



# POLYTECHNIQUE MONTRÉAL

## LOG8430 - Software Architecture and Advanced Design

### Travail Pratique 2 (TP2)

Professor: Zohreh Sharafi

Group Name: Atwater

Team members:

Student ID	Name
2313663	Xi-Zhen Wang
2078987	Thomas Lusignan
2317239	Roberto Molina
2227105	Mohamad Hadi Ajami

Montréal, Quebec

2023

# Table of Content

<b>1. Abstract.....</b>	<b>2</b>
<b>2. Introduction.....</b>	<b>4</b>
<b>3. Pixel-Dungeon Design Quality Measurement.....</b>	<b>5</b>
3.1 Classic metrics.....	5
3.1.1 Size Metrics.....	5
3.1.2 Interface complexity metrics.....	6
3.1.3 Structural complexity metrics.....	7
3.1.4 Inheritance metrics.....	8
3.1.5 Coupling metrics.....	9
3.1.6 Cohesion metrics.....	9
3.1.7 Documentation metrics.....	10
3.2 Quality Criteria for QMOOD.....	11
3.2.1 Method for measuring QMOOD criteria.....	11
3.2.2 QMOOD criteria.....	15
3.3 Correlation of metric values.....	16
3.3.1 Understandability.....	16
3.3.2 Reusability.....	17
3.3.3 Functionality.....	18
<b>4. Pixel-Dungeon Anomalies.....</b>	<b>19</b>
4.1 Definition of metric thresholds.....	19
4.1.1 Long Method.....	20
4.1.2 Large Class.....	21
4.1.3 Duplicate Code.....	21
4.1.4 Shotgun Surgery.....	21
4.1.5 Feature Envy.....	22
4.2 Specific Anti-Patterns in Pixel Dungeon.....	23
4.2.1 Long Method.....	23
4.2.2 Large Class.....	23
4.2.3 Duplicate Code.....	24
4.2.4 Shotgun Surgery.....	25
4.2.5 Feature Envy.....	26
<b>5. Pixel-Dungeon Anomaly Correction.....</b>	<b>27</b>
5.1 Anomaly Correction by Refactoring.....	27
5.1.1 Long Method.....	27
5.1.2 Large Class.....	28
5.1.3 Duplicate Code.....	29
5.1.4 Shotgun Surgery.....	30
5.1.5 Feature Envy.....	32
5.2 Metrics after Refactoring.....	33
5.2.1 Long Method.....	33
5.2.2 Large Class.....	33
5.2.3 Duplicate Code.....	34
5.2.4 Shotgun Surgery.....	34
5.2.5 Feature Envy.....	35
5.3 QMOOD after Refactoring.....	36
<b>6. Conclusion.....</b>	<b>38</b>

# 1. Abstract

This assignment focuses on evaluating and enhancing the design quality of the open-source game, Pixel Dungeon. We employ a variety of software metrics to measure cohesion, complexity, coupling, and size at different levels of the project, from packages to individual classes. Furthermore, we analyze the system's adherence to ISO 9126 quality criteria, including functionality, reusability, understandability, flexibility, efficiency, and extensibility. The goal is to identify anomalies in the codebase, correlate them with the metrics, and propose refactoring solutions to improve the overall system quality.

## 2. Introduction

In this assignment, we turn our attention to the open-source project Pixel Dungeon, a popular roguelike game with a significant codebase. Our objective is to comprehensively assess the design quality of this project, identify anomalies, and propose refactoring strategies to enhance its overall quality.

To achieve this, we follow a structured approach. First, we gather a plethora of metrics, including measures of cohesion, complexity, coupling, and size, at various levels of the software project. By presenting these metrics and calculating averages, maximums, and standard deviations, we aim to quantify the current state of Pixel Dungeon's design quality.

Our assessment doesn't stop at traditional software metrics. We also delve into ISO 9126 quality criteria. This holistic approach allows us to understand the software's quality from different perspectives.

Following the initial assessment, we then move to identify anomalies in the codebase. These anomalies, or "code smells," are deviations from best practices that can impede maintainability, readability, and overall system quality. To do this, we define metric thresholds based on authoritative sources and then apply our judgment to identify where Pixel Dungeon falls short.

Having identified these anomalies, we propose refactoring solutions. We present the code before and after refactoring. Furthermore, we assess the impact of refactoring on individual metrics. The goal is to confirm that our refactoring efforts have improved the design quality. Finally, we evaluate whether these refactoring efforts result in improvements in terms of the ISO 9126 quality criteria.

In this assignment, we bring together a comprehensive analysis of Pixel Dungeon's design quality, highlight code anomalies, and provide practical refactoring solutions that align with recognized software engineering principles.

## 3. Pixel-Dungeon Design Quality Measurement

### 3.1 Classic metrics

In this section we are gonna observe the classic metrics. These metrics have been observed using the IDE IntelliJ with the plugins MetricTree and MetricReloaded.

Each metric has been measured in the appropriate level. The different levels are: project, package, class and method. For each metric, we will provide level, average, minimum value, maximum value, standard deviation and total value.

#### 3.1.1 Size Metrics

**LOC:** It counts the number of return characters (e.g.,) in a method, a class, a compilation unit.

To assess program size, we've examined lines of code at various levels: project, class, and method. For the project level, we've determined that the project contains 49,460 lines of code, indicating a substantial program size for Pixel Dungeon.

When inspecting class-level lines of code, we find that the average is 54.26, which is relatively modest for a class. However, there's a class with an unusually high count of 956 lines, suggesting a violation of the single responsibility principle and the potential presence of a long class antipattern.

A similar pattern is observed for lines per method, where the average stands at 9.51, a common value. But the maximum value is 187, indicating the possible existence of a large method antipattern.

Table 1: Size metrics

Metric	Level	Avg. value	Min. value	Max. value	Std. Dev.	Total
<b>LOC (Lines of code)</b>	Project	N/A	N/A	N/A	N/A	49460
<b>CLOC (Class lines of code)</b>	Class	54.26	2	956	77.72	36031
<b>MLOC (Method lines of code)</b>	Method	9.51	1	187	11.79	29768

### 3.1.2 Interface complexity metrics

**NOM:** It counts the number of methods declared in a class, not the derived methods.

**SIZE2:** It counts the number of attributes and methods of a class.

Analyzing Table 2, we can see that at the class level, the count of methods suggests the potential presence of a large class antipattern. This inference is drawn from the exceptionally high maximum value of 59, significantly exceeding the norm for this metric. Similarly, when examining SIZE2, we reach a similar conclusion as the maximum value of 168 deviates substantially from the average, indicating a potential design issue.

