

Applying Java Best Practices

Tamás Gálffy

Bachelor's thesis

May 2018

School of Technology, Communication and Transport

Information and Communications Technology

Software Engineering

Author(s) Gálffy, Tamás	Type of publication Bachelor's thesis	Date 30.05.2018 Language of publication: English
	Number of pages 42	Permission for web publication: x
Title of publication Applying Java Best Practices		
Degree programme Information Technology, Software Engineering		
Supervisor(s) Manninen, Pasi		
Assigned by EPAM Systems, Inc.		
Abstract <p>In larger projects, code must be easy to read, easy to understand and easy to maintain. Otherwise, this can become a major pain point on multiple levels; it can frustrate the individual, the team, and even the client. This provides a strong motivation to write clean code.</p> <p>„Clean code” as a concept can be very vague and hard to define. Even if one succeeds in summarizing the concept, it may still be unclear how to achieve clean code, and how to continually use it. The concept needs to be established in the context of the project at hand, based on industry standard practices. Although many of these practices provide clear instructions, some of them can be difficult and impractical to manually enforce.</p> <p>Since the project itself is confidential, an example project was introduced. This project contained many errors and multiple anti-patterns, that lent themselves very well to introducing and applying clean code practices.</p> <p>To implement these practices, tools were introduced that continuously inspect the code and provide reports on code quality. If certain requirements are not met, the application's build becomes unsuccessful, prompting the developer to fix the reported issues. Software engineering concepts were introduced and applied, to make the code more readable and maintainable.</p> <p>These practices span from low-level principles concerning class design and package structure, up to high-level concepts such as module structure, testing, and code metrics.</p> <p>At the end of the project, code metrics reports were created to objectively decide if these practices helped.</p> <p>The results were positive, fixing many code smells, introducing a consistent coding style, testing, and better API design.</p>		
Keywords/tags Clean code, java, best practices, unit testing, class design, maven		
Miscellaneous		

Contents

1	Introduction	4
1.1	Motivation	4
1.2	Example project.....	4
2	Project Setup.....	5
2.1	Project management tools	5
2.2	Apache Maven.....	6
2.3	Checkstyle.....	9
2.4	Surefire	12
2.5	Reporting.....	13
2.5.1	Project comprehension	13
2.5.2	Maven Javadoc plugin	13
2.5.3	JaCoCo.....	14
3	Refactoring.....	17
3.1	Dos and don'ts.....	17
3.2	Structure.....	18
3.2.1	Module structure.....	18
3.2.2	Package structure	20
3.2.3	Class structure	22
3.3	Testing	27
3.3.1	Unit testing	27
3.3.2	Integration testing.....	29
4	Code metrics	29
4.1	Benefits of metrics	29
4.2	SonarQube, SonarCloud	30

5 Results 30

6 Conclusion..... 32

7 References 34

Appendices 36

Figures

Figure 1. Example Javadoc output	14
Figure 2. Example of JaCoCo report	17
Figure 3. Members of the original Table class	24
Figure 4. GameState, based on Table	24
Figure 5. Original code report	31
Figure 6. New code report	32

Tables

Table 1. New module structure.....	19
------------------------------------	----

1 Introduction

1.1 Motivation

After one gets proficient with a few programming languages, it becomes more and more easier to write code. At some point, one might get even fluent with converting the desired logic into the appropriate expressions in the language of choice.

However, one must not forget, that the code being written is not only intended for the compiler or interpreter to understand. Code also has an audience. For example, other people working on the project could be the case when working at a company, or other collaborators if one collaborates to an open-source project. However, most importantly, the code's audience also includes the author's future self.

Products need to be delivered, maintained, fixed, improved. Since at least one of these mentioned situations apply to most of the code written nowadays, it is important to keep in mind the code's audience. Doing this will help improve the maintainability and extensibility of the code.

This brings the reader to the whole point of the thesis - while writing code might be easy, it is much less trivial to write readable, well-structured and future-proof code.

The objective of this thesis to introduce the reader to important software design concepts, architectural and design patterns and other methods to help the reader with his/her journey to writing better code.

1.2 Example project

To illustrate the points elaborated later in the thesis, an example project is introduced. This project was written in Java, early in the author's acquaintance with the language itself. The task at hand did nonetheless pose some interesting architectural questions. Thus, it is a great example of ant patterns and patterns executed wrong, and how to refactor, redesign or otherwise fix them, without breaking the original functionality, or straying off from the project's original goals.

The project itself is an open-source platform for experimenting with different artificial intelligence algorithms. This is achieved by implementing a selected game's rules

and providing a framework for implementing a wide range of problem-solving agents. The project's main goal is to provide developers with a base that they can build on, without getting too involved with the game's specifics or any kind of domain knowledge about the game - the focus is not on the game itself, but the algorithms built on the framework.

To underline that point, and evade applying pre-made artificial intelligence, the project opts for a custom-specified, turn-based, multiplayer game called nchess. As the name might suggest, this game is based on classic Chess; however, it differs from it in multiple ways. To quickly summarize:

- The game can be played by an arbitrary number of players (at least two)
- The playing table itself is an almost arbitrary graph of cells laid out on a plane, with possible holes
- The links between the cells are not bidirectional – accordingly, it is possible to move from cell A to cell B, but it is impossible to move back to cell A
- Game rules are mostly unchanged with only a few modifications were applied, so the game would make sense with more than two players
- Rules for movement are reinterpreted for the altered table, providing some interesting gameplay situations

The project's scope also includes a visual renderer for the game, to enable and encourage human interaction. To that point, the project also includes a visual client, for those who would like to test their algorithms against the human mind at first.

The project's source and related files can be found on GitHub:

<https://github.com/nchess>.

At the point of writing, the project comprises a single Git repository, however, in the future that is subject to change.

2 Project Setup

2.1 Project management tools

One can certainly be tempted to jot down the logic as fast as possible, without structure or planning, to focus only on the functionality and not on form. This could prove to be a viable approach, but at a certain complexity, this method runs the risk

of creating a Big ball of mud (Foote and Yoder 1999). This manifests itself in an accumulation of tech debt (Fowler 2003), where the more work gets done and the more features get developed, the amount of remaining work increases. It is similar to monetary debt in the sense that if it is not taken care of, it will increase, and in the case of software development, it becomes more difficult to implement any changes later on.

On the most fundamental level, this can be combatted by a software project management tool. By using such software, one can standardize the source code's layout, automate dependency acquiring and management, automate and simplify the build process, and improve project comprehension.

Softwares that provide a subset or all of the aforementioned features:

- **Apache Ant**, a build automation and dependency management tool
- **Apache Ivy**, a dependency management tool
- **Gradle**, a build automation tool

2.2 Apache Maven

The focus of this thesis lies on Apache Maven, referred to as Maven from this point onwards.

Maven is a software project management and project comprehension tool, with a few key objectives (What is Maven? 2018):

Simplify the build process

Maven abstracts away a significant fraction of the manual work in compiling a project from source code into Java .class files and packaging them to the archive of choice. This makes for faster development times, better developer focus and a more pleasant development experience.

Provide a unified build system

Having the same set of commands to manipulate a project, e.g. compile, package, and validate can reduce the project's learning curve significantly. A developer can get to delivering features much faster if it is not needed for them to read, analyze, learn and comprehend a multitude of build scripts and configurations. Once the developer

gets acquainted with Maven's basic concepts, they can understand almost any Maven-based project.

