Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Oskari Liukku

Clean Code:

Analysing game code using agile programming practices

Master's Thesis
Espoo, February 27, 2020

Supervisor: Asst. Prof Perttu Hämäläinen

Advisor: Asst. Prof Perttu Hämäläinen

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Oskari Liukku

Title:
Clean Code: Analysing game code using agile programming practices

Date: February 27, 2020 **Pages:** 51

Major: Computer Science **Code:** SCI3046

Supervisor: Asst. Prof Perttu Hämäläinen

Advisor: Asst. Prof Perttu Hämäläinen

Agility in software development means being able to adapt to ever-changing requirements. A major factor in agility is keeping the codebase easy to read and to extend. This kind of code is commonly called "clean code". Clean code is simple, elegant, and does not repeat itself. Writing clean code means that the cost of adding, changing, and removing features is kept to a minimum. It front-loads more of the work into the design stages of development, where it is cheaper both financially as well as mentally.

This thesis describes some of the main principles of clean code and shows how they can be used in analyzing game code. The game under scrutiny is The Last Cube, a large-scale puzzle game project in which the author is the lead programmer. The game has been in development in the Unity game engine over 3 years. Parts of the code of The Last Cube are described and then analyzed using various methods and tools. The analysis of the code revealed hundreds of major violations of the clean code principles listed in the thesis, including problems with coupling, complexity, and duplication. The problems are analyzed and discussed. One instance of coupling is taken under closer scrutiny and analyzed further. The section is then refactored, improving the section's maintainability and reducing coupling.

Keywords: game, programming, clean code, agile, object-oriented programming, analysis

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Oskari Liukku

Työn nimi:
Clean Code: Ketterien ohjelmointikäytäntöjen hyödyntäminen pelikoodin analysoinnissa

Päiväys: 27. helmikuuta 2020 **Sivumäärä:** 51

Pääaine: Tietotekniikka **Koodi:** SCI3046

Valvoja: Asst. Prof Perttu Hämäläinen

Ohjaaja: Asst. Prof Perttu Hämäläinen

Ohjelmistokehityksessä termi "ketteryys"viittaa kykyyn sopeutua alati muuttuviin vaatimuksiin. Yksi suurimmista tekijöistä ketteryydessä on koodin luettavuus ja laajennettavuus. Tälläistä koodia kutsutaan usein termillä "clean code"eli "puhdas koodi". Puhdas koodi on yksinkertaista ja eleganttia, eikä se toista itseään turhaan. Puhtaan koodin kirjoittaminen auttaa pitämään toiminnallisuuksien lisäämisen, muuttamisen ja poistamisen kulut minimissä. Käyttämällä enemmän aikaa työn suunnitteluvaiheissa, ohjelmointi on helpompaa sekä taloudellisesti että henkisesti.

Tämä työ kuvaa puhtaan koodin pääperiaatteita ja esittelee niiden käyttöä pelikoodin analysoinnissa. Tutkittava peli on The Last Cube, suuren mittakaavan pulmapeliprojekti, jossa työn kirjoittaja on pääohjelmoijana. Peliä on kehitetty Unity-pelimoottoriin pohjautuen yli kolmen vuoden ajan. Osia The Last Cube -pelin koodista esitellään ja analysoidaan käyttäen erilaisia metodeja ja työkaluja. Koodin analysointi paljasti satoja kohtia, joissa työssä esiteltyjä puhtaan koodin periaatteita oli rikottu. Löytyneitä ongelmakohtia esitellään ja analysoidaan, ja erästä funktiota tarkastellaan lähemmin ja lopulta muutetaan puhtaammaksi. Muutoksen jälkeen tehdyn analyysin mukaan funktio on lyhyempi sekä huomattavasti parempi ylläpidettävyyden kannalta.

Asiasanat: peli, ohjelmointi, clean code, ketterä, analyysi, olio-ohjelmointi

Kieli: Englanti

Acknowledgements

I wish to thank Miikka Junnila and Perttu Hämäläinen for guiding me through my master's studies, as well as Max Samarin, Miika Kanerva, Jussi Joki and Lauri Lappalainen for founding Improx Games Oy with me and working so hard on the game this thesis is based on: The Last Cube. Thank you to Garen DiBernardo and Max Samarin for help with proof-reading.

I also want to thank my family as well as Erika Anttila for supporting me throughout all this.

Espoo, February 27, 2020

Oskari Liukku

Abbreviations

API	Application programming interface
DRY	Don't repeat yourself
IDE	Integrated development environment
IL	Intermediate language
TDD	Test-driven development
XP	Extreme Programming

Contents

Chapter 1

Introduction

More time is spent reading code than writing it. Working on bad, messy code leads to large amounts of time wasted on just understanding it. Changes are difficult to make and programmers therefore avoid them, instead opting to write their own solutions, leading to duplicated code. Management sees how slow progress is and hires more programmers, who have to start parsing the messy codebase from zero. Productivity slows down to a halt. Making code easy to read makes it easy to write. [1]

Robert C. Martin's book Clean Code [1] is one of many works of literature offering guidance in writing code that is understandable, reusable, and consistent. Good software design following these guidelines helps keep development timelines predictable and the code base agile — poised for changes and new requirements [2, 3]. Martin uses the term "code sense" to refer to the ability to not only recognize that code is "bad", but to also see opportunities to change it towards these guidelines. [1]

This thesis aims to bring forth various rulesets and methodologies of agile software design and development, as well as apply them to analyze a real game project's source code. The game under analysis is The Last Cube, a large-scale soon to be released indie puzzle game made in the Unity game engine by Improx Games Oy. The author is a lead programmer in the project. The game was written in the C# programming language.

