Software Quality – Part 1

David Mendes
#44934
dj.mendes@campus.fct.unl.pt

Francisco Cunha
#45412
fjm.cunha@campus.fct.unl.pt

ABSTRACT

In this paper we present the analysis work that we did for two Java projects of considerable size and complexity. Using different OO metrics, tools and thresholds, we give an overview of the software quality of these systems, diving deeper on unusual situations and symptoms of code smell that we found along the way. As we compare the two, we showcase how base architectural decisions translate into the most notable differences between two projects of similar dimension.

1. INTRODUCTION

Errors in software development are inevitable, but good development practices on top of a solid architectural foundation could go a long way to reduce them. On this paper, we showcase and analyze two Java systems of non-trivial size nor complexity, Bootique and JHotDraw. Using various well-known, battle-tested OO metrics and tools, we present an overview of the quality of these systems. Furthermore, we bring up design patterns and code smells, never leaving out potential refactoring decisions.

SYSTEMS DESCRIPTION

1.1 Bootique (v0.25)

Figure 1 – Bootique – Framework for Runnable Java Apps

Bootique is designed to be a platform for developing modern Java applications, making use of dependency injection and configuration to create full container-less running Java apps. It allows the development of fully functional apps (by module composition) that run as if they were simple commands. One can use it to create not only REST services, web apps and database migration tasks, but also fully-fledged, multi-functional apps or even microservices.[1] [2]

1.2 JHotDraw (v5.2)

JHotDraw is a Java GUI framework for technical and structured two-dimensional graphics.[3]

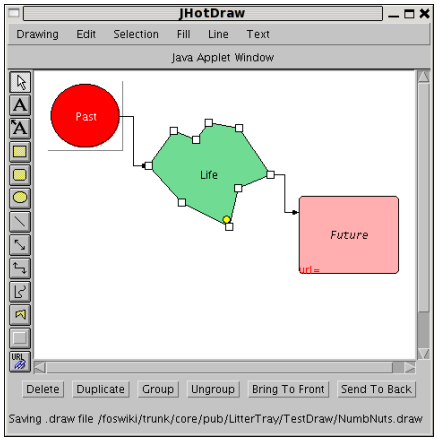

Figure 2 – JHotDraw, Java GUI for 2D Graphics

2. SOFTWARE METRICS

According to Lanza et al. "Metrics are good at summarizing particular aspects of things and detecting outliers in large amounts of data" [1], and even though they scale well, they cannot be seen as definitive proof that a piece of code is "good" or "bad".

That said, including metrics assessments during development is a reasonable way to make sure the produced code keeps a high level of maintainability and fault-tolerance.

Our analysis aims not to see these metrics as the "holy-grail" of software-quality. Though we take them in high consideration while suggesting changes to the aforementioned systems, we always keep in mind that they can't replace good taste and experience.

3. TOOLS / SOFTWARE USED

For reproducibility purposes, our analysis was made using OSX (High Sierra) and Eclipse version 2018-09 (4.9.0), which are the latest stable versions as of the time of writing

[1] Bootique is a minimally opinionated platform for modern runnable Java apps.: bootique/bootique. Bootique Project, 2018.

[2] A. Adamchik, "Bootique: a Minimally Opinionated Platform for Modern Java Apps," Medium, 17-Feb-2016.

[3] "JHotDraw Start Page." [Online]. Available: https://www.jhotdraw.org/. [Accessed: 01-Oct-2018].

3.1 SourceMiner (v1.0.0)

"SourceMiner is a multiple view environment (MVE) designed and implemented as an Eclipse plug-in to enhance software comprehension activities".[4] It enables the developer to understand, in a visual and interactive manner, what are the pitfalls and quality issues with their own software.

Its visual aiding features include Tree-Maps, Graphs and even matrices that give information about a number of essential metrics, regarding software quality, from which we point out the McCabe Complexity, Size, Coupling (Afferent / Efferent / Global). A few more are present, all with the end goals of enabling a more complete view of the system and tackling some blind-spots in the design of applications.

3.2 Metrics (v1.3.8)

The Metrics plugin provides software quality metrics alongside dependency analysis. Not only it provides values for each produced metric (including mean and standard deviation), but it is also capable of creating graphs according to the program's dependencies.

This plugin can be integrated into an application's build cycle, to warn the developer of possible problems ("range violations"). This enables continuous monitoring of the state of the system, during both development and maintenance phases. The plugin also enables metrics exporting for further analysis and registry.[5]

3.3 inFusion (v7.2.7)

inFusion works as a standalone application, and like the previous two, provides system metrics. inFusion is specifically tailored towards object-oriented (OO) programming.

It also provides a very intuitive and detailed overview pyramid analysis, alongside other interpretations of the systems complexity, coupling and encapsulation.[6]

3.4 EclEmma (v3.1.1)

EclEmma is a code coverage plugin for Eclipse, a port of the original EMMA library developed by Vlad Roubtsov.[7] The tool provides a good set of features, from which we highlight the Coverage Overview and the Source Highlighting. For more details, please refer to the official website.[8]

EclEmma can be installed via the Eclipse Marketplace (simply searching for 'EclEmma') or the update site[9].