Table 2: Complexity of Interface Metrics

Metric	Level	Avg. value	Min. value	Max. value	Std. Dev.	Total
<b>NOM (Number of methods)</b>	Class	3.85	0	59	26.7	3354
<b>SIZE2 (Number of attributes and methods)</b>	Class	41.85	0	168	57.07	36351

### 3.1.3 Structural complexity metrics

**CC** (cyclomatic complexity): It counts the number of independent paths in a slice of code.

**RFC** (response for class): It counts the number of public methods of a class and all methods accessed directly by these public methods.

**WMC** (Weighted method count): It counts the sum of a metric over all methods of a class. The metric could be the complexity, the LOC or none (unweighted)

Upon examining the outcomes of these metrics in Table 3, several observations can be made:

The average Cyclomatic Complexity (CC) is 2.42, signifying that the code is generally easy to comprehend. However, the presence of an exceptionally high maximum value of 43 suggests the potential existence of a large method, introducing complexity and making it challenging to read.

When assessing RFC values, the average reflects a typical number of public methods in classes, indicating a reasonable design. Nonetheless, the existence of a maximum value of 198 implies the presence of a class with several public methods, potentially indicative of anti-patterns such as "shotgun surgery" or a "god class."

In the case of WMC, it demonstrates an overall favorable average value, suggesting that classes tend to have a reasonable number of independent paths. Yet, the presence of an extremely high maximum value underscores the possibility of classes exhibiting anti-patterns.

Table 3: Structural Complexity Metrics

Metric	Level	Avg. value	Min. value	Max. value	Std. Dev.	Total
<b>CC (Cyclomatic complexity)</b>	Method	2.42	1	43	3.33	7575
<b>RFC (Response for a class)</b>	Class	11.76	0	198	13.49	6982
<b>WMC (Weighted Class Methods)</b>	Class	10.59	0	243	19.24	7036

### 3.1.4 Inheritance metrics

**DIT** (Depth of Inheritance): It calculates the maximal length from a derived class to a basic class in the inheritance structure of the system.

**NOC** (Number of children): It counts the number of classes immediately derived from a basic class.

Observing at the values in Table 4, we can observe that the average DIT and NOC values generally fall within reasonable ranges, which is a positive sign for code maintainability.

The presence of a class with a DIT of 5 indicates that there are some classes with relatively deep inheritance, but this may not necessarily be a problem if managed well.

The presence of a class with 43 child classes (NOC) is noteworthy and may warrant further investigation, as it could indicate complex class hierarchies or a violation of design principles.

In conclusion, while the average values for DIT and NOC appear reasonable, it is essential to investigate the classes with extreme values to ensure that they do not introduce design problems or anti-patterns into the codebase.

Table 4: Inheritance Metrics

Metric	Level	Avg. value	Min. value	Max. value	Std. Dev.	Total
<i><b>DIT (Depth of Inheritance)</b></i>	Class	1.49	0	5	1.36	949
<i><b>NOC (Number of children)</b></i>	Class	0.83	0	43	3.77	532

### 3.1.5 Coupling metrics

**Ca** (Afferent Coupling): It counts the number of external classes that are coupled (as per the definition of CBO) with classes from another package with respect to incoming coupling (i.e., external classes use methods of internal classes)



**CBO** (Efferent Coupling (Ce)): It counts the number of classes that are “coupled” with another class.

For the Ca metric, the wide range from 11 to 3101 is noteworthy. The presence of a package with 3101 afferent couplings indicates that it's heavily used and has a substantial impact on other parts of the codebase.

For the CBO metric, the presence of a class with a CBO of 293 implies that there is a class with a high level of coupling to other classes, which could affect its maintainability and understandability.

In both cases, the high standard deviations indicate that there is significant variation in these metrics, suggesting the need for a closer examination of the packages with high Ca and the classes with high CBO. Reducing such dependencies and ensuring proper design principles are followed may lead to improved code maintainability and readability.

Table 5: Coupling Metrics

Metric	Level	Avg. value	Min. value	Max. value	Std. Dev.	Total
<b>Ca (Afferent coupling)</b>	Package	451.29	11	3101	529.06	16919
<b>CBO (Coupling between objects)</b>	Class	14.78	1	293	26.10	9817

### 3.1.6 Cohesion metrics

**LCOM** (Lack of cohesion of methods): It measures the degree of relatedness or coupling between methods within a class.

The LCOM (Lack of Cohesion of Methods) metric at the class level provides valuable insights into code cohesion. With an average LCOM of 2.23, it appears that, on the whole, classes exhibit moderate cohesion, indicating that methods within classes tend to be relatively cohesive. However, the presence of a substantial standard deviation at 1.89 highlights considerable variability in cohesion among classes. Of particular concern is the class with the maximum LCOM value of 10, which signals a severe lack of cohesion and raises the possibility of the "God Class" anti-pattern.

High LCOM values within classes can introduce complexity, hinder maintainability, and suggest a violation of the Single Responsibility Principle, necessitating careful review and potential refactoring to improve code quality and adherence to design principles.

Table 6: Cohesion Metrics

Metric	Level	Avg. value	Min. value	Max. value	Std. Dev.	Total
<b><i>LCOM (Lack of cohesion of methods)</i></b>	Class	2.23	0	10	1.89	1483

### 3.1.7 Documentation metrics

**CRAT** (Comment Ratio): Calculates the ratio of lines of comment code to total lines of code for the project. Lines of whitespace are not counted.

Table 7: Documentation Metrics

Metric	Level	Avg. value	Min. value	Max. value	Std. Dev.	Total
<b><i>CRAT (Comment Ratio)</i></b>	Package	17.59%	5.34%	43.60%	9.91%	N/A

Observing the data from Table 7 we can say that an average CRAT of 17.59% suggests a moderate level of documentation, which is beneficial for code maintainability. However, it's essential to ensure that the level of comments remains balanced and that exceptionally high or low CRAT values are reviewed to maintain code quality and readability.