Provide information about the project

Maven provides many opportunities to gather information about certain aspects of a project, including reporting modules which output documentation, test reports and coverage, dependency list and convergence, and more. Some of these plugins will be covered in more detail later.

These goals are achieved through a few key concepts and processes that Maven defines:

Project Object Model

The Project Object Model, or POM, is an XML file that contains a description of the project (Introduction to the POM 2018). This description can include a variety of metadata, such as project name, description, authors and their contact information, the project license, and other various properties. The build process can also be customized here via plugins, which will be further discussed in the next point.

Oftentimes, most of the details need not be tailored to the project's needs – for example, in most of the projects, the generated output is stored in the directory called `target`. Having to specify this manually, for every single project, would impose an unnecessary burden on the developer. These default properties are described in the Super POM, Maven's default POM. Every POM can have a parent, which it extends. If not set, this is the Super POM. If set, stepping through the parent chain, one eventually ends up at the mentioned Super POM. Inheritance is an excellent tool to specify properties that are common to multiple Maven projects.

Aside from inheritance, POMs also support aggregation. In practice, this means that the POM contains modules, which may also contain modules, to an arbitrary depth.

The project object model also describes the project's packaging – it can output a Java archive (JAR, WAR, EAR and similar archive types) or its packaging can be POM, which means that the project itself does not produce an archive, but contains modules.

Other packaging types are possible as well, such as maven-plugin (Maven Coordinates 2018).

Dependency management

One of Maven's best-known features is dependency management. It can easily manage dependencies for projects from small codebases to major applications with complex module- and dependency structures. The process is fully automated here as well. The developer needs to enumerate the dependencies in the POM file, and Maven takes care of the rest, including transitive dependencies and version resolution.

Since libraries and other used components change often, even incorporating breaking modifications, each published version is stored separately. This enabled developers to stick to a single version, a range of versions, or even to just use the latest version.

In addition, dependencies are scoped. As an example, testing frameworks are only needed when performing tests. It does not make sense to bundle them with the application itself, since at point the tests have been performed, passed, and the testing framework no longer serves a purpose. In this case, the scope of the dependency is stated as a test.

Another scope to highlight is runtime. This scope specifies that the given dependency should not be available at compile time, but be included on the class path when performing tests and included in the output package. Runtime scoped dependencies can be used to great effect when developing libraries. The library would consist of three modules, an API module, an implementation module, and a bundle module. The bundle would include both the API and the implementation as a dependency, to tie them together. However, users of the library should not be able to access the internal classes of the library, so the bundle includes the implementation module as a runtime scoped dependency.

The above list contains only a few examples, Maven provides others scopes as well (Dependency Scope 2018).

Plugins

Maven can be thought of as a plugin execution framework. The work is done by the plugins, which Maven orchestrates. Plugins are used throughout the whole build process, for example, to clean the target directory, compile source files, run tests, create project documentation, and so on (Introduction to Maven Plugin Development 2018). While Maven provides a few built-in plugins, more often than not, additional plugins need to be involved. This can be easily done by adding the needed plugins to the POM.

2.3 Checkstyle

The preferred coding style differs mostly from programmer to programmer. This includes nuances such as:

Indentation

Whether to use tab characters, four spaces, eight spaces, or something else to mark the code block the statement belongs to. Bracket placement is also considered here, e.g. to move the starting bracket to a new line, leave it on the same line as the condition, etc.

Whitespace usage

The Java programming language's grammar is not concerned about whitespaces. This means that a developer is free to choose whether to use them and where to use them. Examples include between operands of binary operators, before and/or after language keywords, before commas in an argument list.

On multi-developer projects, this can make the codebase harder to read. If every programmer simply used their preferred coding style, it would mean that the code of every individual coder would be aligned differently, which would make it harder to cooperate and work with others developer's code.

Therefore, the need arises to somehow enforce a common, agreed-upon coding convention that each member of the team honors during the development. This could be controlled manually; however, that would take an unreasonable amount of resources and would lead to frustration.

Instead, a tool called Checkstyle can be used to solve the above problems. Checkstyle is a tool to help programmers write Java code that conforms to a coding standard of choice. The tool is automated, requiring no manual effort from the team.

Coding conventions can be described through Checkstyle's extensive configuration capabilities. A configuration file is an XML descriptor, containing the rules and their details about the coding conventions to be adhered to.

Amongst others, Checkstyle can be configured to scan for e.g. indentation and formatting errors, line length, Javadoc presence and thoroughness, unused or illegal imports, redundant modifiers, and more. In certain situations, Checkstyle can spot potential bugs as well.

Checkstyle offers two predefined rulesets:

- `sun_checks.xml` – Based on [Code Conventions for the Java TM Programming Language](#) from Sun Microsystems, version of April 20, 1999
- `google_checks.xml` – Based on [Google Java Style](#), version of 28 February 2017

These rulesets provide a good starting point. However, it is of utmost importance to choose a ruleset that makes sense to the team and applies best to the project.

Otherwise, developers will experience significant friction when working with the codebase. An example configuration, based on Sun's conventions, is provided in Appendix 1.

In Maven, Checkstyle can be configured as a reporting plugin. To do this, the project's `pom.xml` needs to have a `<reporting>` section with Checkstyle added:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>${checkstyle.version}</version>
      <reportSets>
        <reportSet>
          <reports>
            <report>checkstyle</report>
          </reports>
        </reportSet>
      </reportSets>

      <configuration>
        <configLocation>checkstyle.xml</configLocation>
      </configuration>
    </plugin>
  </plugins>
</reporting>
```

With this snippet present, Maven will run Checkstyle as part of reporting; for example when issuing the command `mvn site`. In this example, the XML descriptor's location relative to the Maven project has also been provided.

Unfortunately, this configuration only reports errors, it does not enforce them. In order to do that as well, Checkstyle also needs to be added as a build plugin.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>${checkstyle.version}</version>
      <executions>
        <execution>
          <id>validate</id>
          <phase>validate</phase>
        </execution>
      </executions>
      <configuration><configLocation>checkstyle.xml</configLocation>
        <encoding>UTF-8</encoding>
        <consoleOutput>>true</consoleOutput>
        <failsOnError>>true</failsOnError>
        <linkXRef>>false</linkXRef>
      </configuration>
      <goals>
        <goal>check</goal>
      </goals>
    </plugin>
  </plugins>
</build>
```

The above XML part will instruct Maven to run the Checkstyle plugin as part of the build process. This is defined under the executions node, with phase. This binds the plugin to Maven's validate phase, which is run before compiling the source code. This ensures that code not conforming to the configured coding standard will not compile, in other words, breaks the build. This is a beneficial effect, since broken builds are very apparent, and the errors are reported in a very early stage.

Keeping Checkstyle's benefits in mind, a possible benefit can be developer frustration. This is a factor apparent when new developers join the project and need to adapt to the enforced ruleset.

2.4 Surefire

After a certain complexity, it becomes very difficult to write applications without errors. A small oversight or mistype from the programmer's part can lead to subtle errors that are hard to track down, even though easy to fix. On other occasions, through the continuous development of the codebase, a multitude of seemingly harmless changes can alter the behavior, leading to unwanted results. Major refactoring efforts can also harm the stability of the code.