The thesis is structured as follows. After this introduction, chapter 2 will list and explain numerous principles and rules presented in the literature, as well as providing background on the state of the art of the effectiveness of some of those principles. Chapter 3 will introduce the game to be analyzed, giving a description of the gameplay as well as an overview of the structure of the game's code. Chapter 4 contains an analysis of the game code using concepts explained in chapter 2. Finally, chapter 5 contains conclusions and a summary of the work.

Chapter 2

Agile Programming Practices

Agility equals being able to adapt to ever-changing requirements. Following certain principles while programming makes code less stiff and easier to change. This chapter reviews common principles, rules, and terms used in writing agile software. The chapter concludes with a review of refactoring, as it is the practical process of reworking a codebase towards clean code principles.

2.1 Clean Code

The term clean code was popularized by Robert C. Martin (who is best known for being one of the co-authors of the Agile Manifesto) in his books [1, 4] and blog posts. These books provide rule-sets for writing clean, more agile code that is easy to read, refactor and reuse.

Grady Brooch, a pioneer of object-oriented design and co-author of [5], describes clean code in [1, p. 8] as: "Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control."

Martin writes that authoring good code is crucial in the long term. Bad code is quick to write, but it will eventually crumble into a mess that no one wants to work on. This causes losses both financially as development times grow due to difficulty of understanding the code and adding onto it, as well as mentally, causing morale to drop. [1]

A study done on the effects of clean code on software's understandability [6] found that code that was written using the clean code ruleset was significantly faster to add features to. They did however also find that changing the functionality of clean code and finding bugs in it was slower than with their control code. They propose that clean code might be more effective in larger

systems and in environments where the developers have gotten a chance to thoroughly learn the rulesets.

2.2 The SOLID principles

The SOLID principles are another concept popularized by Martin. SOLID is an acronym for five principles promoted in their work [7]. It consists of the following: Single-responsibility principle, Open-closed principle, Liskov substitution principle, Interface segregation principle, and Dependency inversion principle. [8] Table 2 contains short definitions for these principles.

Principle	Short description
Single-responsibility principle	There should never be more than one reason for a class to change.
Open-closed principle	Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
Liskov substitution principle	Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
Interface segregation principle	Clients should not be forced to depend upon interfaces that they do not use.
Dependency inversion principle	A. High-level modules should not depend upon low-level modules. Both should depend upon abstractions. B. Abstractions should not depend upon details. details should depend upon abstractions.

Table 2: The SOLID principles and their definitions [8]

The Single-responsibility principle rules that a single class or function should do one thing and one thing only. If a class has more than one responsibility, whenever any of those responsibilities' requirements change, the class has to change too. Thus, separate functionality should be split into separate classes. An example of a violation of the principle could be a Triangle class that has functions both for rendering the triangle and calculating its area. To avoid the violation, these responsibilities should be separated, for example by creating and using a separate class for rendering the shape. [8] Having more than one responsibility also means those responsibilities are coupled — changing one part might affect or break the other. Coupling is discussed further in Section 2.3.

The Open-closed principle states that classes should be extendable without modifying them. This is most often achieved through polymorphism and

abstraction. According to Martin, by planning ahead, code can be written in a way that adding new features does not require the modification of old code. [7] This not only lessens the amount of code a new programmer needs to parse before they can implement a new feature but also that working code never, or rarely, needs to be modified and subjected to the risk of breaking again. Martin recognizes that reaching full compliance of the open-closed principle is near-impossible in meaningful programs, and thus closures must be strategically chosen to achieve even partial compliance. [7, 9]

The Liskov substitution principle was first introduced by Liskov in [10, p. 7] where they describe a desired substitution property of a type hierarchy: if a class or function references a type, a subtype of that type must be able to be substituted in its place without the class or function knowing. If this is not the case, the Open-closed principle is also being violated because the function knows about the referenced type's subtypes and must thus be changed any time a new subtype is created. Indeed, Martin describes the Liskov substitution principle as a feature of programs that conform to the Open-closed principle. [11] The ability to substitute objects in place of others based on their common parent or interface is known as polymorphism [5].

The Interface segregation principle states that interface clients (classes implementing an interface or inheriting from another class) should not have to care about methods that are not of interest to them. A "polluted" interface's functions must either be implemented or overridden in each client or marked as optional which can lead to violations of the Liskov substitution principle. To avoid this, large interfaces should be split up into smaller ones based on functionality. [12] These smaller, more specific interfaces can then be implemented and used in the places they are needed. For example, a class servicing two separate clients A and B should instead implement two interfaces: IClientAFunctions and IClientBFunctions. The clients can then reference the same class via these interfaces without having access to functions outside their needs. [7]

The Dependency inversion principle rules that modules should be separated via abstraction. A high-level module might utilize multiple lower-level modules. Instead of directly referencing them, the higher-level module should reference them via an interface (or abstract class) which the lower-level modules implement. This process decouples the modules: changes to a lower-level module no longer mean the higher-level module has to change. [7, 13] According to Martin [13] the principle of dependency inversion is at the root of writing code that is reusable, resilient to change, and maintainable.