3.5 SonarLint (v4.0)

SonarLint is an open-source code linter that offers full integration with Eclipse.[10] There's often not a lot of remarkably different features among linters, and SonarLint is no exception. One thing that's worth nothing is that although opinionated, SonarLint grants a nice degree of configuration freedom, so that one can fine-tune analysis settings as they wish.

As a fully-fledged static analysis tool, a linter can point out two classes of issues. The first class is that of stylistic issues, such as no closing semi-colon on declarations, or multiple instructions per LoC. The second class regards the problematic issues, such as errors, bugs and suspicious constructs. [11]

3.6 JSpIRIT (v1.0) and JDeodorant (v5.0.68)

Both JSpIRIT[12] and JDeodorant[13] were used to detect code smells automatically and recommend appropriate refactoring to resolve them. Whilst the former supports the identification of some of the most well-known code smells - following the detection strategies presented by Lanza et al. [1] - the latter complements it with additional information on conditional type checking and duplicate code.

4. Bootique system analysis

4.1 inFusion

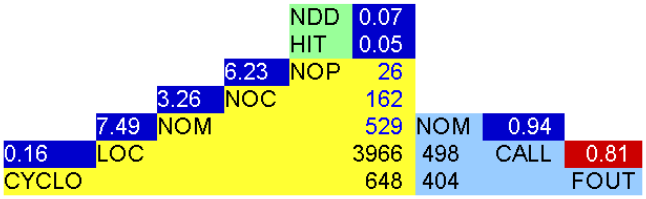

Figure 3 – Bootique's Overview Pyramid created with inFusion

We used inFusion to generate our system's Overview Pyramid. The three key metrics quantified by the Overview Pyramid are the Size and Complexity (shown in yellow), Coupling (shown in blue) and Inheritance (shown in green).

We will start our analysis with the Size and Complexity, conducted in a top-down approach. At the higher level, we can tell that the system's classes tend to be organized in rather fine-grained packages. In other words, we can see that there are few classes per package - on average, around six of them, as we can tell by the index.

The classes tend to be rather small themselves, containing, on average, three to four methods each, placing them comfortably on the "Low" column of the thresholds proposed by Lanza et al. [1]

Methods are also short and simple, averaging seven to eight lines of code each, the latter with a remarkably low cyclomatic complexity.

---

[4] "Sourceminer," *SourceForge*. [Online]. Available: https://sourceforge.net/projects/sourceminer/. [Accessed: 01-Oct-2018].

[5] "Metrics 1.3.6." [Online]. Available: http://metrics.sourceforge.net/. [Accessed: 01-Oct-2018].

[6] https://moodle.fct.unl.pt/mod/resource/view.php?id=290510

[7] "EMMA: a free Java code coverage tool." [Online]. Available: http://emma.sourceforge.net/. [Accessed: 19-Oct-2018].

[8] "EclEmma - Java Code Coverage for Eclipse." [Online]. Available: https://www.eclemma.org/. [Accessed: 19-Oct-2018].

[9] "EclEmma - Installation." [Online]. Available: https://www.eclemma.org/installation.html. [Accessed: 19-Oct-2018].

[10] "Fix code quality issues before they exist | SonarLint." [Online]. Available: https://www.sonarlint.org. [Accessed: 20-Oct-2018].

[11] "lint (software)," Wikipedia. 10-Aug-2018.

[12] "JSpIRIT - Santiago Vidal." [Online]. Available: https://sites.google.com/site/santiagoavidal/projects/jspirit. [Accessed: 19-Oct-2018].

[13] "JDeodorant | Eclipse Plugins, Bundles and Products - Eclipse Marketplace." [Online]. Available: https://marketplace.eclipse.org/content/jdeodorant. [Accessed: 19-Oct-2018].

Moving on to the coupling analysis, we can see that the system's methods present low intensity but high dispersion. In other words, the few methods that they tend to call are usually from many other classes.

Class hierarchies tend to be rather sparse, i.e. there are mostly standalone classes and few, flat inheritance relations. This can be an indicative that the system's authors are opting for composition over inheritance. Furthermore, inFusion did not detect the presence of *God* nor *Brain Classes*.

Finally, inFusion also informs us that the system contains three data classes. There are opposite views on this. While some "OO purists" argue this is a code smell[14], others favor it for relationship modeling or ORM construction.

Upon further analysis of these classes we realized they are somewhat "special cases" of *Data Classes*.

From our research, each of them is used only within the package where they were defined. Yet, many - if not all - of their methods and constructor(s) are public instead of package-protected, and it's not clear why.

On the instance variables *Encapsulate Field* was applied, as they are private and only accessible via getter methods.

The absence of setter methods implies immutability after instantiation; in other words, the data is not being mutated by other classes. This indicates an application of the *Remove Setting Method*. Although technically redundant, we argue that marking the private instance variables with the final keyword would increase the authors' intent, even in cases like this.