To combat this, it is standard practice to write different kinds of tests. These tests act as safety nets and as documentation of the expected behavior. A codebase with proper test coverage can ensure that errors such as described above happen significantly less frequently. This topic is further discussed in Chapter 3.3 Testing.

It is desirable that these tests are to be run automatically, are reported thoroughly, and depending on the error, are signaled early.

The Maven Surefire plugin is a component that runs these tests as part of the build process. Its setup is straightforward and helps with most of the mentioned needs. In order to be used, the plugin simply needs to be added as a build plugin, as can be seen in the snippet:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.21.0</version>
</plugin>
```

This snippet will make sure the plugin is run as part of the build process, in the **test** phase, after the source files have been successfully compiled. If any of the tests fail, the build will be unsuccessful.

Thus, errors can be found early, during the build process. The tests are also run automatically, without developer effort. Surefire also provides reports on the tests results, so the reason for the unsuccessful build is also easy to find.

2.5 Reporting

2.5.1 Project comprehension

As mentioned earlier, Maven is also a project comprehension tool. One facet of this attribute is Maven's reporting capability. Depending on configuration, Maven can provide reports on multiple aspects of the project, including dependencies, possible conflicts in them, the project's description, module structure, test results and coverage, API documentation, and more. The reporting capabilities of Maven can be called with the `mvn site` command. The following sub-chapters give an introduction to two plugins that are useful in this regard.

2.5.2 Maven Javadoc plugin

Java's de facto industry standard approach for API documentation is including the documentation in the code itself, as comments. These comments must conform to an established comment format in order to be treated as part of the documentation, instead of plain comments. These comments provide a description about the language element at hand (class, method, constant, etc.), input, output and template parameters if applicable, optionally can highlight related elements, cross-references through hyperlinks, and other details.

An example of the Javadoc comment format, detailing a class' purpose and behavior is illustrated below:

```
/**
 * <p>Class to represent the King piece.
 * <p>
 * The king can take a single step, either diagonally or straight.
 In the implementation's terms, this means that
 * the king can either step on a neighboring node, or a secondary
 neighbor node.
 */
public class King extends Piece {
```

To extract these comments, and generate an interlinked set of documents, the aptly named Javadoc tool must be used. This tool scans the input code for documentation comments and generates a collection of HTML files that can be conveniently navigated via hyperlinks. Documentation generated by Javadoc can also be intuitively

browsed, as the output includes a glossary of all the packages and classes. (See Figure 1.)

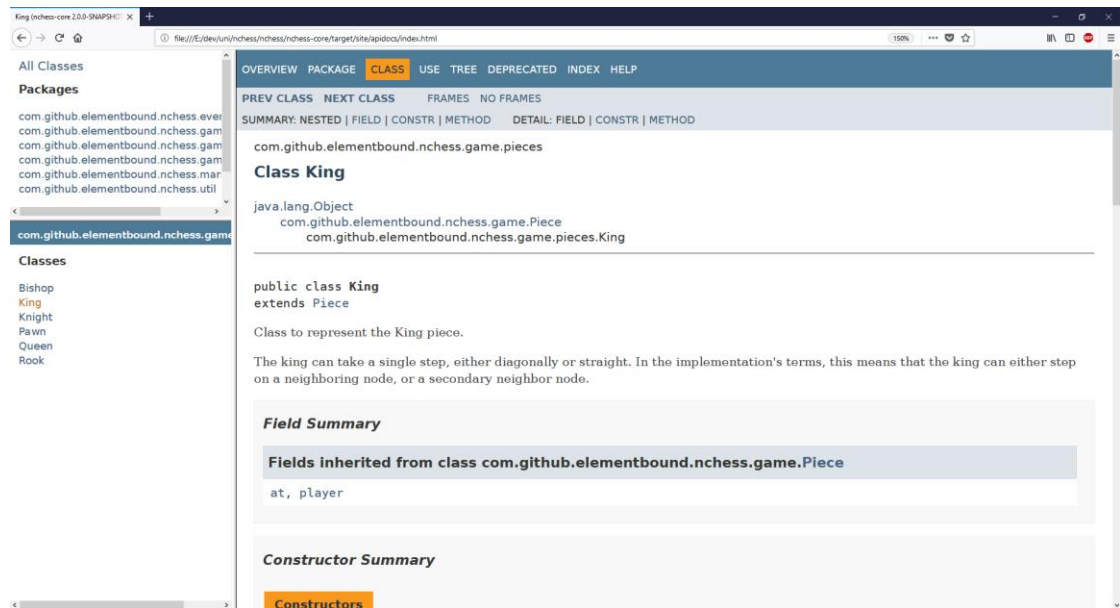


Figure 1. Example Javadoc output

The Maven Javadoc plugin integrates the Javadoc tool to be run on the project. As with Checkstyle, the plugin needs to be added to the project as a reporting plugin. However, contrary to Checkstyle, its configuration is very simplistic:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-javadoc-plugin</artifactId>
  <version>3.0.0</version>
</plugin>
```

This configuration provides no further details to the plugin, so it will proceed with default values. With this snippet in place, the Javadoc section will appear on the generated site under the Project reports menu.

2.5.3 JaCoCo

Introducing tests to a codebase is definitely a step in the good direction. However, simply writing tests may not guarantee that there are enough tests to properly cover the codebase. Even if the quantity of the tests might intuitively right, one cannot

easily decide if the required functionality is well covered. To summarize the above with a few simple questions:

- How much test code do we have?
- How much is enough?
- Do we have enough?

The Java Code Coverage (JaCoCo for short) is a free code coverage library and corresponding Maven plugin, based on the experiences of using similar libraries for many years. JaCoCo can help with the above questions in the following way:

How much test code do we have?

JaCoCo generates reports from all the test cases run, thus providing an easily quantifiable way to answer this question. In addition, the plugin also provides metrics about said tests, like code coverage (how many of the test class' source code lines were tested) and branch coverage (how many possible code paths were covered).

How much is enough?

This question is left to the judgment of the developer, as a number of factors are at play in this case. Code coverage alone is not the ultimate metric, however, it is a helpful tool.

JaCoCo's role can be seen more clearly in the next question.

Do we have enough?

Once a decision has been made as to how much is enough, JaCoCo can help in enforcing this rule. If the test coverage is too low, the build will fail.

JaCoCo can be set up both as a reporting plugin and as a build plugin. The former provides only reports on test coverage, however, it does not enforce any kind of coverage limit, and is not able to break the build. However, it is needed to include the reports in those generated by Maven. To do so, this plain snippet must be added:

```

<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.1</version>
</plugin>

```

This will ensure that Maven will include JaCoCo's results in the generated site.