2.3 Coupling

In [13] Martin argues that the cause of "bad code" is the interdependence of modules within it. This interdependence is called coupling, and it is one of the core problems many programming guides and principles try to avoid. Tight coupling causes code to be fragile: changing one part affects all its dependents, possibly breaking them. [8, p. 150] This makes the code stiff and hard to manipulate and extend, causing difficulties in adapting to new requirements and features. [13] Another incentive to reduce coupling is having easily testable code — writing tests for code that is tightly coupled is difficult because the component requires other components to function. [1, p. 172] Testing and testability are discussed further in Section 2.9.

Designing loosely-coupled code increases the probability of the code being reused, which is one of the main goals of software design. Tight coupling is among the top reasons for having to redesign parts of code. [14] The process of removing coupling from code is called decoupling. The goal of decoupling is to transform code from a tightly coupled state to a weakly coupled form. [8, 15] One way to achieve this is by moving related portions of code together into appropriate and encapsulated (meaning self-contained) classes [5, 16] that then communicate with each other via abstractions, as per the Dependency inversion principle [7]. The term cohesion describes the relatedness of a class's contents, and it usually improves when coupling is reduced. [16, p. 1] Complete decoupling is rarely possible due to limitations of the hardware, operating system or programming language [8, p. 152], but tools and methodologies have been developed to help recognize and perform decoupling actions. [16]

Many rules and guidelines have been written to promote loose coupling, one of them being the Law of Demeter [17]. The Law of Demeter limits what other functions a function can access. It rules that a function should only invoke functions in its own class, in its parameter objects, any objects it creates, or its direct components [18]. This helps reduce coupling — the function is only in contact with functions that it is directly related to, which makes the code more robust. The downside of following the Law of Demeter is having to write wrapper functions that do nothing other than pass along a message. [19]

Types of coupling can be grouped into categories. One framework by Eder and Schrefl [15] sorts coupling into three dimensions: interaction coupling, component coupling, and inheritance coupling. Methods and classes that invoke each other or share data with one another are said to be coupled by interaction. Component coupling applies only to classes and is in effect if a

Degree	Description
Content	The method directly accesses parts of internal structure, i.e. private variables
Common	Methods communicate via unstructured global data, i.e. global variables
External	Methods communicate via structured global data, i.e. public variables
Control	Methods communicate via passing parameters, and the parameters affect the execution of one of the methods
Stamp	Methods communicate via passing whole data structures
Data	Methods communicate only via relevant parameters

Table 3: Dimensions and degrees of interaction coupling [15]

class is referenced in another class via variable or method parameter. Classes that are in a parent-child inheritance relationship are inheritance coupled. The framework further splits each dimension into categories by the severity of the coupling. Table 3 presents these categories, sorted from worst to best (most to least coupling). In [20] researchers show that a set of metrics can be linked to these categories. They also present a unified framework for mathematically measuring and categorizing coupling.

Degree	Description
Coincidental	Elements have nothing in common
Logical	Elements have similar functionality
Temporal	Logically cohesive, and related in time
Procedural	Connected by some control flow
Communicational	Procedurally cohesive, and refer to the same data
Sequential	Communicationally cohesive, and connected to the same sequential control flow
Functional	Sequentially cohesive, and contribute to the same task

Table 4: Degrees of method cohesion of code elements [15, pp. 22-29, 21]

Similarly to coupling, cohesion can also be grouped into domains. Eder and Schrefl [15, pp. 22-29] describe three domains of cohesion: method, class, and inheritance cohesion. They split each domain into degrees of severity based on their characteristics. Table 4 shows their seven degrees of method cohesion, sorted from least to most cohesive.

2.4 Composition over inheritance

Composition — a class using instances of other classes — and inheritance — a class extending another one — are both methods of reusing functionality. Inheritance is sometimes referred to as "white box reuse" because the contents of the parent class are often visible to the child. Composition, on the other hand, is known as "black box reuse", because the internal details of the class are by design not visible, only its public API. [14, p. 32]

The question of when to use inheritance versus composition is a question that does not have a definite answer: both have their upsides and downsides. Generally, programmers are taught to use inheritance in situations where the two classes are in an "is-a" relationship, meaning the child class is a more specialized version of the parent, whereas composition is to be used for "has-a" relationships, where the owner class has a feature described by the child object. [22, p. 61]

With inheritance, the child class is tightly coupled to its parent's implementation, forcing it to change whenever the parent changes [5, 14]. Bloated parent classes can lead to violations of the Interface segregation principle by forcing their children to include functionality they do not need. All child classes overriding a virtual function from the parent depend on not only its functionality but its signature — any change to for example the parameters of the function must be manually cascaded to all children [22, pp. 61-63]. Using composition, the programmer is forced to respect the class's interface and incentivized to outsource functionality, helping the code obey both the Liskov substitution principle and the Single responsibility principle [14].