Most importantly, they seem to be used strictly as immutable *Parameter Objects*. And here's where it gets interesting: in cases like this where the parameter object does not have any explicit operations of their own, aren't we actually introducing a Data Class smell in favor of readability and conciseness? Whether that is fundamentally a "good" or "bad" idea is something we can't answer for sure.

4.2 Metrics

Figure 4 – Software Quality Metrics captured by the Metrics Plugin

4.2.1 Cyclomatic Complexity

On average, the Cyclomatic Complexity fits into the first level of the McCabe threshold table. With a mean value of 1.533, we can understand that most functions are very simple, following a single execution flow.

Despite this fact, there are some alarming outliers. The maximum complexity found reports a value of 22. The function in question presents a seriously big switch statement, which leads to a control flow graph containing a great number of decision points. Even though the number rings some alarms, there is no simple solution to resolve the issue at hand, as the function needs to determine a data type provided in the function argument by matching to all known types (primitive and non-primitive) to provide the correct answer.

4.2.2 High Number of Parameters

The number of parameters that our system's methods take in averages at 0.92. This is positive; we'll go in depth shortly. However, we can see that there's at least one method that takes in seven parameters.

We argue that methods that take in a high number of parameters introduce a considerable mental overhead for the programmer. Oftentimes, the order of the parameters is not logically ordered, making those methods' calls difficult to reason about. Some languages, such as Kotlin or Python, support named parameters to ease the readability of such calls. Java, however, does not.

More so, the presence of such methods can be indicative of a code smell.

On his book Clean Code [2], Robert C. Martin explains that the optimal number of parameters for a function (or, in our case, method) is zero (niladic function). One and two parameters are also acceptable, but three or more should be avoided whenever possible. The reasoning behind this is that a high number of parameters could mean that a function/method is doing too much work, violating the Single Responsibility Principle.

Typically, splitting it in multiple, smaller functions could be a way to (indirectly) solve this problem. Alternatively, the encapsulation of multiple parameters in a single domain-specific POJO could also be helpful - the so-called Parameter Object pattern.

But in this project the methods that take in the biggest number of parameters are class constructors. For these cases, we find the introduction of the Builder pattern (referred by J. Bloch on his book, Effective Java [3], not to be confused with the Gang of Four builder pattern) to be a very viable refactoring choice. In fact, by applying this creational pattern, one would be able to improve the semantics and readability of those calls. [15]

4.2.3 Nested Block Depth

Nested Block Depth analysis revealed that, on average, the depth was 1.247 with a Standard Deviation as small as 0.727. Even considering the deviation, this fits into the category we would call "Regular/Casual", in terms of nested block depth [4]. The biggest outlier comes with a value of six. A value this high makes it so bugs are more likely to appear and become harder to detect and resolve. Furthermore, it also turns the code review and analysis much harder, as one would need to keep track of the different indentation levels to follow the execution flow. In this particular case, a solution to reduce the nested block depth would be to join nested

[14] "Design Patterns and Refactoring." [Online]. Available: https://sourcemaking.com/. [Accessed: 01-Oct-2018].

[15] "Telescoping Constructor Pattern alternatives | Vojtech Ruzicka's Programming Blog." [Online]. Available: https://www.vojtechruzicka. com/avoid-telescoping-constructor-pattern/. [Accessed: 20-Oct-2018].

conditionals into a single one, as some are mutually exclusive. By doing so, one would achieve a flatter, more readable structure.

4.2.4 Classes and Methods

Out of the four most populated classes (27, 23, 23 and 20 methods, respectively), the first three belong to the same package. In total, these four hold almost 12% of Bootique's method count. Below that it's balanced, with most classes holding three to six methods each.

It's worth mentioning that around 6% of the system's Java files have zero methods. We were intrigued by this and went for a deeper analysis. Interestingly, these files were very heterogeneous among each other. For instance, files such as ConfigurationSource.java would implement a Facade[16], while others would serve as tag/marker interfaces (PolymorphicConfiguration.java comes to mind). There were also a couple of Enum classes (usually file-private) and a good amount of Annotation classes.

4.2.5 Coupling Intensity (Calls / NoM)

With a small value of 0.94, we can understand that the system presents low coupling intensity, since each method calls only close to 1 other method on average.

4.2.6 Coupling Dispersion (Fanout / Calls)

From the information given by the Overview Pyramid, we can conclude that coupling dispersion is the main issue with the system. The value of 0.81 means that for every five operation calls, four of them are made to other classes. This indicates a high coupling dispersion; in other words, classes don't really appear to be self-contained, and thus, any change made to a method can have an undesired ripple effect, one that can affect a great part of the system.

4.2.7 Static Attributes and Singletons

The plugin also informs us that around 18% of the system's attributes (read: variables and values) are static. Having such a large percentage of static variables could be a problem, and so we checked the source code. Most of these static attributes were values (constants), which is a good thing, but we stumbled upon a few singletons. The Singleton pattern is rather controversial in the OO community, and the adoption of Dependency Injection could be a viable replacement, as it would loosen up the system's coupling and ease testability. Nevertheless, the Singleton pattern surely has pros and cons that should be weighted accordingly in a deeper analysis.