## 3.2 Quality Criteria for QMOOD

In this section, we will discuss how to use the QMOOD criteria to evaluate the system metrics. We obtained the values of the QMOOD criteria using MetricsTree/MetricsReloaded in IntelliJ Idea.

### 3.2.1 Method for measuring QMOOD criteria

The Quality Model for Object-Oriented Design (QMOOD) assessment is an evolution of the ISO 9126 standard for software quality. To address the issues with the previous quality model, which offered abstract definitions of lower-level details, QMOOD offers a solution by providing a well-defined and computable approach.

QMOOD model comprises six quality attributes, as presented in Figure 1. This set of attributes possesses a sufficient breadth to identify desirable characteristics within object-oriented systems. However, it remains an abstract concept and lacks direct observability.

Quality Attribute	Definition
Reusability	Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort.
Flexibility	Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functionally related capabilities.
Understandability	The properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure.
Functionality	The responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.
Extendibility	Refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.
Effectiveness	This refers to a design's ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques.

Figure 1: Quality Attribute Definitions

QMOOD defines a collection of design properties, as shown in Figure 2, that represent tangible and directly measurable concepts. These properties can be evaluated through a comprehensive examination of the internal and external structure, relationship, and functionality of the design components, encompassing attributes, methods, and classes within an object-oriented framework.

Design Property	Definition
Design Size	A measure of the number of classes used in a design.
Hierarchies	Hierarchies are used to represent different generalization-specialization concepts in a design. It is a count of the number of non-inherited classes that have children in a design.
Abstraction	A measure of the generalization-specialization aspect of the design. Classes in a design which have one or more descendants exhibit this property of abstraction.
Encapsulation	Defined as the enclosing of data and behavior within a single construct. In object-oriented designs the property specifically refers to designing classes that prevent access to attribute declarations by defining them to be private, thus protecting the internal representation of the objects.
Coupling	Defines the interdependency of an object on other objects in a design. It is a measure of the number of other objects that would have to be accessed by an object in order for that object to function correctly.
Cohesion	Assesses the relatedness of methods and attributes in a class. Strong overlap in the method parameters and attribute types is an indication of strong cohesion.
Composition	Measures the “part-of,” “has,” “consists-of,” or “part-whole” relationships, which are aggregation relationships in an object-oriented design.
Inheritance	A measure of the “is-a” relationship between classes. This relationship is related to the level of nesting of classes in an inheritance hierarchy.
Polymorphism	The ability to substitute objects whose interfaces match for one another at run-time. It is a measure of services that are dynamically determined at run-time in an object.
Messaging	A count of the number of public methods that are available as services to other classes. This is a measure of the services that a class provides.
Complexity	A measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.

Figure 2: Design Property Definitions

Every design property within the QMOOD model represents a well-defined attribute of a design and is assessed through the specified design metrics. The proposed design metrics, along with detailed descriptions, are provided in Figure 3, while the corresponding design properties are shown in Figure 4.

METRIC	NAME	DESCRIPTION
DSC	Design Size in Classes	This metric is a count of the total number of classes in the design.
NOH	Number of Hierarchies	This metric is a count of the number of class hierarchies in the design.
ANA	Average Number of Ancestors	This metric value signifies the average number of classes from which a class inherits information. It is computed by determining the number of classes along all paths from the “root” class(es) to all classes in an inheritance structure.
DAM	Data Access Metric	This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class. A high value for DAM is desired. (Range 0 to 1)
DCC	Direct Class Coupling	This metric is a count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
CAM	Cohesion Among Methods of Class	This metric computes the relatedness among methods of a class based upon the parameter list of the methods [3]. The metric is computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class. A metric value close to 1.0 is preferred. (Range 0 to 1)
MOA	Measure of Aggregation	This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of data declarations whose types are user defined classes.
MFA	Measure of Functional Abstraction	This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. (Range 0 to 1)
NOP	Number of Polymorphic Methods	This metric is a count of the methods that can exhibit polymorphic behavior. Such methods in C++ are marked as virtual.
CIS	Class Interface Size	This metric is a count of the number of public methods in a class
NOM	Number of Methods	This metric is a count of all the methods defined in a class.

Figure 3: Design Metrics Descriptions

Design Property	Derived Design Metric
Design Size	Design Size in Classes (DSC)
Hierarchies	Number of Hierarchies (NOH)
Abstraction	Average Number of Ancestors (ANA)
Encapsulation	Data Access Metric (DAM)
Coupling	Direct Class Coupling (DCC)
Cohesion	Cohesion Among Methods in Class (CAM)
Composition	Measure of Aggregation (MOA)
Inheritance	Measure of Functional Abstraction (MFA)
Polymorphism	Number of Polymorphic Methods (NOP)
Messaging	Class Interface Size (CIS)
Complexity	Number of Methods (NOM)

Figure 4: Design Metrics for Design Properties

Figure 5 illustrates the weighting of linkages from design properties to quality attributes. With an understanding of the definitions of each design attribute, property, and metric, these computational equations can be applied to precisely calculate values that enable the assessment of system quality.

Quality Attribute	Index Computation Equation
Reusability	$-0.25 * \text{Coupling} + 0.25 * \text{Cohesion} + 0.5 * \text{Messaging} + 0.5 * \text{Design Size}$
Flexibility	$0.25 * \text{Encapsulation} - 0.25 * \text{Coupling} + 0.5 * \text{Composition} + 0.5 * \text{Polymorphism}$
Understandability	$-0.33 * \text{Abstraction} + 0.33 * \text{Encapsulation} - 0.33 * \text{Coupling} + 0.33 * \text{Cohesion} - 0.33 * \text{Polymorphism} - 0.33 * \text{Complexity} - 0.33 * \text{Design Size}$
Functionality	$0.12 * \text{Cohesion} + 0.22 * \text{Polymorphism} + 0.22 * \text{Messaging} + 0.22 * \text{Design Size} + 0.22 * \text{Hierarchies}$
Extendibility	$0.5 * \text{Abstraction} - 0.5 * \text{Coupling} + 0.5 * \text{Inheritance} + 0.5 * \text{Polymorphism}$
Effectiveness	$0.2 * \text{Abstraction} + 0.2 * \text{Encapsulation} + 0.2 * \text{Composition} + 0.2 * \text{Inheritance} + 0.2 * \text{Polymorphism}$

Figure 5: Computation Formulas for Quality Attribute

### 3.2.2 QMOOD criteria

The QMOOD quality attributes of this project can be divided in four categories: Negative, Medium, High and Very High. By comparing the values to those obtained for the Microsoft Foundation Classes (MFC) and Object Windows Library (OWL) described in Bansiya & Davis, 2002 which contain many values between 1 and 3, we concluded that no values in our sample fit in a “Low” category.