2.5 DRY

DRY, short for Don't Repeat Yourself, is one of the core rules of agile programming. It is another rule that points towards the aforementioned goal of code reuse. It was first formalized in [19] as "every piece of knowledge must have a single, unambiguous, authoritative representation within a system." It is one of the most important principles behind the ruleset of Extreme Programming [1]. [23] recognizes code duplication as a top reason for the most common code design problems. In [24] Fowler recommends a "Rule of Three" for when to refactor to avoid duplication: the second time you write code similar to something you already have, you take note of the duplication — on the third time you refactor the segments.

Most code design patterns that have come about after the popularization of the DRY principle are just ways to remove duplication and introduce

abstraction [1]. For example, the Extract Function refactoring mentioned above moves a block of code to a function, which can then be reused. Moving duplicated dependencies also improves import coupling by localizing the dependencies into a single place [16, p. 2].

In [19] the causes of code duplication are categorized into four groups. Imposed duplication happens when a developer feels forced to duplicate code, for example when two target platforms use different programming languages. Inadvertent duplication happens when a developer rewrites functionality by accident. Impatient duplication happens when a developer feels like duplicating code is easier than reusing it. Finally, interdeveloper duplication happens when two or more developers write their own solutions to the same problems instead of reusing a common piece of code.

2.6 Functions

Martin [1] advises that functions should only do a single thing and be short, ideally just a few lines. Doing only one thing keeps the function in accordance with the Single Responsibility Principle. Having a single, clear goal also helps in keeping the name of the function descriptive. Large functions should be refactored into smaller ones, improving clarity as blocks of difficult-to-parse code get replaced by descriptive function names. [1] On the other hand, trivially small functions that do not see enough reuse should be removed and moved inline into the modules calling them. [21]

While short functions are generally said to be desirable and a boon to understandability [1, 25], several studies suggest that larger functions require proportionally fewer changes and are less prone to errors. A study by Basili and Perricone [26] found that the length of a function (up to 200 lines of code) is inversely correlative to the number of errors found within it — as the length and complexity of a function grow, there are proportionally fewer errors within it. Another study by Selby and Basili [27] found that while shorter functions contained more errors than long ones, they were significantly less expensive to fix.

Beck and Fowler [24] recommend using functions instead of comments to explain the intention of code. Existing comments within functions are often signs that the blocks of code are far enough apart semantically to be separated into their own functions. The names of the new functions then serve the same documenting purpose as the comments did before. [24]

Each function should only contain a single level of abstraction. High-level functions call lower-level functions to form their functionality. Martin recommends ordering functions in a top-to-bottom order within a class: high-

level functions at the top of the class call functions further down. [1]

Functions should have no side-effects, they should only depend on their inputs. Calling a function of an object instance should not change the state of that object unless it is clear from the function signature (its name and/or parameters). Side effects introduce temporal coupling: the order in which functions are called changes their outcome. Side effects also make testing tricky. [1] A function relying solely on properly abstracted inputs can be tested easily by providing fake mock objects in place of real objects [28, 29].

The number of arguments a function takes should be kept to a minimum (ideally zero) for clarity. Functions with zero arguments are called niladic functions, one argument monadic, two arguments dyadic and three arguments triadic. Functions with more than three arguments — polyadic functions — should be avoided. [1] Long lists of related arguments can instead be replaced by an argument object. If an argument's only job is to determine which block of code within the function gets run (a so-called flag argument), the function should instead be split into two explicit functions. [1, 24]

The more arguments a function has the more difficult it is to quickly parse when reading code. The same applies to writing tests for the code. Testing a niladic function is trivial, but a triadic function requires writing dozens of test cases to cover all possible inputs. [1]

To measure a function's complexity, a metric called cyclomatic complexity has been developed [30]. It is calculated from the function's control flow graph: a graph where blocks of code are represented by the nodes, and the edges represent different decision flows through them. For example, an if-statement would form a split in the graph. A function's cyclomatic complexity M is given by the function

$$M = E - N + 2$$

where E is the number of edges and N is the number of nodes in the control flow graph [30]. A simple function with no decision points (a single node with zero edges) would therefore have a cyclomatic complexity of 1. In other terms, cyclomatic complexity is equal to the number of decision points within the function plus one [26].

2.7 Meaningful names

Function, module and variable names should be descriptive, intention-revealing and pronounceable [1]. In an ideal situation, a programmer reading the code only needs to read the function's name to know that it does [24]. For example, a variable for storing the state of a tic-tac-toe game board should not be called "theList" or "array". Instead, the name should describe

the variable: “gameboard” or “boardstate”. Additionally, names should not contain disinformation, unnecessary context or jokes. The boardstate variable should not be named “boardStateList”: the additional context is both unnecessary in today’s world where programming IDEs (Integrated Development Environments) reveal and handle such information automatically, and has the risk of being inaccurate if the boardstate is ever refactored to not be a list. [1, pp. 17-30]

Stylistic choices in naming do not matter, as long as the names are consistent. Consistency is especially important in outward-facing API (Application Programming Interface), like public functions and their parameters, where the name ideally contains all the information their user needs about their functionality. [22, pp. 56-58] This also includes the actual module’s name, which should be explanatory of its contents. A programmer should not have to look at the class’s or function’s implementation to know what they do. [1] Many programming languages have style guides and naming guidelines that help in writing code consistent with the industry standard. For example, Microsoft’s guidelines for C# [31] rule that all identifiers should be in “PascalCase” with the exception of function parameters, which should be in “camelCase”.