The remaining requirement is that JaCoCo must be able to fail the build, if the test coverage is under the configured threshold. This is implemented by including JaCoCo as a build plugin as well, as the below example illustrates:

```

<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.1</version>
  <executions>
    <execution>
      <id>default-prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

By listing the plugin's executions, JaCoCo has been included in the build process. To make JaCoCo fail the build if the coverage ratio is not met, the plugin needs to be further configured, by adding this snippet to the plugin body:

```

<configuration>
  <rules>
    <rule
      implementation="org.jacoco.maven.RuleConfiguration">
      <element>BUNDLE</element>
      <limits>
        <limit
          implementation="org.jacoco.report.check.Limit">
          <counter>INSTRUCTION</counter>
          <value>COVEREDRATIO</value>
          <minimum>0.80</minimum>
        </limit>
      </limits>
    </rule>
  </rules>
</configuration>

```

With this configuration in place, the build will be unsuccessful if the code coverage ratio is below 80%.

Having successfully configured the plugin, it will produce a report, containing a table with the statistics as seen. (See Figure 2.)

JaCoCo























Element	Missed Instructions	Cov.	Missed Branches	Cov.
 org.jacoco.examples		58%		64%
 org.jacoco.agent.rt		84%		88%
 jacoco-maven-plugin		90%		80%
 org.jacoco.core		98%		95%
 org.jacoco.cli		97%		100%
 org.jacoco.report		99%		98%
 org.jacoco.ant		98%		99%
 org.jacoco.agent		86%		75%
Total	998 of 24,179	95%	100 of 1,747	94%

Figure 2. Example of JaCoCo report

3 Refactoring

3.1 Dos and don'ts

This chapter aims to provide examples of dos and don'ts by comparing two versions of the example project's code. The first version can be found under the branch `release-v1.0.0`, and was last modified in 2016. This is a prototype version, coming from a time where the author was a novice at the Java language, unaware of most of its conventions and tools to offer. This version will provide most of the don't examples.

This version is compared to the one with the practices contained in the thesis applied. It can be found under the suitably named branch `the-refactoring`. This branch will address the shortcomings of the previously mentioned version, and provide the do examples to solve them. The effectiveness of these examples is further discussed in the later chapters.

3.2 Structure

Referring to a “project’s structure” can mean different things, depending on the observer’s perspective. One can think of low-level concepts, e.g. the inner workings of classes and patterns applied to them, or high-level concepts such as modules and their relations. This chapter addresses multiple viewpoints.

3.2.1 Module structure

Apache Maven also provides support for modules inside projects. This can be thought of as the reverse of inheritance – instead of multiple projects specifying their parent POM, the aggregating project can be split up into modules, specifying a skeleton for its children.

This feature has the welcome benefit of adding another layer of separation between concepts expressed in code. By categorizing code into multiple modules, it becomes much more apparent which part of the code depends on which. Hence, it can be easier to keep inter-dependencies sane and less coupled. Putting thought and effort into the project’s module relations can thus make the code more maintainable and well-structured.

An example of a good candidate could be applications based on the multitier architecture. In effect, this means that the application is comprised of multiple layers, each figuratively stacked on each other. Each layer can communicate with the one ‘above’ or ‘below’, but not others. Manually enforcing this can be no small feat. However, creating a module for each of the tiers can automate this process. By restricting the dependencies of each layer to the one it should be able to reach, it becomes impossible to create unwanted dependencies, since the module will not compile if done so.

As of the point of writing, there is no universal consensus of how these modules should be structured (How Do You Organise Maven Sub-Modules? 2013). One can opt for a flat structure, where the project has a simple list of modules. A tree structure is also possible, since modules can also have their own modules, which in turn can also have their modules, as deep as the developer deems appropriate.

We have applied a simple principle to the example project: anything that could be a standalone product is a separate module. This approach was chosen partly because the codebase contains multiple classes that can be run as a separate application. The original project consisted of a simple project without any modules. This meant that in order to output multiple runnable artifacts, the assembly plugin had to be configured with multiple executions. This resulted in difficult to read configuration and tedious work when a new runnable class was added.

The original codebase can also be conceptualized as a few modules, with a few dependencies on each other. For example, the game rules could be thought of as a module, the network protocol implementation as another module, and the game visualization components as a third.

So, in this sense, it is established that the project would have two kinds of modules: applications and libraries. The libraries provide game-specific functionality and the applications would facilitate the gameplay itself. The refactored version contains the Maven modules as described in Table 1.

Table 1. New module structure

Module	Description
nchess-core	Implementation of the game rules, marshalling functionality, and utility functions related to the game rules.
nchess-net	Contains functionality to handle the networking protocol.
nchess-view	Provides a Swing-based UI component to draw the current game state.
nchess-resources	Contains image files of the chess pieces, and a few example maps.
server	A simple command line based game server.
randomclient	An example implementation of a game client returns randomly chosen moves.
swingdemo	A demo application showcasing the capabilities of the nchess-view module.
swingui	A full-fledged visual client application.

The above module structure has a few benefits over the original structure, which contained everything in a single project. Firstly, having this breakdown at hand it is very apparent how the modules should depend on each other. It would make little sense for the nchess-view module to depend on the nchess-net package, since the former is only concerned with the visuals, and only needs a game state to draw. Hence, networking should not be a concern in this module.

Applications are also contained in their own modules, which removes the need for complicated plugin configurations to produce multiple output JAR files. In addition, a notable positive effect is that now only the applications themselves are aware of their supporting classes. This way, it is impossible to accidentally import an application's utility class where it should not be accessible.

3.2.2 Package structure

One of Java's prominent features is packages. Packages provide a convenient way to group classes, interfaces and others together under a common namespace (Gosling, et al. 2013, 163). This can provide a nice touch of organization and can make browsing the codebase a pleasant experience. However, it remains unclear how to actually organize the codebase into packages.

It is standard practice to define a root package, using the reverse domain name notation (Naming a Package). This avoids the problem of multiple classes – possibly coming from dependencies - having the same name in the project, since two classes can have the same name as long as they are in a different package. It is beneficial to use a domain name that in some way belongs to the developer. For example, the example project's root package is `com.github.elementbound`, as it has been derived from the author's GitHub account.

To further organize the code and make it less prone to the mentioned problem, the package tree can be deepened to provide more nuanced categorization. For this task, the following conventions can be utilized (Vermeulen, et al. 2000, 76).

The Common Reuse Principle

Classes that are used together should be placed in the same package. In other words, if one of the classes is used, all of them are used. These classes are often so closely related that it is not practical to use them in isolation.

An example of this can be found in the `com.github.elementbound.nchess.game` package of `nchess-core`. Inside, the `GameState` class can be found as an aggregator of all the information needed to describe the game's current status. Since this state is just a collection of information, the class is used to retrieve some information from it, like the next player. This is represented by a class named `Player`. Since `Player` instances also need to belong somewhere, it does not make sense to use the class in isolation either. This applies to the other classes found in the package as well.

