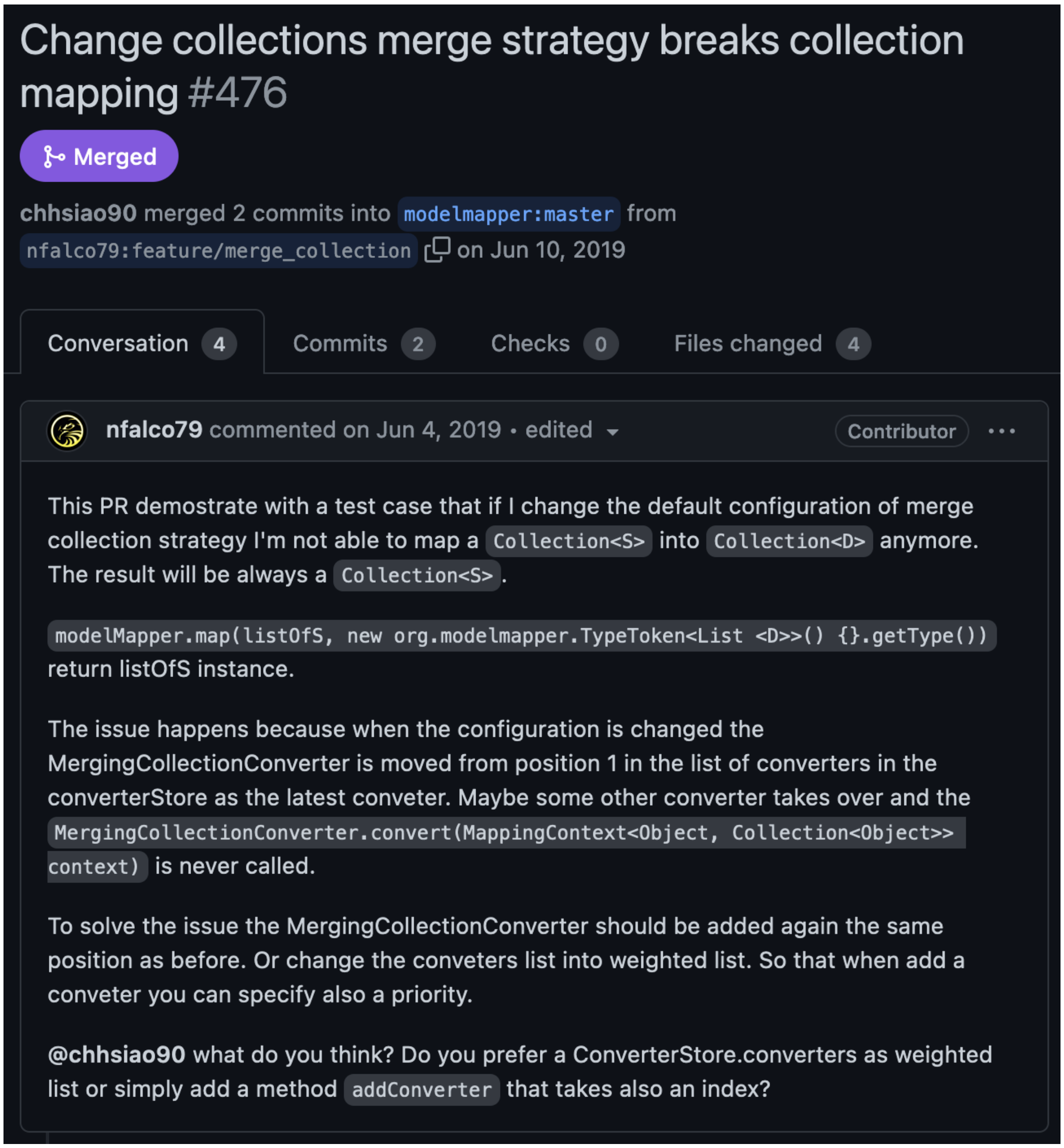

Figure 11: Pull request associated with the 6^{th} commit

In terms of readability, it depends which commit one considers. This is because some commits are inherintely very simple. For example the 10^{th} commit is a perfective commit; just bumping the versions of the described dependencies in the `pom.xml` file. Also, there are commits coming from pull requests which are corrective in nature that are also very well explained by the author (see figure 11).

However, there are commits from the core maintainers of the repository which are not very well explained. This is a symptom of having extensive knowledge of the project. To core maintainers many parts of the project might seem simple whereas, to new maintainers they might not be so simple. Of course some initial investment to acclimate to the project is still required.

Open–source projects suffer the same cost of onboarding new developer as For–profit companies, but to a greater extent since there is often no monetary motivater.

In previous yreas (2018–2020) the project was much more active. There was significant activity on GitHub and in each development cycle it was more common for the core maintainers to merge pull–requests submitted by contributors. Additionally, the core maintainers would also help contributors by reviewing code and providing suggestions to improve code (see the 10th commit).

Pull requests and issues were significantly used and were a good measure of the amount of work being done. Additionally, the core maintainers also adopted a system of calling certain test classes `GH{issue-number}`, where `GH` stands for GitHub to directly link the class to a GitHub issue or bug where most of the discussion and exposition for the bug would be stored.

Additionally, GitHub provides features for more explicit planning and development of certain features however, it is very likely that the core maintainers believe that the library itself is feature complete and does not need additional changes, except for maintenance and new features added by contributors.

3.3.3 Analysing Commit `00a45001a73ee65f4351b48f97bb083cc2ae0d6e`

```
+  Errors skipConflict(String skip, List<String> paths) {
+    return addMessage("Not able to skip %s, because there are already nested properties are mapped: [%s]. "
+        + "Do you skip the property after the implicit mappings mapped? "
+        + "We recommended you to create an empty type map, and followed by addMappings and implicitMappings calls",
+        skip, String.join(" ", paths));
+  }
+
```

Figure 12: Addition of a new error message in `Errors.java`

The above commit hash refers to the first commit in our list of analysis. Specifically, "Add error message while skip conflict". This commit falls under the category of perfective maintenance. This is because the core maintainer is adding an additional error message (see figure 12) meant to aid users and help users understand how `modelmapper` is supposed to work.

The core maintainer decided to add this commit after a number of users commented on GitHub issue 633 and GitHub issue 654 about some expected behvaiour. The expected behaviour by the users was that the nested fields of internal object where also skipped. However, this was not the case.

Additionally, the maintainer also added a test class called `GH654.java` in the `org.modelmapper.bugs` package to make sure that the behaviour is properly documented.

Note: At the time of writing both issue 633 and 654 were open.