2.8 Consistent style

As mentioned above in Section 2.7, consistency is more important than any specific stylistic choice. The rule, however, extends much further than naming conventions, applying to for example indentation and the use of white space and braces. [1, 32] As style can be very subjective, it is important to have a unified coding standard and style. This prevents arguments over stylistic choices, reduces unnecessary changes, and makes the code easier to parse. [22, p. 39]

According to Martin’s rules for clean code [1] and Fowler’s and Beck’s guidelines [24], class and function names should act as headlines: explaining their contents. Within them, related blocks of code should be grouped together: lines within functions and functions within classes. Additionally, a class’s functions should generally be sorted top to bottom by their level of abstraction such that the lowest-level functions are at the bottom. This way the top-level, outward-facing public functions are at the top and easy to find. For anyone reading the code, the functions tell a story that goes into more and more details the further they read. [1]

Automatic utilities for formatting code have been developed. They are applied either in the IDE or in a source control repository, taking a block of

code as an input and automatically producing a formatted output according to a specified set of rules. [32, 33]

2.9 Automated testing

Beck popularized the concept of test-driven development (TDD) in [2], presenting a loop of "red, green, repeat" — red and green meaning failing and passing tests, respectively. Martin describes the system further [3, p. 32] as a loop consisting of three rules. First, before writing production code, there needs to be a failing unit test. Second, that unit test must not be any longer than is needed for it to fail. Third, enough production code must be written to make the test succeed and no more. According to Martin, this loop should ideally last only a few minutes.

TDD makes development more predictable, allowing for more accurate time estimations and less stress on the developers [2, 3, 23]. It allows developers to make changes to code without the fear of it breaking. It alleviates the risk of small changes and refactorings. Large changes are equally risk-free when broken up into smaller ones as suggested in [24]. By following the aforementioned three-rule loop and by adhering to small incremental changes, manual testing and debugging become unnecessary as broken tests indicate broken code added in the previous loop. [3]

Tightly coupled functions are difficult to test. Ideally, the tests are minimal and test only one function, but they cannot if that function calls other functions or has side effects. TDD encourages good design by enforcing decoupling, leading to easily testable code [2]. Good tests act as documentation for the rest of the codebase. [3, p. 35] They should thus be treated like any other code: readable, simple and clean. [1]

The effectiveness of tests can be tested and measured in a variety of ways. Thomas and Hunt suggest [19] appointing a "saboteur", whose job is to introduce bugs in an effort to verify the tests. Analytical measurements also exist: coverage analysis, for example, can be used to track what portion of the code is being executed during the test. [19]

2.10 Refactoring

Refactoring is the process in which the code's structure is improved by rewriting and restructuring it without changing its behavior [16, 24]. Fowler [24] notes that while traditionally software has been meticulously designed beforehand in order to avoid having to rewrite any of it, in modern programming a

common practice is to write the functionality first and then refactor the code to meet quality standards. In the end, both methods aim to optimize time spent programming.

Refactoring involves reading code and changing it in small steps with the goal of improving its design, making it easier to understand and work with [23]. Working in small increments is recommended because it is desirable to keep the program in a functioning state as much of the time as possible [24]. Constant refactoring is one of the main rules of Extreme Programming (XP) [34, 35], a set of rules and best practices for agile software development. The authors acknowledge that it might lead to more work when implementing a feature, but state that it ensures that future features and changes are easier to implement [35].

Automated testing can be used to aid refactoring. In order to ensure that functionality remains the same before and after refactoring, thorough tests are often written. If the tests are well made, they will catch any errors and changes in behavior mistakenly introduced while rewriting code. [24].

Common refactoring procedures can be classified and named, for example, the refactoring "Extract Function" moves common functionality into a function [24, p. 106] and "Inline Singleton" is used to improve the way a class accesses another [23, p. 114]. These procedures are called refactorings [23, 24, 36]. Refactorings can be as small as renaming a variable or involve merging a whole module into another. Many refactorings can be described algorithmically in steps [23, 24]. This has allowed developers of programming languages and development software such as IDEs to integrate automated refactorings into their products. Most modern IDEs can rename all instances of a variable at once (even in comments) and perform the Extract Function refactoring with just one click, vastly reducing the time taken. [24, 37, p. 23]

On a more abstract and general level, the solutions implemented in refactorings are often called patterns. A pattern is a description of a solution to a common problem [23]. Complex systems are often formed out of repeated patterns and reused small components [5]. The goal of refactoring is to move the code towards positive patterns. Negative and harmful patterns are known as antipatterns. Antipatterns should be avoided, but are often used as examples of what not to do in literature and when teaching refactoring. [37]

Studies on the effectiveness of refactoring have produced varying results. In [36], researchers found that drastically changing the structure and design of code, while in the long-term beneficial, might require developers to reintroduce themselves to it. Developers appreciated refactored, clean code more, but were not always faster working with it as compared to flawed code. Working on refactored code however led to the developers fixing the root cause of the problem, following good practices, instead of applying a quicker but

superficial fix. A study on the effects of refactorings [38] found that while many refactorings improve code reusability, extendibility and flexibility, many produce no improvement at all and some even deteriorate these factors.