Using these categories, the Understandability quality is Negative, the Effectiveness and Extendibility qualities are Medium, the Flexibility and Functionality qualities are High and the Reusability quality is Very High.

Table 8: Computation Formulas for Quality Attribute

Quality Attribute	Value	Qualitative category
Effectiveness	3.1704	Medium
Extendibility	3.0313	Medium
Flexibility	5.1520	High
Functionality	4.9456	High
Reusability	5.4036	Very High
Understandability	-9.3767	Negative

### 3.3 Correlation of metric values

Bansiya & Davis, 2002 define the QMOOD Quality Attributes as a sum of weighted metrics. For each discussed quality, we will remind the reader of this formula, and point to one or more metrics that seems to have contributed to the obtained score.

#### 3.3.1 Understandability

*Quality Attribute value: -9.3767*

*Formula:*

$$\begin{aligned} & - 0.33 * \text{Abstraction} \quad + 0.33 * \text{Encapsulation} \quad - 0.33 * \text{Coupling} \\ & + 0.33 * \text{Cohesion} \quad - 0.33 * \text{Polymorphism} \quad - 0.33 * \text{Complexity} \\ & - 0.33 * \text{Design Size} \end{aligned}$$

This is the only QMOOD quality that is negative. This implies that the project has high scores in the negatively multiplied design properties like abstraction, coupling, complexity or design size.

Bansiya & Davis associate complexity with the Number Of Methods (NOM) metric, which we measured. With a value of 3.85, this metric might have made the Understandability value slightly lower.

Moreover, abstraction is linked to the Average Number of Ancestors (ANA) metric. This is similar to the DIT metric we measured, which has an average of 1,49, a fairly low number. It would therefore be driven more by other metrics, like Design Size.

The Design Size in Classes (DSC), the metric associated with the Design Size design property, is the number of classes in the project. Though not included in our tables, it stands at 840 for the project, which is a large number that probably affected the quality attribute value.



The Coupling design property is associated with the Direct Class Coupling metric, which is similar to the CBO we measured. With an average value of 14.78, this could have helped drag the quality in the negatives.

### 3.3.2 Reusability

*Quality Attribute value:* **5.4036**

*Formula:*

$$\begin{aligned} & - 0.25 * \text{Coupling} \quad + 0.25 * \text{Cohesion} \quad + 0.5 * \text{Messaging} \\ & + 0.5 * \text{Design Size} \end{aligned}$$

Reusability is the quality attribute that scored the highest of the QMOOD metrics. It was the only one in our “very high” category. Let’s look at a few of the metrics that influenced this score.

We already mentioned how Coupling can be linked to CBO, which has a value of 14.78. This attribute being multiplied by -0,25 means that the other metrics must have been much larger to compensate.

For instance, the Design Size, with its DSC metric of 840, may very well have more than counterbalanced the effect of the Coupling property. Considering its multiplier is high, at 0,5, this makes even more sense.

The other metrics used by Bansiya & Davis to judge this quality attribute are the Cohesion Among Methods in Class (CAM), and the Class Interface Size (CIS). The CAM is associated with cohesion, and the CIS, with messaging.

Though we didn’t include CAM in our choice of metrics, we did include LCOM, which is approximately the inverse of CAM. With an average value of 2.23, the correlation with Reusability is quite low.

Finally, the metric we did include which is closest to CIS is RFC. The average value of RFC is relatively high at 11.76. Along with the 0.5 multiplier in the formula for messaging, this is also correlated with the high Reusability value.

### 3.3.3 Functionality

*Quality Attribute value:* **4.9456**

*Formula:*

$$0.12 * \text{Cohesion} + 0.22 * \text{Polymorphism} + 0.22 * \text{Messaging} \\ + 0.22 * \text{Design Size} + 0.22 * \text{Hierarchies}$$

The last quality attribute we will discuss is Functionality. According to Bansiya & Davis, it is defined by five design properties, including many we have already discussed: Cohesion, Messaging, and Design Size. The following discusses how the metrics most closely related to these design properties correlate with the value of 4.9456 of the Functionality of Pixel Dungeon, which we categorized as “high”.

Once again, the highest correlation is in the Design Size design property. With 820 classes, the DSC must have played a role in the final score for functionality.

Cohesion is a different story. With a lower multiplier of 0.12, and a middling average LCOM value, it is hard to see a correlation with the value of the quality attribute.

Finally, messaging’s most closely associated metric is RFC, which also has a relatively high value correlated with the Quality Attribute value.

## 4. Pixel-Dungeon Anomalies

### 4.1 Definition of metric thresholds

One appropriate method to define metric thresholds is to use statistics. A metric that is more than one standard deviation above or below the average value for that metric would cross the anomaly threshold.

For instance, if a metric computed on classes has an average of  $A$  and a standard deviation of  $S$ , any class whose metric is higher than  $A + S$  would be above the “High” threshold, and if it is higher than  $A + 1.5S$ , it would cross the “Very High” threshold. The metrics reported here have a right-skewed distribution since their minimum value is zero. Thus, trying to compute a “Low” threshold using  $A - S$  would result in many impossible to attain negative thresholds. We therefore decided to not include “Low” threshold values.

Table 9. General threshold definitions

Name	Constant
<i>NONE</i>	0
<i>SEVERAL</i>	2
<i>FEW</i>	4
<i>Many</i>	> 7

The thresholds are defined in Tables 9 and 10, except for the metrics that were calculated on the whole project: since these had a single value, their average and standard deviation could not be computed.