4.3 SourceMiner

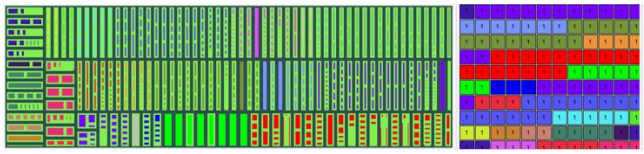

Figure 5 – Bootique SourceMiner Analysis regarding McCabe Complexity (Classes)

By analyzing the Tree-Map extracted with the SourceMiner plugin, we can see that, in terms of complexity, the system is extremely well distributed. This confirms what we had already seen with the Metrics tool. The system presents just a few outliers.

While comparing the complexity results given by the Metrics and the SourceMiner plugin, we noticed some drastic differences in the way they classify things. For example, the method that Metrics classified as having the biggest complexity value (22), only showed a complexity value of 2 in SourceMiner. Upon some analysis, we concluded that while Metrics considers every outcome of a switch as a different decision point - and thus, dramatically increases the method complexity - SourceMiner does not linearly correlate the switch statement cases to control decision points, explaining why the overall complexity of the method is drastically reduced.

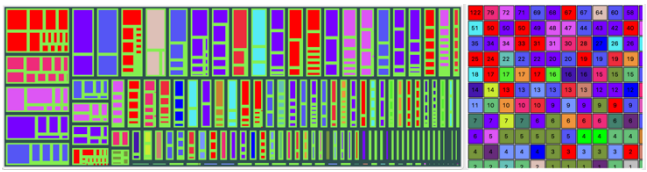

Figure 6 - Bootique SourceMiner Analysis regarding Coupling (Classes)

The Tree-Map in figure 6 confirms the information we already had in regard to coupling, via the Metrics plugin, as it clearly denotes some worrying discrepancies that could be troublesome for the maintainability of the project.

Diving deeper and bearing in mind the problem detected by the both the Overview Pyramid and SourceMiner - coupling dispersion - we did some further analysis on the coupling dynamics of this application.

We used the Instability Index formulae to determine which classes were the most unstable, and thus more prone to the creation of a ripple effect that would break the application if a bug were to be introduced there.

The instability Index - an indicator of the resilience to change - can be calculated by dividing the Efferent Coupling by the Total Coupling (sum of Afferent and Efferent).[17]

While a value closer to zero for this index indicates stability, a value closer to one indicates instability. High instability means that, if a bug were to be introduced, the potential to create a disastrous ripple effect is much higher than on a section with low instability. With this being said, it's important to keep in mind that, as with most other metrics and indexes, a high Instability Index is not a guarantee of a need to refactor. Each case should be analyzed independently, as one could argue that, sometimes, it could actually be a good design decision.

The class with the biggest instability index was the Application.optionMetadata.java (0.86), a class that aims to create a descriptor for a command-line option. This high value is understandable, given the system's functionality: this class is highly demanded by the other parts of the system. Upon further code analysis, the code is straightforward which should, in theory, decrease the chance of bugs being introduced by this component. Still, a remark can be made that, if there is a need to introduce significant changes in this

[16] tutorialspoint.com, "Design Patterns Facade Pattern," www.tutorialspoint.com. [Online]. Available: https://www.tutorialspoint.com/design_pattern/facade_pattern.htm. [Accessed: 01-Oct-2018].

[17] A. Erickson, "Using Metrics to Find Out if Your Code Base Will Stand the Test of Time," 2010.

class, the derived change to the system becomes, obviously, much harder to trace.

4.4 EclEmma

After running the test cases for Bootique, EclEmma reported an underwhelming code coverage of 38.6%. Untested third-party code is unpredictable, potentially dangerous code that should be used with caution. This would be a considerable "red flag", had the project been tagged production-ready. And that's the key point we want to highlight here: Bootique is not a finished product. As of the time of writing, Bootique is on version 0.25. A work in progress like this usually implies breaking changes and an overall unstable API, and so test coverage will probably increase as the project matures towards the Version 1.0 Milestone.[18]

4.5 JSpIRIT and JDeodorant

We analyzed the system with both tools and ended up with similar results. Given that fact, we based our judgment on JSpIRIT as it seemed to provide the more reasonable results, while maintaining a fairly strict approach to system design.

JSpIRIT detected a total of 44 design flaws, distributed as follows:

- *26 Feature Envy*
- *9 Dispersed Coupling*
- *4 Intensive Coupling*
- *2 Shotgun Surgery*
- *2 Refused Parent Bequest*
- *1 God Class*

Given that the tool orders the smells by a "relevance" ranking (i.e. how impactful they might be in the evolution and maintainability of the program), we focused our attention on the ten most relevant smells, as they can give us a more accurate overview of eventual flaws at a higher level.

From this top ten, 60% of the flaws detected were *Feature Envious* methods. *Dispersed* and *Intensive Coupling* also make an appearance, holding 20% and 10% respectively. The remaining 10% belongs to a situation of *Refused Parent Bequest*, in fact marked as the most relevant code smell in the Bootique system, peaking at number one in the classification.