Chapter 3

The Last Cube

The Last Cube is an as of yet unreleased sci-fi puzzle game being developed by the independent games company Improx Games Oy. The author is a lead programmer in the project. In The Last Cube, the player controls a metallic cube-character who can move by rolling from one side to another. By collecting so-called stickers – colored shapes, see Figure 1 – onto the cube's faces and using the special powers of those stickers, the player solves puzzles and advances through levels. The planned release date for the game is Q1 of 2021. The game is going to be released on PC marketplaces such as Steam and on consoles, including Xbox One and Nintendo Switch.

Figure 1: The Cube with a yellow sticker on it.

The game contains 6 themes (corresponding to the six sticker colors) which consist of 3 levels each. Each level contains several puzzles. Completing a level unlocks the next level in its theme, and completing the whole theme unlocks the next one. The levels are accessible through a level selection area called The Hub, from which the player is free to enter any unlocked level or one of many bonus levels, which can be unlocked by collecting hidden relics within levels. The Hub also contains some secret areas and relics, acting as an alternative path of exploration in between levels.

3.1 Mechanics

In The Last Cube, the six stickers play a large role. There are six colors of stickers: blue, yellow, red, green, purple and orange. Each sticker is tied to its own area of the game where it and its powers are introduced and explored in conjunction with previously discovered stickers. At its core, the main mechanic of the game is rolling the cube in a way that a specific face of the cube lands on a specific tile (for example, a button) of the world while at the same time avoiding stepping on detrimental tiles such as water, which wipes away any stickers touching it. This cube-rolling mechanic has been explored in dozens of puzzle games before (both physical [39] and electronic), but the stickers' unique powers bring an innovative twist to it.

The stickers' powers are as follows. The blue sticker allows the player to spin the cube in place, easing navigation in tight areas. Activating the yellow sticker makes the cube dash forward four spaces, allowing for quicker movement and fitting into areas too tight to roll into. The red sticker creates ethereal tiles in the air under the cube, letting the player descend over gaps and other obstacles. The green sticker generates a temporary clone-cube on top of the player that, for example, can go through doors that require the main cube to hold open. The purple sticker allows for teleportation up to 3 spaces away, which helps in getting over small gaps. Finally, the orange sticker's power makes the player's cube jump onto its corner, shaking up all movement rules. While standing on its corners or edges, the cube can walk on tiny edges, walk over buttons without triggering them and much more.

Levels in the game are designed to get progressively harder and more complex towards the end of the game. Puzzle mechanics introduced in earlier levels are frequently reused later, often in combination with other mechanics. Early-game mechanics include buttons, lifts, and pistons. These are then mixed in with mechanics such as redirectable light beams, other cubes that move by themselves with simple movement rules and magnetic platforms that can shift gravity in an area. Dozens of puzzle mechanics combined with the

player's sticker-powers produce hundreds of possible combinations.

3.2 Code Structure

The Last Cube is being developed using the Unity game engine and C#. Its levels are of varying sizes and can contain hundreds of interacting components. The levels contain moving objects, which must be able to interact with all other objects on their path. Interactions within the world are not handled by game physics — in fact, every movement and collision is handled in code via a custom-made coordinate system that ensures accurate grid-based movement.

Taking advantage of the cubic nature of the player character and movement, the game world is divided into a 3D grid. Initially, the game world was implemented by storing multiple 3D arrays of booleans in memory. However, even a relatively small game world with a side length of 500 units would take hundreds of megabytes of memory, since each square needs multiple bytes of information. This led to problems with memory usage and bad performance.

The array system was later refactored to instead store information about the game world in octrees. Octrees are 3D tree-like data structures ideal for querying information about nearby objects in the world [40]. These octrees contain instances of a Coordinates data structure, which contains X, Y and Z values for the position, along with some helper functions. The octree is not a perfect solution to the problem, since every time an object moves, its position in the tree needs to be updated. In the current implementation, this means removing and reinserting them into the tree. Searching an octree has a worst-case complexity of $O(n * log(n))$ and a best-case complexity of $O(log(n))$, where n is the number of nodes in the tree [41]. This is significantly slower than the $O(1)$ of the previous implementation's array lookup, especially in combination with having to insert and remove entries for moving objects, but it stores the data in a much more memory efficient way. In profiling performance, the octrees have not thus far been a performance bottleneck.

To enable Coordinates to not always lie in a perfect grid, the Coordinates instances contain a reference to a CoordinateSpace object. Each CoordinateSpace hosts its own octree, which contains all Coordinates belonging to that CoordinateSpace. For most Coordinates, their CoordinateSpace is just a reference to the default: world-space. In some cases, however, the CoordinateSpace can be translated and rotated in the game world, moving Coordinates within as well. This is the case with, for example, gravity-altering magnetic platforms, which are just normal platforms that have their up-direction set to a different value. Upon initialization, if a Platform component's world rotation is set to a non-identity value (meaning the Platform

does not have the same rotation as world space), it creates a CoordinateSpace for itself that all its children then attach to.