Table 10. Statistical threshold definitions

<b>Metric</b>	<b>Avg. value</b>	<b>Std. Dev.</b>	<b>Low Threshold</b>	<b>High Threshold</b>	<b>Very High Threshold</b>
<b>CLOC</b>	54.26	77.72	0	131.98	197.97
<b>MLOC</b>	9.51	11.79	0	21.30	31.95
<b>NOM</b>	3.85	26.7	0	30.55	45.825
<b>SIZE2</b>	41.85	57.07	0	98.92	148.38
<b>CC</b>	2.42	3.33	0	5.75	8.63
<b>RFC</b>	11.76	13.49	0	25.25	37.88
<b>WMC</b>	10.59	19.24	0	29.83	44.75
<b>DIT</b>	1.49	1.36	0.13	2.85	4.28
<b>NOC</b>	0.83	3.77	0	4.6	6.9
<b>Ca</b>	451.29	529.06	0	980.35	1470.53
<b>CBO</b>	14.78	26.10	0	40.88	61.32
<b>LCOM</b>	2.23	1.89	0.34	4.12	6.18
<b>CRAT</b>	17.59	9.91	7.68	27.50	41.25

#### 4.1.1 Long Method

The condition that must be met in a method in order to be a long method is:

$MLOC > 10.65$  AND  $MAXNESTING \geq 2$  AND  $CYCLO \geq 5.75$  AND  $NOAV > 7$

Where MLOC makes reference to the number of lines of the method, that needs to be higher than 10.65 (high threshold/2), MAXNESTING makes reference to the maximum level of nesting of control structures (if, for, switch...) that needs to be higher than 2 (several) and also to be a long method NOAV (number of accessed variables) needs to be higher than 7 (many).

### 4.1.2 Large Class

The condition that must be met for a class to be called a Large Class, or God Class, is:

$$\text{ATFD} > 4 \text{ AND } \text{WMC} \geq 44.75 \text{ AND } \text{TCC} < \frac{1}{3}$$

To constitute a Large Class, a class needs to fit the following three criteria: 1) Its Weighted Method Count (WMC) is greater than or equal to the “Very High” threshold (more than 44.75 in our case). 2) Its Tight Class Cohesion (TCC) metric is above a value of  $\frac{1}{3}$ . 3) Its Access To Foreign Data metric is greater than the “Few” threshold (more than 4 in our case).

### 4.1.3 Duplicate Code

The condition that must be met in a method in order to be a duplicate code is:

$$\text{SEC} > 9.51 \text{ OR } (\text{SDC} \geq 11 \text{ AND } \text{SEC} > 2 \text{ AND } \text{LB} \leq 5)$$

SEC, short for Size of Exact Clone, measures the size of duplicated code by counting the number of consecutive lines of code that are detected as exact clones.

LB, short for Line Bias, represents the number of non-matching lines of code, between two consecutive exact clones when comparing two pieces of code. It helps to identify if these clones belong to the same cluster of duplicated lines.

SDC, short for Size of Duplication Chain, focuses on organizing small duplication chunks into more meaningful blocks called duplication chains, where chunks are considered part of the same chain if LB values are below a specified threshold.

### 4.1.4 Shotgun Surgery

For a code segment to be classified as exhibiting "Shotgun Surgery," the following conditions must be met:

WCM (Weighted Class Method) > 7, indicating a high level of interdependence between classes and methods. This suggests that a change in one class or method may require modifications in multiple other classes.

CC (Changing Classes) > 7, indicating that a significant number of classes are affected by changes in this particular code segment. This implies a wide-reaching impact on the system, potentially leading to increased maintenance efforts and complexity.

#### **4.1.5 Feature Envy**

The condition that must be met in a method in order to be a feature envy is:

$$\text{ATFD} > 4 \text{ AND } \text{LAA} < 1/3 \text{ AND } \text{FDP} \leq 4$$

Where ATFD means access to foreign data (getters included), that needs to be higher than 4 (few). LAA is the relation between the number of attributes from the class divided by the total number of variables accessed, this parameter needs to be less than 1/3. The last metric is FDP, that is the number of classes with the attributes from ATFD that needs to be less than 4 (few).

## 4.2 Specific Anti-Patterns in Pixel Dungeon

### 4.2.1 Long Method

The first anomaly identified was a long method in the class `displacement.java`, this class has a long method named `proc`:

```
@Override
public int proc( Armor armor, Char attacker, Char defender, int damage ) {

    if (Dungeon.bossLevel()) {
        return damage;
    }

    int level = armor.effectiveLevel();
    int nTries = (level < 0 ? 1 : level + 1) * 5;
    for (int i=0; i < nTries; i++) {

        int pos = Random.Int( Level.LENGTH );
        if (Dungeon.visible[pos] && Level.passable[pos] && Actor.findChar( pos ) == null) {

            WandOfBlink.appear( defender, pos );
            Dungeon.level.press( pos, defender );
            Dungeon.observe();

            break;
        }
    }

    return damage;
}
```

Figure 6: `proc` method from the `displacement` class

The method exceeds the recommended line limit (MLOC = 10.65) with a code length of 18 lines. Additionally, the Cyclomatic Complexity (CC) is 7, which is higher than the acceptable threshold of 5.75. The MAXNESTING is within the specified limit (MAXNESTING = 2), meeting the criteria for this parameter. Finally, the Number of Accessed Variables (NOAV) is 12, which exceeds the set threshold of 7.

### 4.2.2 Large Class

The second anomaly we found is a Large Class, also known as God Class.