Bearing in mind these tools are very demanding, and after carefully analyzing these, we came to the conclusion that the system is generally well built/designed. As with most things, some corrections could be made to improve the overall design and maintainability of the project. However, we can only "guess" some of the decisions made in the project, as we are outsiders to its development, and so it would be unreasonable to point out major architectural changes. Following this note - and considering the overall quality of Bootique - we decided to point out some of the good design patterns that we were able to detect.

4.5.1 Facade Design Pattern

In the Bootique class, we detected a great usage of the *Facade* design pattern. In this specific case the construction of the object can be made enabling an “AutoLoadModules” feature, meaning that the calling function doesn’t need to specify which modules it requires to run the app. All the modules that the app needs are hidden from the client request. This way we isolate the client from all the modules components and reduce its coupling with the module subsystem.

Note that one can usually correlate the usage of the *Facade* pattern with a higher-than-usual usage of foreign data (the modules, in this specific case) and a higher functional complexity, due to the added complexity of managing the subsystems. These two factors (that are key to identify a *God Class*), seem to corroborate the fact that JSpIRIT considers this class to be a *God Class*.

4.5.2 Builder

The developers of Bootique make some use of the builder pattern throughout the project (e.g. JsonNodeConfigurationBuilder, CommandManagerBuilder). This pattern is, among other things, a good choice to avoid falling into the Telescoping Constructor (anti) pattern, as mentioned in Effective Java, 2nd Edition [3]. It also enhances readability and creational flexibility.

4.5.3 Decorator

On the structural side of things, we must highlight the proper usage of the *Decorator* pattern (not to be confused with the *Adapter* pattern). Bootique provides a command decorator API; one starts out by creating a decorator policy (with CommandDecorator) and follows up by "decorating" the main command with said policy.[19]

In practice, this pattern enhances object functionality without the need to change their interfaces. Furthermore, it does so in a flexible way, allowing the addition/removal of behaviors at runtime.[20]

4.5.4 Fluent Interfaces

One interesting thing that's worth mentioning is the presence of what Evans and Fowler coined as *Fluent Interface* (though we've also seen the term *Semantic Facade* being used).[21] Often implemented with *Method Chaining*, *Fluent Interfaces* strive to make highly readable APIs, essentially creating *Domain-Specific Languages* within the interface.

```
contributeCommands()
  .addBinding()
  .to(commandType)
  .in(Singleton.class);
```

Figure 7 – Fluent Interface Design

(BQCoreModuleExtender#addCommand)

4.6 SonarLint

When employing a linter at a more advanced phase of a project (which is the case here), the stylistic problems flagged will tend to be highly opinionated, and therefore should not be taken as absolute truths. That being said, adopting a linter (or maybe a formatter) at the beginning of development is a great idea. They can usually be

[18] “Software versioning,” Wikipedia. 18-Oct-2018.

[19] “Bootique: Minimally Opinionated Framework for Runnable Java.” [Online]. Available: https://bootique.io/docs/0/bootique-docs/#_chapter_10_commands. [Accessed: 20-Oct-2018].

[20] “Decorator.” [Online]. Available: https://refactoring.guru/design-patterns/decorator. [Accessed: 20-Oct-2018].

[21] “bliki: FluentInterface,” martinfowler.com. [Online]. Available: https://martinfowler.com/bliki/FluentInterface.html. [Accessed: 20-Oct-2018].

tweaked to suit each team's preferences, enforcing a common, homogeneous code style.

SonarLint detected a wide variety of code smells and stylistic issues - 192 to be exact – from which we highlight the most prominent.

4.6.1 Cognitive Complexity

Cognitive Complexity was introduced by SonarSource back in 2016 as a way to measure how difficult it is to understand the control flow of a method. It distantiates itself from Cyclomatic Complexity by bringing a new light on measuring maintainability.

While Cyclomatic Complexity does a great job in covering testability, it comes short in reflecting how hard it is to understand a given method - and therefore maintain it.

"It starts from the precedents set by Cyclomatic Complexity but uses human judgement to assess how structures should be counted". [22]

To further understand the concept, we suggest the reading of "Cognitive Complexity, A new way of measuring understandability" by SonarSource. [23]

The method ConsoleAppender#foldToLines presented a Cognitive Complexity of 26, which is way higher than the recommended threshold of 15, as defined in the SonarLint tool. While analyzing this method, we can immediately see why the cognitive complexity is so high: the nesting goes four levels deep, which dramatically increases the overall (Cognitive Complexity) score of the method.

We suggest using *Extract Method* in some of the method's blocks to improve understandability. These code blocks are heavily commented, and their usage matches the *Comments* code smell, considering they're clearly being used as a "deodorant". Augmented with proper naming, the previously stated *Extract Method* strategy would be a viable solution.