Coordinates, and the octrees containing their data, are the building block of most components in The Last Cube. For the player cube to be able to walk on top of or to be blocked by a game object, it needs to have an entry in an octree set to either be "walkable" or "blocked", respectively. All areas meant for the player to walk on top of have a component attached to them that marks their space as walkable, while a door might have its area be blocked until it opens and becomes unblocked. Coordinates act as an interface between the relatively low-level octrees and other classes, offering helpful functions for manipulating the data within. This system enables emulating physics in a customizable, consistent and frame-perfect way.

Buttons, lifts, and other interactable features in the game world inherit from the abstract WorldFeature class, which in turn inherits from MonoBehaviour, Unity's base class for components. This class contains data common to all WorldFeatures, such as functions for initializing and interacting with Coordinates. It also contains many virtual functions these inheritor classes can override, for example, the StepOn and StepOff functions which Cubes call when stepping onto and off of the WorldFeatures. As of writing this, 76 classes inherit from WorldFeature.

The WorldFeature class implements the interface IToggleable. It contains the declarations for three functions: Toggle, ToggleOn, and ToggleOff. The functions ToggleOn and ToggleOff are implemented in WorldFeature to smartly call Toggle when necessary, but all three functions are virtual and can be overridden in subclasses. Toggling WorldFeatures on and off is a core feature in The Last Cube. When the player steps onto a button, the button calls Toggle on WorldFeatures it has been set to control, which then react in their own ways. Each WorldFeature can also define its default state: Lifts, for example, can start in either the top or bottom position depending on their use-case in the level.

3.3 Automated testing and continuous integration

The Last Cube was not developed using TDD. Tests exist now, but they only cover a small part of the codebase and mostly test in-game functionalities instead of individual functions. This means a broken test does not give perfect information about the location of the error: it only means that the end result of the test was not what was expected. For example, in a test in

which the cube uses the yellow sticker's power to dash forward over a button, the expected result would be that the cube dashes forward four squares and activates the button on its way. If at the end of the test, the button was not activated, the problem could be found in a number of places. The button could have an error preventing it from being toggled. The yellow sticker power might not correctly toggle WorldFeatures along its path. The coordinate system might be broken in a way that prevented the yellow sticker power from finding the button or stopped the cube from moving at all.

Unity has a built-in Test Runner feature [42], shown in Figure 2. It can be used to run two types of tests: edit mode tests and play mode tests. Edit mode tests are close to traditional unit tests. As the name suggests, they can be run without having to run the whole game. This means they are quick, making them ideal for the TDD loop [3]. Play mode tests, on the other hand, require the game to be running. When running the tests, Unity loads a special testing scene on top of the scene that the game uses, enabling the programmer to send commands to the game and make assertions about the resulting game state. Having to run the game to start testing, having to load a new scene for each test, and having to obey the in-game timescale make play mode tests much slower than edit mode tests. In The Last Cube, edit mode tests take under a second to run, while play mode tests take over three minutes to complete, making them too slow to run every few minutes like Martin suggests [3].

Tests were implemented comparatively late into the development of The Last Cube. A major incentive to have them was safety in refactoring, as [2] suggests. The game has too many features to test manually, and so much of the code is interdependent that refactoring one part could unexpectedly break an unrelated portion of it. Having tests as a snapshot of the state of the game enables developers to refactor without the fear of breaking or changing functionality.

To enable automated testing, some parts of the code had to be rewritten. Initially, player input was not abstracted behind an interface or even into a single class, but instead, raw values from Unity's input system were used wherever they were needed. This meant that there was no way to fake player input in the tests, and thus no way to automatically test, for example, the Cube's movement code. To solve this, input was abstracted behind a IControls interface (in accordance with the Dependency inversion principle [13]) that was then implemented in a PlayerControls class. This allows the test code to create fake substitutes (mock objects [28, 29]) for the interface using NSubstitute [43], a library included in Unity's testing package, that can then be plugged into the code.

The code snippet below shows how a mock object is created for the

Figure 2: Play mode tests in Unity's Test Runner

IControls interface using NSubstitute, fed into the cube's Controls member using a setter, and configured to always return a vector pointing right when queried for input. The cube queries the controls after its every step using the GetMoveVector function, asking for the value of the movement input axis. Using this configuration, the cube always receives Vector3.right, and moves right until it receives another movement vector. Vector3.zero can be used to reset the movement vector, making the cube stop moving.

```
IControls mockControls = Substitute.For<IControls>();
cube.Controls = mockControls;
mockControls.GetMoveVector().Returns(Vector3.right);
```

In order to be able to use the above configuration to move a predictable number of steps, the tests must be able to control how many steps the cube takes. This was implemented using a public Landed event in the Cube class.

This event is invoked every time the cube has finished taking a step. The tests can then subscribe to this event and use it to count the steps taken.

Another crucial component of testing The Last Cube was being able to set the Cube's stickers via a function. The Cube class has a public function GetFace for getting a specific face of the cube. The Face class's function SetSticker then allows the programmer to set the face's sticker. This makes the tests quicker, as the Cube no longer has to be manipulated to collect the stickers from the game world.