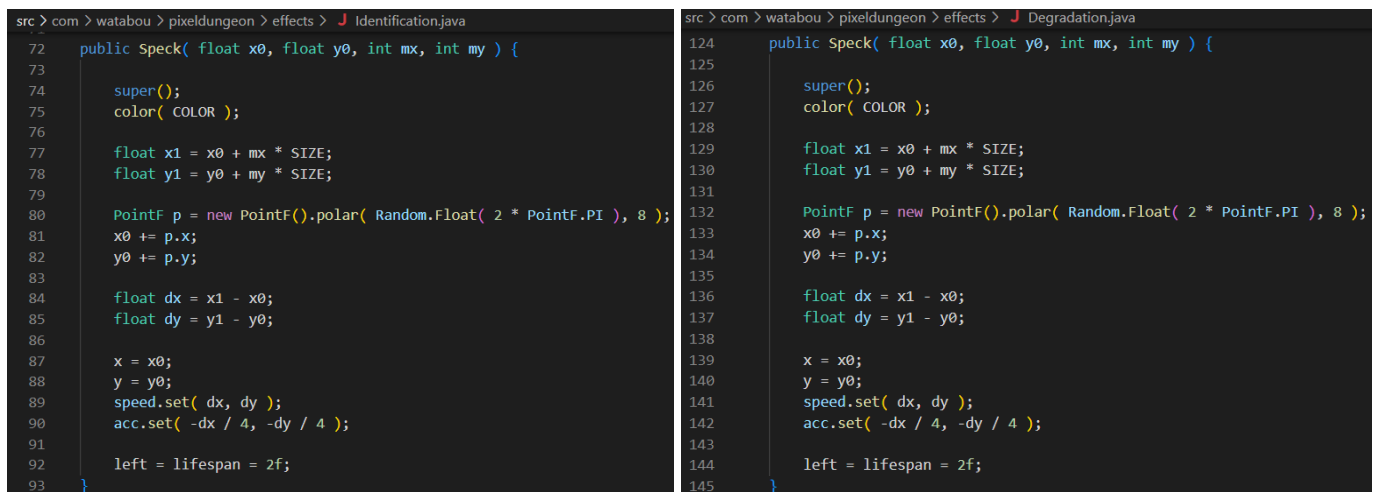
Class	ATFD	WMC	TCC ▼
Actor	5	52	0.3557

Figure 7: Metrics values of the Actor class, found using MetricsTree

The Actor class matches each part of the criteria for this anomaly. It has a WMC of 52 which is higher than the threshold of 44.75 and its TCC of 5 exceeds the “Few” threshold of 4. Finally, it has an ATFD of 0.3557, which is slightly above the cutout of 0.333.

### 4.2.3 Duplicate Code

The third anomaly we found in the project is Duplicate Code. As the figure below shows, it is obvious that some code snippets in Identification.java and Degradation.java are exactly the same.



```

src > com > watabou > pixeldungeon > effects > Identification.java
72 public Speck( float x0, float y0, int mx, int my ) {
73
74     super();
75     color( COLOR );
76
77     float x1 = x0 + mx * SIZE;
78     float y1 = y0 + my * SIZE;
79
80     PointF p = new PointF().polar( Random.Float( 2 * PointF.PI ), 8 );
81     x0 += p.x;
82     y0 += p.y;
83
84     float dx = x1 - x0;
85     float dy = y1 - y0;
86
87     x = x0;
88     y = y0;
89     speed.set( dx, dy );
90     acc.set( -dx / 4, -dy / 4 );
91
92     left = lifespan = 2f;
93 }

src > com > watabou > pixeldungeon > effects > Degradation.java
124 public Speck( float x0, float y0, int mx, int my ) {
125
126     super();
127     color( COLOR );
128
129     float x1 = x0 + mx * SIZE;
130     float y1 = y0 + my * SIZE;
131
132     PointF p = new PointF().polar( Random.Float( 2 * PointF.PI ), 8 );
133     x0 += p.x;
134     y0 += p.y;
135
136     float dx = x1 - x0;
137     float dy = y1 - y0;
138
139     x = x0;
140     y = y0;
141     speed.set( dx, dy );
142     acc.set( -dx / 4, -dy / 4 );
143
144     left = lifespan = 2f;
145 }

```

Figure 8: Code snippets in Identification.java and Degradation.java is a copy-paste case

According to the definition of metrics corresponding to Duplicate Code, there is no difference between the two methods (LB = 0). On the other hand, the two clones in



this case have 15 identical lines (SEC = 15), which is above the acceptance threshold of 9.51. As can be seen, it meets our standard of Duplicate Code.

#### 4.2.4 Shotgun Surgery

The identified anomalies in the codebase are indicative of Shotgun Surgery, a phenomenon where a single code change necessitates multiple modifications across various parts of the project. This is typically symptomatic of code that is tightly coupled or poorly structured.

An example of this phenomenon can be observed in the extensive use of buffs within the provided code. There are numerous types of buffs (such as `Bleeding`, `Burning`, `Invisibility`, `Paralysis`, etc.) that can be applied to characters. Each type of buff requires specific handling logic within the `add` and `remove` methods. Consequently, introducing a new type of buff or making changes to an existing one could potentially demand alterations across multiple areas of the codebase.

```
public void updateSpriteState() {
    for (Buff buff:buffs) {
        if (buff instanceof Burning) {
            sprite.add( CharSprite.State.BURNING );
        } else if (buff instanceof Levitation) {
            sprite.add( CharSprite.State.LEVITATING );
        } else if (buff instanceof Invisibility) {
            sprite.add( CharSprite.State.INVISIBLE );
        } else if (buff instanceof Paralysis) {
            sprite.add( CharSprite.State.PARALYSED );
        } else if (buff instanceof Frost) {
            sprite.add( CharSprite.State.FROZEN );
        } else if (buff instanceof Light) {
            sprite.add( CharSprite.State.ILLUMINATED );
        }
    }
}
```

Figure 9: Code snippets showing shotgun surgery for buffs

Regarding the metrics, the Weighted Method Count (WMC) for `Char.java` is 8, as eight methods in this class are dependent on buffs. However, for the CC metric, it

extends to 14 classes (including `Albino.java`, `Bandit.java`, etc.) where similar interdependencies exist. This further emphasizes the widespread nature of Shotgun Surgery in the codebase.

### 4.2.5 Feature Envy

The last anomaly we identified was Feature Envy in the class Blacksmith were in the method verify() there are multiples access to the class Item as it can be observed in the next Figure:

```
public static String verify( Item item1, Item item2 ) {  
  
    if (item1 == item2) {  
        return "Select 2 different items, not the same item twice!";  
    }  
    if (item1.getClass() != item2.getClass()) {  
        return "Select 2 items of the same type!";  
    }  
    if (!item1.isIdentified() || !item2.isIdentified()) {  
        return "I need to know what I'm working with, identify them first!";  
    }  
    if (item1.cursed || item2.cursed) {  
        return "I don't work with cursed items!";  
    }  
    if (item1.level() < 0 || item2.level() < 0) {  
        return "It's a junk, the quality is too poor!";  
    }  
    if (!item1.isUpgradable() || !item2.isUpgradable()) {  
        return "I can't reforge these items!";  
    }  
  
    return null;  
}
```

Figure 10: verify method from the Blacksmith class

This method makes 10 ATFD (accesses to foreign data), all to the same class (FDP = 1) and it has a ratio of LAA of 0.1 (we did not take into account the final static attributes as they are constants). As it can be observed it accomplishes our criteria for a Feature Envy.