4.6.2 Variable Shadowing

Variable Shadowing occurs when we declare a variable in a given scope, with a name that is already being used by another variable in some outer scope.[24]

An example can be seen at Bootique.java#createJVMShutdownHook, where the declaration of the local variable shutdownManager "hides" the upper definition of the global variable, making it harder to understand which one is being used. This practice reduces readability and can be avoided by simply changing the variable name, usually in the inner-most scope.

4.6.3 Switch Statements

```
switch (o.getOption().getValueCardinality()) {
  case REQUIRED:
    parts.add(" ");
    parts.add(valueName);
    break;
  case OPTIONAL:
    parts.add(" [");
    parts.add(valueName);
    parts.add("]");
    break;
}
```

Figure 8 Unsafe Switch

(DefaultHelpGenerator.java#printOptions)

This method contains a switch statement in which the cases match an Enum. Yet, not all of the Enum constants are in the switch; this could potentially not be a problem, if the remaining constants were caught by the default case. But there is no default switch case, which makes this a potentially dangerous piece of code, by defying a basic rule of defensive programming.[25] [26] The obvious solution would be to add a default case and deal with it accordingly.

4.6.4 Commented-Out code Blocks

This smell was detected in many classes and methods.

Commented-out blocks of code create visual noise and reduce readability. Nowadays, with the use of version control tools, there is no longer a justification to keep a codebase cluttered with dead commented code.

4.6.4.1 Unnamed Exceptions

The use of generic exceptions prevents calling methods from handling true, application-generated exceptions differently than the default java-built in exceptions. Custom exceptions should be created to facilitate the detection and handling of the generated errors.

5. JHotDraw

5.1 inFusion

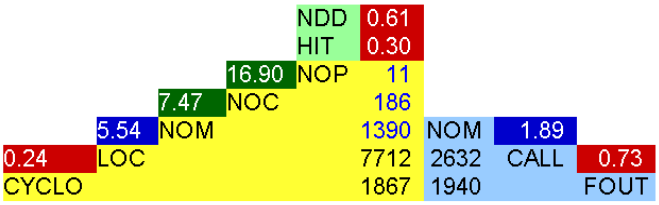

Figure 9 – JHotDraw Overview Pyramid created with inFusion

Like we did for Bootique, we used inFusion to generate an Overview Pyramid of JHotDraw.

[22] "Cognitive Complexity™ | SonarSource." [Online]. Available: https://www.sonarsource.com/resources/white-papers/cognitive-complexity.html. [Accessed: 20-Oct-2018].

[23] ibid

[24] "Variable Shadowing and Hiding in Java - DZone Java," dzone.com. [Online]. Available: https://dzone.com/articles/variable-shadowing-and-hiding-in-java. [Accessed: 20-Oct-2018].

[25] Diego, "The Art of Defensive Programming," WengVox, 25-Dec-2016.

[26] "Defensive programming | TheCodingMachine Best practices." [Online]. Available: http://bestpractices.thecodingmachine.com/php/defensive_programming.html. [Accessed: 20-Oct-2018].

Its classes are organized within a reasonable number of packages. However, each class has, on average, eight to nine methods, which is considered high according to the threshold proposed by Lanza et al [1]. The methods themselves are small yet complex (i.e. have many conditional branches).

Furthermore, despite having low intensity, they have high dispersion. In other words, the few methods that they tend to call are usually from many other classes. There's also a lot of inheritance going on. The inheritance trees tend to be very deep, and the base-classes have several directly derived sub-classes. This implies a very tightly-coupled architecture, which is often harder to update, test and maintain. Something to keep in mind.

inFusion also detected the presence of multiple disharmonies, but to avoid repetition, we will only delve into the ones not yet covered.

ShortestDistanceConnector#findPoint was marked as a *Brain Method* - and rightfully so, considering its use of many variables and clear violation of the Proportion Rule (with roughly 100 LoC). The multiple comments that precede blocks of code give clear hints that *Extract Method* should be used to refactor the method.

Moreover, UngroupCommand#execute was detected to suffer from *Intensive Coupling*. Checking with Lanza & Marinescu's [1] detection strategy, we confirmed that the method called too many methods from few unrelated classes and had few nested conditionals. The coupling dispersion is relatively low, with 67% of the method's calls being to methods of the same provider class, DrawingView. Thus, defining a new, more complex serving method in that provider class would replace the need for multiple calls.