The Common Closure Principle

Classes that are likely to change at the same time, for the same reason should be kept in the same package. This principle is another way to indicate relations between classes; if classes are so closely coupled that changes in one class affects the other, they should be kept together.

Consider the package `com.github.elementbound.nchess.view` in the `nchess-view` module. This package consists of the class `GamePanel`, a Swing-based UI component, and its supporting classes. These classes are dependent on `GamePanel`, since they implement functionality on top of it. If the `GamePanel` class were to be changed considerably, it would expose information to the outside world differently, thus its dependents, like `DefaultGamePanelListener` would change. However, for example, if the mentioned listener class needed different information in the `NodeSelectEvent` class, then `GamePanel` would need to provide that information. Thus, a change to either of these classes is very likely to result in a change of the other.

The Stable Dependencies Principle

Moving past the inner semantics of an individual package, this principle is concerned with the relations between packages. It states that more stable packages less likely to change should be the dependencies of those packages more likely to change. In

other words, the package's dependencies should be more stable than the package itself.

Consider the package `com.github.elementbound.nchess.net` and its child package, `protocol`. The former class contains the classes `Client` and `Server`, both acting as components to provide networking logic. These two classes depend on the child package `protocol`. The rationale here is that the networking protocol is a strongly established package, changing only when a strong reasoning is present. However, it is more likely that either the `Server` or the `Client` component may change, for example exposing more information via event sources. Such changes would not affect the `protocol` package. On the other hand, introducing a new message type, or changing the fields of an existing one would prove to be a bigger undertaking.

Keeping these principles in mind, the packaging structure will be consistent, easy to navigate and easy to maintain.

3.2.3 Class structure

This chapter addresses the most glaring problems with the internal workings of classes in the original version of the codebase. Since these issues are one of the easiest red flags to spot, these were a strong part of the motivation to refactor the application. This applies in a more general sense as well, since these code smells are a strong indicator of the code's diminishing quality.

Single responsibility principle

Bigger applications often contain a huge amount of both complex and straightforward logic. These pieces need careful organization, otherwise it becomes hard to get a grasp of the project, get a clear understanding of the systems it contains, their relations. Furthermore, maintenance proves to be a difficult task. These issues also lead to higher bug counts, which in turn results in more work for developers.

The single responsibility principle provides an answer to the above by stating every module or class should provide a single, clearly defined part of the logic, and that responsibility encapsulated fully.

"A class should have only one reason to change." (Martin 2003, 95)

Neglecting this principle may result in God objects, an anti-pattern where an object knows too much or does too much (Anti-Patterns and Worst Practices – Monster Objects 2009). Applications employing the God object anti-pattern typically code most of the functionality into one or a few classes. These classes handle most of the information and provide most of the methods, thus becoming God-like, all-knowing and all-encompassing. Other objects do not communicate directly with each other; instead, they do it through a God object. This way the God object becomes tightly coupled to most parts of the program, making maintenance unreasonably difficult.

Consider the Table class in the original version of the codebase, as seen in Figure 3.

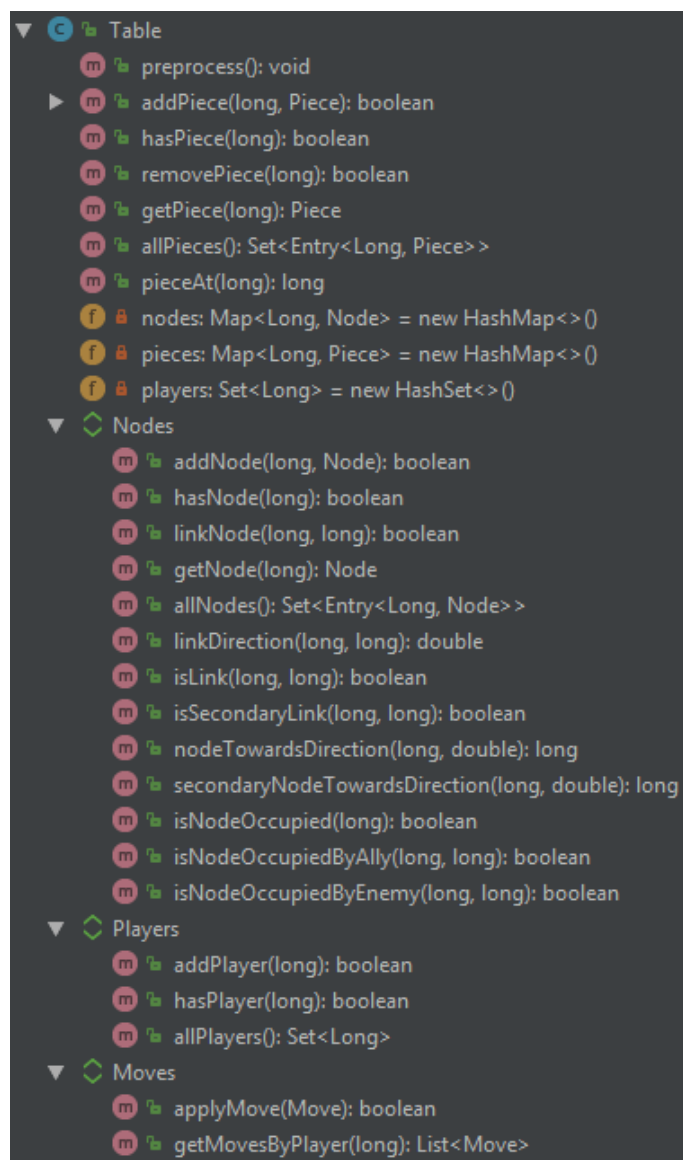


Figure 3. Members of the original Table class

As it is apparent, the Table class handles a large amount of semi-related information (nodes, players, pieces), and provides a huge amount of unrelated logic via methods. This violates the single responsibility principle since the class serves multiple functions. It acts as a data object, aggregates information about the current game state, provides utility functions to work with nodes, utility functions to validate piece moves (see `isNode*` functions) and more.

In order to solve this, the class functionality needs to be divided into multiple parts. Firstly, the multiple responsibilities of the class need to be specified. For the discussed case, they are the following:

- Prepare game table for usage
- Store the game table
- Store the participating players
- Store the chess pieces
- Provide higher level utilities

The above list provides ample opportunity for splitting. If possible, it is also beneficial to keep the intention of the original class in mind. The Table class, among its many responsibilities, provides an overview of the current state of the game: how does the table look, where do the pieces stand on it, etc.

According to the above, the Table class can be refactored as seen in Figure 4.

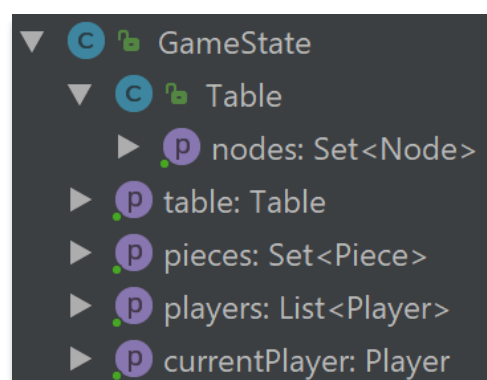


Figure 4. GameState, based on Table

The newly created GameState class encompasses information about the game state. It contains an instance representing the chess table, collections of pieces, players, and the current player. This structure also better represents the concepts used to describe the game rules.

This also opens up a chance to address another problem; namely the Table class was wrongly named. It did not represent the chess table, it represented the game state. Now, the Table class contains the information necessary to describe and work with the chess table, nothing else.

Mutable and immutable objects

An immutable object is an object, whose state can be altered in no way after it has been created. (Goetz 2006, 31) All the necessary data needs to be provided up-front, since it cannot be later added to the instance.

In contrast, a mutable object is an object whose state can be modified during the flow of the program.

An example of immutable objects in the Java language is the String class. Once a String instance is created, the text it represents will not change. If an operation is applied to the string, for example, concatenation, neither of the involved instances change. Instead, a new instance is returned with the resulting text.

Immutable classes have multiple benefits (Bloch 2008, 73), such as:

- Easier usage
- Simpler design and implementation
- Easy to share between multiple instances
- Inherently thread-safe

In general, even if a class cannot be designed as immutable, it is a good consideration to reduce mutability as much as possible.

The original codebase neglected this aspect of class design, which later posed problems, as it was difficult to handle game states, and it did not fit conceptually either. A game state is a discrete representation. Thus, any changes applied to it result in a new, different state. In this regard, representing the game state with an immutable class suits the concept well.

One of the caveats of immutable class design is collection. Consider an example, where an immutable class contains a list. The list can be retrieved. The list reference cannot be set to point to a different list (no setter function). So far, this could be considered an immutable class. The caveat here is that once the list is retrieved via a getter function, the list itself is not necessarily immutable and can be changed.

This was a minor problem during development, since simply returning an unmodifiable view solves the issue (Collections. Java Platform SE7).

Spaghetti code

In the context of this thesis, Spaghetti code refers to writing procedural code in an object-oriented language such as Java, which results in lengthy methods that are hard to read, maintain or test. Since the programmer needs to keep multiple things in mind while working on said methods, it is prone to errors as well.

Consider the original codebase's `JsonTableLoader` class as an example. While it also is a good showcase of violating the single responsibility principle, the focus is on its parse function. The function itself is 204 lines long, so it could be considered lengthy. To compensate this, the author has placed comments to define sections, such as seen in the snippet below:

```
//=====
//Parse links
JsonValue links = root.get("links");
if(links.getValueType() != JsonValue.ValueType.ARRAY) {
    System.out.println("Links array not an actual array!");
    return false;
}
```

This is a good indicator that the function contains too much logic and should be split up into multiple smaller functions. It is a key concern that when extracting pieces of logic into methods, the method name and signature describes the logic inside concisely but well. This way, the original function does not only become shorter, but easier to read and understand as well. Instead of the 204 lines of the original function, this snippet is to be considered:

```
// Extract data
Set<Node> nodes = parseNodes(root);
List<Player> players = parsePlayers(root);
Set<Piece> pieces = parsePieces(root, nodes);
Set<Link<Node>> links = parseLinks(root).stream()
    .map(pair -> Pair.of(...))
    .map(pair -> new Link<>(pair.getLeft(), pair.getRight()))
    .collect(Collectors.toSet());
```

The snippet contains descriptive method names, variable names, and the logic is expressed in a manner that can be understood at the first read.

3.3 Testing

As mentioned earlier, to be able to reliably deliver functionality and releases, it is advisable to write tests. The benefit is that it is continuously ensured that the tested items behave as intended, avoiding errors and reducing bug count through the project's lifecycle which can increase velocity since more time is spent on the required features and less time is spent analyzing and fixing existing but erroneous code.

Tests can come in multiple flavors, which are discussed in the following chapters, without attempting to be comprehensive.

3.3.1 Unit testing

Unit testing, aptly named, refers to the testing of individual units of the code. In object-oriented programming and in Java, the units in question are the classes that make up the application.

When creating unit tests, one needs to come up with assertions about the class' behavior and define under what conditions should the assertions be true. To express this in code, one writes a test class for the class to be tested, and adds test cases. Each of these assertions is manifested as a test case.

It is advised that each test case is divided into three sections: given, when, then.

The given section establishes the conditions under which the assertion is true, and defines expected results if applicable. Other, case-specific setups belong here as well.

The when section invokes the class under testing and stores the result if necessary.

The then section contains the assertion expressed through the constructs provided by the testing framework used.

In other words, the given-when-then trio can be illustrated as such:

Given these conditions are true,

When this operation is performed,

Then these conditions must be true.

It is also recommended to express this structure in code via comments.

Oftentimes classes rely on other classes to delegate operations to them and perform further operations on the result. In this scenario, the classes being depended on can be called collaborators.

When writing unit tests, it is important to focus on the behavior of the class and not on that of the collaborators. If unit tests are written and maintained diligently, then the collaborators' behavior is tested in their own corresponding test classes and does not need to be tested in the current class's tests as well. This would also result in redundant tests, which are also prone to error.

To address this, one can use mocks. Mock objects are objects mimicking the interface of some other object, however, its responses can be configured for each test case. They are perfect for substituting the tested class's collaborators. Using mocks this way alleviates two problems. Firstly, the behavior of the collaborators does not need to be tested, since their responses are always as expected. Second, even if for some reason the collaborators' behavior fails, the tests of the current class will not, because they do not rely on the collaborators' implementation.

Consider the class `EventSourceTest` from the `nchess-core` module. An event source has two requirements:

- Pass events to all subscribers
- Don't pass events to non-subscribers

This is reflected in the test class as well, described by the method names.

Looking at the implementations of the test cases, the sections are apparent, making it easy to follow the different steps of the test case. Since the assertions are in their own section, the exact details of the test case's requirements are easy to find.

As reflected on in chapter 2.5.3, it is also beneficial to agree on a minimal test coverage and enforce that limit. The question of minimal code coverage ratio has no universal answer since it varies from project to project. However, a related chapter from *The Way of Testivus* can provide a prosaic solution, as can be read in Appendix 3.

To summarize, the answer depends on the code; nevertheless, 80% is a good rule of thumb.

For further principles of unit testing, see Appendix 2.

3.3.2 Integration testing

While unit testing focuses on a single unit (class), that alone may not be satisfactory. In certain cases, it needs to be verified that said classes interact in a specific way under certain conditions.

It is to be noted, that the term "integration testing" is blurred in meaning, with generally two versions of the concept. This chapter highlights the narrow integration tests, which also involves mocking (Fowler, *IntegrationTest* 2018).

For any given module interaction, integration testing is concerned with exercising the part of the code that calls the other module to interact with. Moreover, only the module under test is instantiated. As for the other module, only a mocked version is used, where the module's public interface is replaced with mocked objects. This way, all expected module interactions can be covered with more lightweight tests.

4 Code metrics

4.1 Benefits of metrics

Code metrics is a measurement that describes certain aspects of an application's code, by providing a grading system, a number, or some other form of information.

Since arguing about a development project's progress based on subjective grounds is difficult, code metrics can be used to provide accurate and objective points to consider.

Thus, it can be said that code metrics is beneficial because it provides objective insight into the source code.