## 5. Pixel-Dungeon Anomaly Correction

### 5.1 Anomaly Correction by Refactoring

Note: the refactored codebase can be found at:

<https://github.com/Rmolina2002/LOG8430---TP2>

#### 5.1.1 Long Method

To correct the method `proc` we decided to refactor it. What we did was divide the method `proc` into 5 different methods, these methods encapsulate specific parts of the code that were previously located in one long method. The resulting code is:

```
@Override
public int proc(Armor armor, Char attacker, Char defender, int damage ) {
    if (Dungeon.bossLevel()) {
        return damage;
    }

    int nTries = calculateTries(armor);
    performTries(nTries, defender);
    return damage;
}

!usage new *
public void performTries(int nTries, Char defender){
    for (int i=0; i < nTries; i++) {
        int pos = Random.Int( Level.LENGTH );
        if (isBlinkPossible(pos)) {
            performBlink(defender, pos);
            break;
        }
    }
}

!usage new *
public boolean isBlinkPossible(int pos){
    return Dungeon.visible[pos] && Level.passable[pos] && Actor.findChar( pos ) == null;
}

!usage new *
public void performBlink(Char defender, int pos){
    WandOfBlink.appear( defender, pos );
    Dungeon.level.press( pos, defender );
    Dungeon.observe();
}

!usage new *
public int calculateTries(Armor armor){
    int level = armor.effectiveLevel();
    return (level < 0 ? 1 : level + 1) * 5;
}
```

Figure 11: code of the method `proc` after refactoring

## 5.1.2 Large Class

In order for a class to count as a God Class, it must meet all three criteria. Therefore, if only one of those is modified to a value under the related threshold, the class no longer fits the definition of a Large Class.

Three options are available to fix the issue. The weighted method count could be decreased to less than 44.5; the Tight Class Cohesion could be lowered to under  $\frac{1}{3}$ , or the Access To Foreign Data could be lowered to less than 4. We chose to decrease the method count by extracting an ActorSet class from the Actor class.

```
35 public abstract class ActorSet implements Bundlable {
36
37     private static HashSet<Actor> actors = new HashSet<Actor>();
38     private static SparseArray<Actor> ids = new SparseArray<Actor>();
39     private static Char[] chars = new Char[Level.LENGTH];
40
41     private static float now = 0;
42
43 > public static HashSet<Actor> getActors() { ...
46
47 > public static Char[] getChars() { ...
50
51 > public static void clear() { ...
60
61 > public static void add( Actor actor ) { ...
64
65 > public static void addDelayed( Actor actor, float delay ) { ...
68
69 > private static void add( Actor actor, float time ) { ...
92
93 > public static void remove( Actor actor ) { ...
104
105 > public static Actor findById( int id ) { ...
108
109 > public static Char findChar( int pos ) { ...
112
113 }
```

Figure 12. The members of the ActorSet class, with their implementation collapsed.

The ActorSet class takes the “all”, “chars” and “ids” attributes from the Actor class and extracts them in a separate class. The associated methods (clear(), both versions of add(), addDelayed(), remove(), findById() and findChar()) are also extracted. This allows the main Actor class to concentrate on its primary

responsibility of defining the properties of an actor, instead of messing with actor lists.

### 5.1.3 Duplicate Code

To solve the problem, first we create a common class that contains a shared constructor method for the `Speck` class.

```
public class SpeckUtils {  
    public static Speck createSpeck(float x0, float y0, int mx, int my) {  
        Speck speck = new Speck();  
        speck.color(COLOR);  
  
        float x1 = x0 + mx * SIZE;  
        float y1 = y0 + my * SIZE;  
  
        PointF p = new PointF().polar(Random.Float(2 * PointF.PI), 8);  
        x0 += p.x;  
        y0 += p.y;  
  
        float dx = x1 - x0;  
        float dy = y1 - y0;  
  
        speck.x = x0;  
        speck.y = y0;  
        speck.speed.set(dx, dy);  
        speck.acc.set(-dx / 4, -dy / 4);  
  
        speck.left = speck.lifespan = 2f;  
  
        return speck;  
    }  
}
```

Figure 13: code of the SpeckUtils for refactoring

After that, in both Degradation.java and Identification.java call the `createSpeck` method from the `SpeckUtils` class to create a Speck instance, eliminating code duplication:

```
Speck speck = SpeckUtils.createSpeck(x0, y0, mx, my);
```

Figure 14: code of the calling method in Degradation.java and Identification.java

By doing this, we can eliminate the code duplication and ensure both classes share the same logic to create a Speck object.

#### 5.1.4 Shotgun Surgery

The original code exhibits shotgun surgery due to coupling the **Char** class with specific buff types. This creates a situation where a change in one part of the code requires modifications in many places, violating the Single Responsibility Principle.

By applying polymorphism, the responsibility of handling behavior associated with each buff is delegated to the respective **Buff** class. It also encapsulates the behavior of each buff within the respective **Buff** class. The refactoring paves the way for easier extensibility. Introducing a new type of buff becomes a straightforward process. This significantly reduces interdependencies, adheres to the Single Responsibility Principle, and effectively combats shotgun surgery.

There's potential for similar shotgun surgery issues in other classes dependent on buffs, such as **Albino**, **Bandit**, **DM300**, etc.

The same refactoring principle can be applied to other classes reliant on buffs. This ensures consistency and improved maintainability throughout the codebase, effectively addressing potential shotgun surgery anomalies.

Table 11: Refactoring code with Shotgun Surgery anomaly

Original code	Refactored
<pre> public void updateSpriteState() {     for (Buff buff:buffs) {         if (buff instanceof Burning) {             sprite.add( CharSprite.State.BURNING );         } else if (buff instanceof Levitation) {             sprite.add( CharSprite.State.LEVITATING );         } else if (buff instanceof Invisibility) {             sprite.add( CharSprite.State.INVISIBLE );         } else if (buff instanceof Paralysis) {             sprite.add( CharSprite.State.PARALYSED );         } else if (buff instanceof Frost) {             sprite.add( CharSprite.State.FROZEN );         } else if (buff instanceof Light) {             sprite.add( CharSprite.State.ILLUMINATED );         }     } } </pre>	<pre> public void updateSpriteState() {     for (Buff buff : buffs) {         buff.applySpriteState(sprite);     } } </pre>
<pre> public abstract class Buff {     // ... }  public class Burning extends Buff {     // ... } </pre>	<pre> public abstract class Buff {     // ...      public abstract void applySpriteState(CharSprite sprite); }  public class Burning extends Buff {     // ...      @Override     public void applySpriteState(CharSprite sprite) {         sprite.add(CharSprite.State.BURNING);     } } </pre>