5.2 Metrics

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per method)		1.479	1.147	13	jhotdraw5.2/src/CH/ifa/draw/util/Geom.java	intersect
Number of Parameters (avg/max per method)		1.028	1.122	8	jhotdraw5.2/src/CH/ifa/draw/util/Geom.java	intersect
Nested Block Depth (avg/max per method)		1.069	0.664	5	jhotdraw5.2/src/CH/ifa/draw/contrib/PolygonFigure.java	smoothPoints
Afferent Coupling (avg/max per packageFragment)		28.091	41.245	121	jhotdraw5.2/src/CH/ifa/draw/framework	
Efferent Coupling (avg/max per packageFragment)		11.909	14.368	47	jhotdraw5.2/src/CH/ifa/draw/standard	
Instability (avg/max per packageFragment)		0.578	0.366	1	jhotdraw5.2/src/CH/ifa/draw/samples/javadraw	
Abstractness (avg/max per packageFragment)		0.131	0.221	0.778	jhotdraw5.2/src/CH/ifa/draw/framework	
Normalized Distance (avg/max per packageFragment)		0.291	0.322	0.857	jhotdraw5.2/src/CH/ifa/draw/application	
Depth of Inheritance Tree (avg/max per type)		2.433	1.842	8	jhotdraw5.2/src/CH/ifa/draw/samples/javadraw/JavaDrawApp.java	
Weighted methods per Class (avg/max per type)	2186	12.784	16.172	108	jhotdraw5.2/src/CH/ifa/draw/standard/StandardDrawingView.java	
Number of Children (avg/max per type)	143	0.836	1.925	12	jhotdraw5.2/src/CH/ifa/draw/standard/LocatorHandle.java	
Number of Overridden Methods (avg/max per type)	188	1.099	1.628	12	jhotdraw5.2/src/CH/ifa/draw/standard/DecoratorFigure.java	
Lack of Cohesion of Methods (avg/max per type)		0.229	0.321	1.5	jhotdraw5.2/src/CH/ifa/draw/figures/BorderDecorator.java	
Number of Attributes (avg/max per type)	271	1.585	2.124	14	jhotdraw5.2/src/CH/ifa/draw/applet/DrawApplet.java	
Number of Static Attributes (avg/max per type)	92	0.538	1.05	7	jhotdraw5.2/src/CH/ifa/draw/application/DrawApplication.java	
Number of Methods (avg/max per type)	1414	8.269	9.898	61	jhotdraw5.2/src/CH/ifa/draw/standard/StandardDrawingView.java	
Number of Static Methods (avg/max per type)	64	0.374	1.693	15	jhotdraw5.2/src/CH/ifa/draw/util/Geom.java	
Specialization Index (avg/max per type)		0.483	0.64	3.111	jhotdraw5.2/src/CH/ifa/draw/standard/ToolButton.java	
Number of Classes (avg/max per packageFragment)	171	15.545	16.194	55	jhotdraw5.2/src/CH/ifa/draw/standard	
Number of Interfaces (avg/max per packageFragment)	23	2.091	3.988	14	jhotdraw5.2/src/CH/ifa/draw/framework	
Number of Packages	11					
Total Lines of Code	9420					
Method Lines of Code (avg/max per method)	5070	3.43	5.345	79	jhotdraw5.2/src/CH/ifa/draw/figures/ShortestDistanceConnector.java	findPoint

Figure 10 – Software Quality Metrics captured by the Metrics Plugin

5.2.1 *Cyclomatic Complexity*

The cyclomatic complexity in JHotDraw is generally good. The low value of 1.479 means that most methods are simple, with a single execution flow. Still, improvements can be made. The method with the maximum value of complexity (13) has the goal of returning the intersection point of two lines in space (each line being defined by four points). The algorithm implemented in the method, though functional, leaves a lot to be desired in terms of complexity, as it contains a lot of conditional statements and verifications.

Our suggestion for reducing the complexity would be to replace the implemented algorithm altogether with one that presents less control flow decision points.[27] The referenced code has basically the same functionality and follows a single execution flow. The only change needed would be to return null when no intersection is found which would only add one decision point and would bring the total complexity to a value of two.

5.2.2 *High Number of Parameters*

The number of parameters that JHotDraw's methods take in averages at 1.028. However, we can see that there's at least one method that takes in eight parameters - just one more than Bootique.

But this time, the method that took in the most parameters was not a constructor, but a "regular" method. More specifically, we're talking about a method that takes in eight parameters (the spatial coordinates of four points, in total) to calculate the intersection point of two lines.

Method Extraction - the pattern mentioned earlier, during the analysis of Bootique - could be a reasonable refactoring option. But this scenario screams for the *Parameter Object* pattern. By creating a Line object - essentially a wrapper for a couple of points in space - we'd simply need to pass two parameters (the two lines) instead of the original eight. This change would be straight-forward to implement, and we argue it would enhance readability and semantics.

5.2.3 *Nested Block Depth*

Nested Block Depth values are good on the overall scope of the program. Indeed, a mean value of 1.069 with a standard deviation of 0.664 emphasizes that. The biggest block depth of the entire program comes with a value of five. This value is on the higher end of the spectrum, but it's reasonable, given the context of the problem the method aims to solve.

5.2.4 *Classes and Methods*

The biggest package in JHotDraw contains roughly 32% - almost a third! - of the entire system's classes. In contrast, the smallest four packages hold only around 4%. Furthermore, inFusion also detected the presence of the Shotgun Surgery[28] code smell in nine methods. Poor separation of concerns is often the root of this code smell, and while we couldn't identify which particular methods had this problem, they will eventually harm maintainability further down the road.

5.2.5 *Coupling Intensity (Calls / NoM)*

Though JHotDraw doubles the coupling intensity of Bootique, it still maintains a low coupling intensity (2.17), which means its methods call only close to two other methods, on average.