As such, it is used to get a proper view of the example project's progress and the effect of the changes introduced to it.

4.2 SonarQube, SonarCloud

SonarQube is an open-source platform to perform automated code inspection. In order to do this, it employs static code analysis to detect potential bugs, code smells and security vulnerabilities. In the years since its inception in 2007, it has become a mature and widely used tool, used by 85,000 organizations. Additionally, it presents its reports in the form of an intuitive dashboard, providing a multitude of configuration options to better suit the project at hand including setting up a Quality Gate, describing a code quality threshold that must be met.

Built on SonarQube, SonarCloud is a service that provides SonarQube's features in the form of an accessible website. Users can sign up, submit their code for review, and SonarCloud's automated process gathers all the necessary data for its metrics. (SonarQube 2018)

5 Results

To evaluate how beneficial it was to introduce the discussed principles to the project, the aforementioned SonarCloud platform was used.

As it can be seen in Figure 5, the original code contained a huge amount of code smells. The project's bug rating was very negative, while also containing a considerable amount of vulnerabilities.

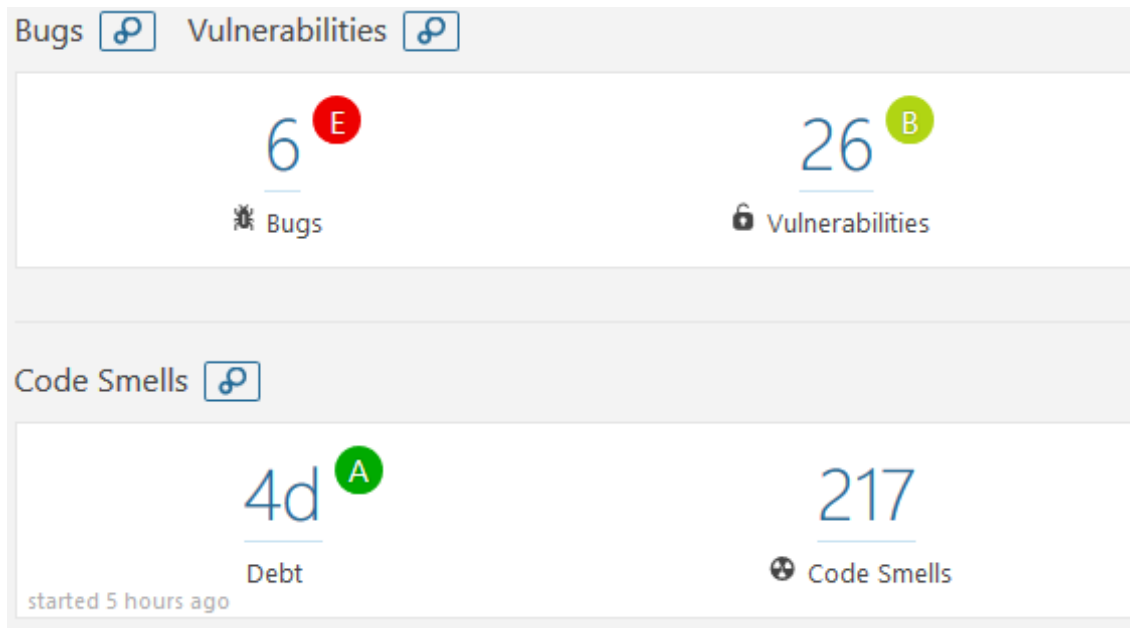


Figure 5. Original code report

From the six bugs, two were conditionals that always evaluate to false, as a quick way to toggle features at compile time. While a conditional always evaluating to the same value could be a major bug in production code, in this case, it is of little consequence. However, their report is still useful, as it makes the developer less likely to forget those conditionals.

The rest of the bugs reported were legitimate concerns:

- The Move class overrides "equals()" but not "hashCode()".
- Two cases where a resource should be closed either with a try-with-resources construct or a finally clause.
- An InterruptedException was wrongly handled.

The majority of the 26 reported vulnerabilities stemmed from the inexperienced language usage, including the lack of coding style and proper pattern usage. This statement is true for a large amount of code smells as well.

The reports for the resulting source code can be seen inion pieces Figure 6.

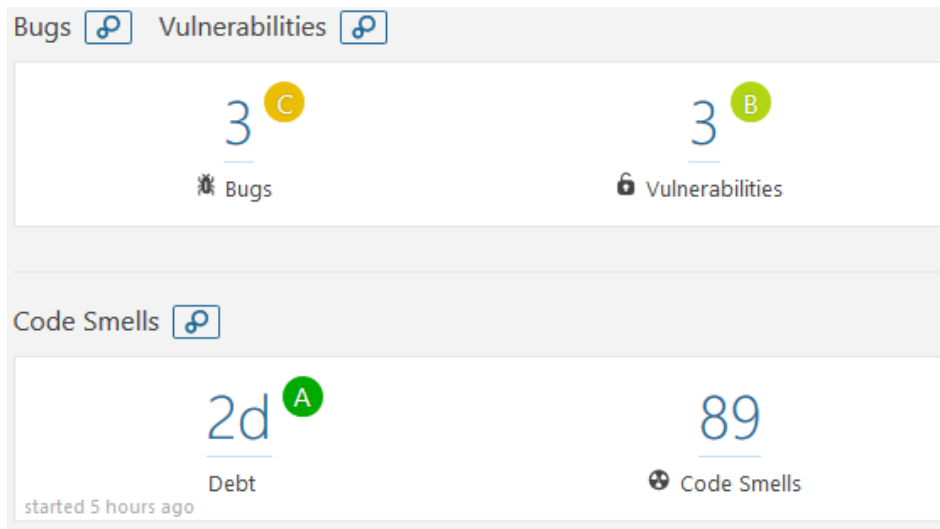


Figure 6. New code report

Only three bugs are present, every one of them reporting that an Optional's value was queried before checking if the value is present. This can be attributed to the indirect calls to the presence check. These bugs are considered false alarms by the author, although arguably they can be considered a code smell.

A large decrease can be noticed in the vulnerabilities count which is now down to three. These are legitimate concerns, even though reported as minor:

- Exceptions should be logged using a logger.
- A member should be made protected instead of private.

As a result of applying the discussed principles, the amount of debt attributed to code smell was halved and the count of reported items reduced to 89.

6 Conclusion

The objective of this thesis was to introduce and apply concepts that improve the code quality of the example project. A tool was used to track progress with objectively measurements.

Examining the results discussed in the previous chapter, it can be said that the objective set out in the introduction of the thesis was achieved. The code quality has improved in a number of ways, and different mechanisms were put in place to keep the

code quality from decreasing. This proves that the topics discussed in the thesis do improve the project's code. In turn, these improvements also lower the effort required to maintain and further develop nchess.

However, it is important to note that the thesis provides the reader only with the first steps. Among the immense amount of books, articles, blog posts, opinion pieces, and researchers are a long list of concepts and useful ideas. Many of these concepts can be useful in a certain context, however, when developing a different kind of software, different ideas are applicable. The point is that one must constantly seek further knowledge since software development is still a rapidly growing field.

References

- Anti-Patterns and Worst Practices – Monster Objects*. 2009. <https://lostechies.com/chrismissal/2009/05/28/anti-patterns-and-worst-practices-monster-objects/> (accessed May 27, 2018).
- Bloch, Joshua. *Effective Java*. Addison-Wesley, 2008.
- Collections*. *Java Platform SE7*. February 27, 2008. [https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableCollection\(java.util.Collection\)](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableCollection(java.util.Collection)) (accessed May 27, 2018).
- Dependency Scope*. 2018. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope (accessed May 27, 2018).
- Foote, Brian, and Joseph Yoder. *Big Ball of Mud*. Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, Boston, MA: Addison-Wesley Longman Publishing Co, 1999.
- Fowler, Martin. *IntegrationTest*. 2018. <https://martinfowler.com/bliki/IntegrationTest.html> (accessed May 27, 2018).
- . *Technical Debt*. 2003. <https://martinfowler.com/bliki/TechnicalDebt.html> (accessed May 27, 2018).
- Goetz, Brian. *Java Concurrency in Practice*. Addison Wesley Professional, 2006.
- Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. 2013.
- How Do You Organise Maven Sub-Modules?* 2013. <https://dzone.com/articles/how-do-you-organise-maven-sub> (accessed May 27, 2018).
- Introduction to Maven Plugin Development*. 2018. <https://maven.apache.org/guides/introduction/introduction-to-plugins.html> (accessed May 27, 2018).
- Introduction to the POM*. 2018. <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html> (accessed May 27, 2018).
- Martin, Robert C. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 2003.
- Maven Coordinates*. 2018. http://maven.apache.org/pom.html#Maven_Coordinates (accessed May 27, 2018).
- Naming a Package*. n.d. <https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html> (accessed May 27, 2018).
- SonarQube*. 2018. <https://www.sonarqube.org/about/> (accessed May 27, 2018).

Vermeulen, Al, et al. *The Elements of Java™ Style*. CAMBRIDGE UNIVERSITY PRESS, 2000.

What is Maven? 2018. <https://maven.apache.org/what-is-maven.html> (accessed May 27, 2018).

Appendices

Appendix 1. An example Checkstyle configuration