### 5.1.5 Feature Envy

The solution we found was to relocate the verification logic to the Item class. By doing so, we ensured that all access operations performed during these verifications remain internal to the Item class. Consequently, the count of foreign classes accessed was reduced to zero, as all access operations are now confined within the boundaries of the Item class. This refactor can be observed in the next Figure, the whole code can be observed in GitHub:

```
public class Item implements Bundlable {  
    1 usage new *  
    public String verifyItems(Item other) {  
        //verifications  
  
        return null;  
    }  
}  
  
public static String verify( Item item1, Item item2 ) {  
    return item1.verifyItems(item2);  
}
```

Figure 15: code for the verify method after refactoring



## 5.2 Metrics after Refactoring

### 5.2.1 Long Method

After measuring again the method proc of the class displacement.java we obtained the following results:

Table 12: comparing the metrics of long method before and after refactoring

Metric	Value Before	Value After				
	proc	proc	performTries	isBlinkPossible	performBlink	calculateTries
<b>MLOC</b>	18	9	9	3	5	4
<b>Nesting</b>	2	1	2	0	0	0
<b>CC</b>	7	2	3	3	1	2
<b>NOAV</b>	12	4	5	3	3	2

As it can be observed in the table, in the `value after` part, each of the individual functions created does not accomplish our criteria for a long method so we can conclude that the anomaly has been fixed.

### 5.2.2 Large Class

After the extraction of the ActorSet class, the values of the metrics associated with the Large Class anomaly changes thusly:

Table 13: comparing the metrics of large class before and after refactoring

Metric	Value Before	Value After
<b>WMC</b>	52	42
<b>TCC</b>	0.3557	0.3216
<b>ATFD</b>	5	7

Therefore, extracting the class made the ATFD worse, but made the other two criteria drop to under the associated thresholds of 44 and 0.3333. This makes sense, since the refactoring removed some methods from the Actor class (which lowered its Weighted Method Count). This made the class more focused on a single responsibility (which gave it a Tighter Class Cohesion). However, since we accessed the methods in the newly extracted ActorSet class from the Actor class, the number of Accesses To Foreign Data grew.

### 5.2.3 Duplicate Code

After refactoring, the `Speck` method in both `Degradation.java` and `Identification.java` were separated into the `SpeckUtile` class, leaving only one identical code that calls the `createSpeck` method.

Table 14: comparing the metrics of Duplicate Code before and after refactoring

Metric	Value Before	Value After
<b>SEC</b>	15	1
<b>LB</b>	0	0
<b>SDC</b>	15	1

### 5.2.4 Shotgun Surgery

After measuring the metrics to verify the refactoring method used we obtained the following results:

Table 15: Metric Comparison before and after refactoring

<b>Metric</b>	<b>Value Before</b>	<b>Value After</b>
<b><i>WMC</i></b>	8	5
<b><i>CC</i></b>	14	0

The values for both metrics are now below the associated threshold of 7. The Shotgun Surgery is therefore fixed.

### 5.2.5 Feature Envy

After measuring again the method verify of the class Blacksmith we obtained the following results:

Table 16: comparing the metrics of feature envy before and after refactoring

<b>Metric</b>	<b>Value Before</b>	<b>Value After</b>
<b><i>ATFD</i></b>	10	0
<b><i>FDP</i></b>	1	0
<b><i>LAA</i></b>	0.1	2

Now, the Blacksmith class no longer accesses any external data, as all the logic is contained within this class Item. Additionally, since the only Foreign Data Provider (FDP) was Item, its value has been reduced to 0. As a consequence, the Lack of Adequacy of a Class (LAA) metric has increased, given that the majority of variable accesses have been consolidated within the Item class. This change results in improved encapsulation and a more self-contained class design and we can conclude that the Feature Envy antipattern was fixed.

### 5.3 QMOOD after Refactoring

After the changes to the code described in the rest of Section 5 were made, the measured qualities of the project were bound to have changed. The values are as follows:

Table 17: QMOOD qualities before and after refactoring

Quality Attribute	Value before	Value after	Change
Effectiveness	3.1704	3.189	+ 0.587 %
Extendibility	3.0313	3.0353	+ 0.132 %
Flexibility	5.1520	5.1981	+ 0.895 %
Functionality	4.9456	4.9452	- 0.008 %
Reusability	5.4036	5.3965	- 0.131 %
Understandability	-9.3767	-9.3788	- 0.022 %

The changes are slight. The greatest effect is a gain of a little less than 1% in Flexibility, followed by a gain of around half a percent in effectiveness. The rest changed by less than 0.2%, including three quality attributes that saw declines.

This is to be expected, considering the relative minuscule amount of code changed on the scale of this project of almost 50 000 lines. It is encouraging however that the largest changes are positive.

Though Flexibility was not discussed in Section 3.3, the design properties most strongly associated with it are Composition and Polymorphism. Their associated metrics, Measure Of Aggregation (MOA) and Number of Polymorphic Methods (NOP), have neither been included in the analysis nor have a close relative included. We can however see that two of the five anomaly fixes included an aggregation, adding a member in a class that is an instance of another class. This could have affected the MOA positively.

As for effectiveness, all the associated design properties have the same weight in computing the Quality Attribute. Changes in any of the Average Number of Ancestors, Data Access Metric, Measure of Aggregation, Measure of Functional Abstraction or Number of Polymorphic Methods metric would have affected it. However, since none of them was directly included in this study, the direct effect is hard to quantify. We can posit that positive change to ATFD from the Feature Envy fix and in NOAV from the Long Method fix had a part to play in this small gain.

## 6. Conclusion

The Pixel Dungeon code base is a great canvas to study the usefulness of code metrics in a real-life scenario. Most of the code is clean and well thought-out, but with some issues that metrics help to shed light on.

This assignment showed how using metrics can dramatically improve our understanding of a code base, with its strengths as well as its flaws, and allow us to direct improvements where they will be the most impactful. However small it was, having had a measurable impact on the QMOOD Quality Attributes with so few changes on such a large project shows the promise of a metrics-based approach to refactoring.