5.2.6 *Coupling Dispersion (Fanout / Calls)*

The JHotDraw system presents a high value of coupling dispersion (0.67). In other words, two out of three operation calls are made to other classes.

5.3 SourceMiner

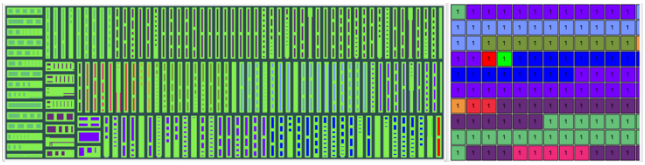

Figure 11 - JHotDraw SourceMiner Analysis regarding McCabe Complexity (Classes)

[27] "Find the intersection of two lines - Rosetta Code." [Online]. Available: https://rosettacode.org/wiki/Find_the_intersection_of_two_lines#Java. [Accessed: 01-Oct-2018].

[28] "The Shotgun Surgery Code Smell - DZone Java," dzone.com. [Online]. Available: https://dzone.com/articles/code-smell-shotgun-surgery [Accessed: 01-Oct-2018].

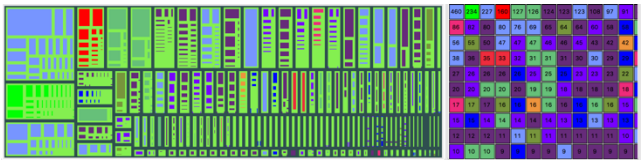

Figure 12 - JHotDraw SourceMiner Analysis regarding Coupling (Classes)

5.4 Sonar Lint

5.4.1 Collapsible If Statements

```
if (fAttributes != null) {
  if (fAttributes.hasDefined(name)) {
    ...
  }
}
```

Figure 13 – Collapsible If Statements

(AttributeFigure.java#getAttribute)

This method clearly presents collapsible if statements. Collapsing said if statements would improve code readability by reducing the adjacent nesting.

5.4.2 Duplicated Code

AbstractFigure#figures and AbstractFigure#decompose

These methods have the exact same implementation, and both are used in different scenarios. Duplicated code is terrible when it comes to maintainability. To solve this issue, one of the methods could be removed and a renaming could be considered (one that would comply with all of its usage scenarios).

5.4.3 Resource Not Properly Closed

```
public String store(...) throws IOException {
  FileOutputStream stream = ...
  ObjectOutput output = ...
  ...
  output.close();
  return adjustFileName(...);
}
```

Figure 14 - Resource not properly closed

(SerializationStorageFormat.java#store)

We detected a possibly application-breaking bug related to the improper closing of resources (ObjectOutput, in this case).

The ObjectOutput class implements the AutoClosable interface[29], and therefore the resource should have been created using a "try-catch" block. Why? Because the object would be closed automatically if it were to fall in the catch statement. The way it is done right now - throwing an exception - implies that it would be handled by the caller (or somewhere up, for that matter), but the resource would remain open, possibly leading to leaks.

6. CONCLUSION

Despite being similar in size, the two systems analyzed are fundamentally different when it comes to architecture. Bootique's class packaging is much more fine-grained than the one of JHotDraw. In fact, the former's fine-grained approach to organization and separation of concerns goes all the way down to its methods - remarkably simple and with little complexity. JHotDraw's methods, on the other hand, tend to have somewhat high levels of complexity, which could difficult further development and maintainability.

Another key difference between the two is their use of inheritance. While JHotDraw makes heavy use of inheritance mechanisms (with large breadth and depth), Bootique seems to prefer composition. This leads to a type of architecture that is usually much looser and, therefore, easier to test, update and maintain.

Bootique employs a great deal of battle-tested design patterns and doesn't seem to suffer from major code smells. JHotDraw uses some similar patterns, focusing heavily on the *Observer* pattern for its inter-procedural communication. Yet it suffers from more (and more varied) code smells, making it potentially more difficult to evolve in the future, in comparison to Bootique. We'd like to reinforce, though, that sometimes design decisions that may seem "odd" and/or lead to apparent smells have not-so-obvious purposes that are unknown to the external observer. On this matter, practice, experience and some degree of familiarity make perfect.

Last but not least, we need to point out how JHotDraw provides a great deal of in-code documentation, in contrast to Bootique.

There is always room for improvements and refactoring in a software project, and neither Bootique nor JHotDraw are exceptions. In terms of software quality, we consider both systems to be on the higher end of the spectrum, ranking Bootique slightly above.

REFERENCES

[1] Lanza, M. and Marinescu, R. "Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems". Springer Science & Business Media, 2007.

[2] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1 edition. Upper Saddle River, NJ: Prentice Hall, 2008.

[3] J. Bloch, Effective Java, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2008.

[4] T. G. S. Filó, M. A. S. Bigonha, and K. A. M. Ferreira, "A Catalogue of Thresholds for Object-Oriented Software Metrics," p. 3, 2015.

[29] "AutoCloseable (Java Platform SE 7)." [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html. [Accessed: 20-Oct-2018].