```

<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
    "-//Puppy Crawl//DTD Check Configuration 1.3//EN"
    "http://checkstyle.sourceforge.net/dtds/configuration_1_3.dtd">

<module name="Checker">

    <property name="fileExtensions" value="java, properties, xml"/>
    <module name="NewlineAtEndOfFile"/>
    <module name="Translation"/>
    <module name="FileLength"/>
    <module name="FileTabCharacter"/>

    <!-- Miscellaneous other checks. -->
    <module name="RegexpSingleline">
        <property name="format" value="\s+$"/>
        <property name="minimum" value="0"/>
        <property name="maximum" value="0"/>
        <property name="message" value="Line has trailing spaces."/>
    </module>

    <!-- Checks for Headers -->
    <module name="TreeWalker">

        <!-- Checks for Javadoc comments. -->
        <module name="JavadocType"/>
        <module name="JavadocStyle"/>

        <!-- Checks for Naming Conventions. -->
        <module name="ConstantName"/>
        <module name="LocalFinalVariableName"/>
        <module name="LocalVariableName"/>
        <module name="MemberName"/>
        <module name="MethodName"/>
        <module name="PackageName"/>
        <module name="ParameterName"/>
        <module name="StaticVariableName"/>
        <module name="TypeName"/>

        <!-- Checks for imports -->
        <module name="AvoidStarImport"/>
        <module name="IllegalImport"/>
        <module name="RedundantImport"/>
        <module name="UnusedImports">
            <property name="processJavadoc" value="false"/>
        </module>

        <!-- Checks for Size Violations. -->
        <module name="LineLength">
            <property name="max" value="120"/>
        </module>
        <module name="MethodLength"/>
        <module name="ParameterNumber"/>
    </module>

```

```

<!-- Checks for whitespace --
>
<module name="EmptyForIteratorPad"/>
<module name="GenericWhitespace"/>
<module name="MethodParamPad"/>
<module name="NoWhitespaceAfter"/>
<module name="NoWhitespaceBefore"/>
<module name="OperatorWrap"/>
<module name="ParenPad"/>
<module name="TypecastParenPad"/>
<module name="WhitespaceAfter"/>
<module name="WhitespaceAround"/>

<!-- Modifier Checks -->
<module name="ModifierOrder"/>
<module name="RedundantModifier"/>

<!-- Checks for blocks. You know, those {}'s -->
<module name="AvoidNestedBlocks"/>
<module name="EmptyBlock"/>
<module name="LeftCurly"/>
<module name="NeedBraces"/>
<module name="RightCurly"/>

<!-- Checks for common coding problems -->
<module name="AvoidInlineConditionals"/>
<module name="EmptyStatement"/>
<module name="EqualsHashCode"/>
<module name="IllegalInstantiation"/>
<module name="InnerAssignment"/>
<module name="MagicNumber">
  <property name="severity" value="warning"/>
</module>
<module name="HiddenField">
  <property name="ignoreConstructorParameter"
value="true"/>
  <property name="ignoreSetter" value="true"/>
  <property name="severity" value="warning"/>
</module>
<module name="MissingSwitchDefault"/>
<module name="SimplifyBooleanExpression"/>
<module name="SimplifyBooleanReturn"/>

<!-- Checks for class design -->
<module name="FinalClass"/>
<module name="HideUtilityClassConstructor"/>
<module name="InterfaceIsType"/>
<module name="VisibilityModifier">
  <property name="protectedAllowed" value="true" />
</module>

<!-- Miscellaneous other checks. -->
<module name="ArrayTypeStyle"/>
<module name="TodoComment">
  <property name="severity" value="warning"/>
</module>
<module name="UpperEll"/>
</module>
</module>

```

Appendix 2. The Way of Testivus – Excerpt

The Way of Testivus

If you write code, write tests.

Don't get stuck on unit testing dogma.

Embrace unit testing karma.

Think of code and test as one.

The test is more important than the unit.

The best time to test is when the code is fresh.

Tests not run waste away.

An imperfect test today is better than a perfect test someday.

An ugly test is better than no test.

Sometimes, the test justifies the means.

Only fools use no tools.

Good tests fail.

Appendix 3. Testivus' wisdom on code coverage

Early one morning, a programmer asked the great master:

“I am ready to write some unit tests. What code coverage should I aim for?”

The great master replied:

“Don't worry about coverage, just write some good tests.”

The programmer smiled, bowed, and left.

...

Later that day, a second programmer asked the same question.

The great master pointed at a pot of boiling water and said:

“How many grains of rice should put in that pot?”

The programmer, looking puzzled, replied:

“How can I possibly tell you? It depends on how many people you need to feed, how hungry they are, what other food you are serving, how much rice you have available, and so on.”

“Exactly,” said the great master.

The second programmer smiled, bowed, and left.

...

Toward the end of the day, a third programmer came and asked the same question about code coverage.

“Eighty percent and no less!” Replied the master in a stern voice, pounding his fist on the table.

The third programmer smiled, bowed, and left.

...

After this last reply, a young apprentice approached the great master:

“Great master, today I overheard you answer the same question about code coverage with three different answers. Why?”

The great master stood up from his chair:

“Come get some fresh tea with me and let’s talk about it.”

After they filled their cups with smoking hot green tea, the great master began to answer:

“The first programmer is new and just getting started with testing. Right now he has a lot of code and no tests. He has a long way to go; focusing on code coverage at this time would be depressing and quite useless. He’s better off just getting used to writing and running some tests. He can worry about coverage later.”

“The second programmer, on the other hand, is quite experience both at programming and testing. When I replied by asking her how many grains of rice I should put in a pot, I helped her realize that the amount of testing necessary depends on a number of factors, and she knows those factors better than I do – it’s her code after all. There is no single, simple, answer, and she’s smart enough to handle the truth and work with that.”

“I see,” said the young apprentice, “but if there is no single simple answer, then why did you answer the third programmer ‘Eighty percent and no less’?”

The great master laughed so hard and loud that his belly, evidence that he drank more than just green tea, flopped up and down.

“The third programmer wants only simple answers – even when there are no simple answers ... and then does not follow them anyway.”

The young apprentice and the grizzled great master finished drinking their tea in contemplative silence.