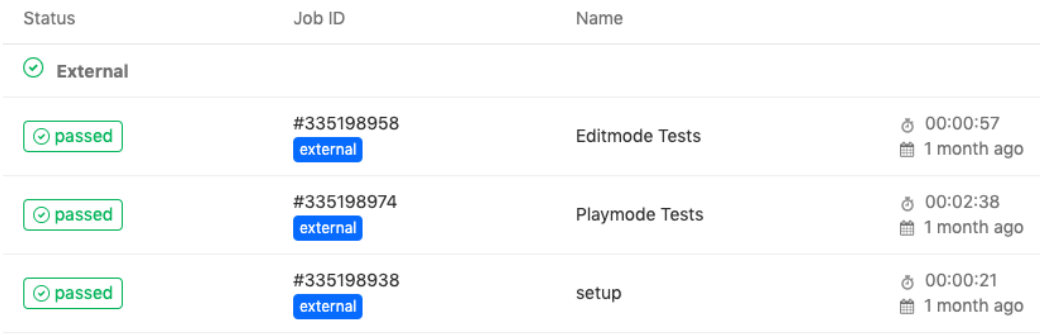
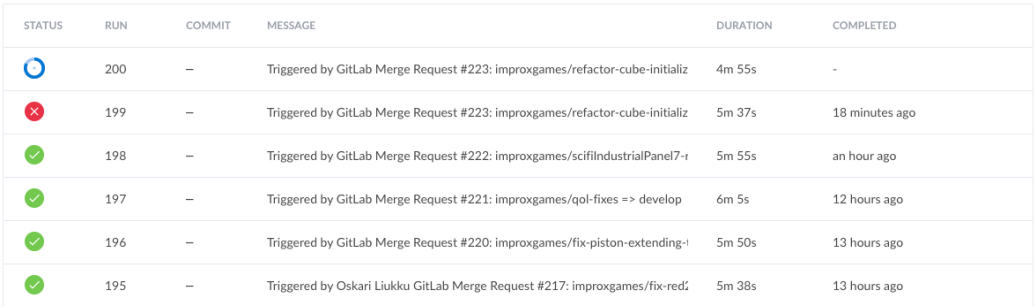


Figure 3: A pipeline in the source control system runs all tests after a merge request

Now that tests had been enabled in Unity, they could be run in between changing the code to check that the game still functioned as expected. An even more robust solution exists, though. Many modern source control providers, such as Github or Gitlab, allow developers to build testing pipelines within their systems. First, directly merging code into the main "develop" branch was disallowed. All new code has to be pushed in by issuing merge requests [44], which allow other developers to review the changes before they are made final, and crucially, allow these pipelines to run on the proposed changes. Figure 3 shows the testing jobs running in the merge request.

In The Last Cube's Gitlab repository, a pipeline was set up to run all tests whenever any developer issues a merge request, the full set of tests is run and merging is not allowed until the test suite has completed with no failing tests. Figure 4 demonstrates the automated testing system built with Jenkins, a configurable automation server, set up to run the tests on command. Once the merge request is accepted and pushed through into the main branch of the source control system, Jenkins [45] also automatically builds the game, allowing developers to download the finished files when necessary instead of manually building the game.



STATUS	RUN	COMMIT	MESSAGE	DURATION	COMPLETED
◌	200	–	Triggered by GitLab Merge Request #223: improxgames/refactor-cube-initializ	4m 55s	-
✕	199	–	Triggered by GitLab Merge Request #223: improxgames/refactor-cube-initializ	5m 37s	18 minutes ago
✓	198	–	Triggered by GitLab Merge Request #222: improxgames/scifiIndustrialPanel7-r	5m 55s	an hour ago
✓	197	–	Triggered by GitLab Merge Request #221: improxgames/qol-fixes => develop	6m 5s	12 hours ago
✓	196	–	Triggered by GitLab Merge Request #220: improxgames/fix-piston-extending-l	5m 50s	13 hours ago
✓	195	–	Triggered by Oskari Liukku GitLab Merge Request #217: improxgames/fix-redZ	5m 38s	13 hours ago

Figure 4: A pipeline in the source control system runs all tests after a merge request

This system of automated testing and continuous integration has lead to the development team noticing dozens of bugs in time before they have made their way into the main branch of the version control system. The system does have its faults too, though. The game code is not robust enough to return perfectly reliable results, causing tests to fail due to a minuscule deviation of timing. This, of course, is the fault of the game code, not the testing system, but these tiny deviations are not noticeable in gameplay and sometimes only happen on a single specific computer, making them difficult to fix. Nevertheless, implementing the system has saved more time and effort than it caused.

Chapter 4

Analysis of game code

In this chapter, the source code of the game The Last Cube is analyzed using various methods and software, and the implications of the results of the analysis are discussed. Finally, portions of the codebase are taken under closer scrutiny, and solutions to their problems are discussed.

4.1 Static analysis

Static analysis refers to the analysis of code in search of problems, done without running it, as opposed to dynamic analysis or testing, which is performed by running code [46]. The testing described in Section 2.9 is dynamic analysis. Most modern IDEs, such as Microsoft's Visual Studio, have built-in functionality [47] to perform static analysis for a selected portion of the codebase.

4.1.1 Coupling and complexity

In Table 5, a list of the top ten results of Visual Studio's static analysis are presented, sorted by the Class Coupling metric. Coupling was chosen instead of Maintainability Index or cyclomatic complexity because dozens of classes have a Maintainability Index of 100, making the cutoff point unclear and because the top results for cyclomatic complexity contain classes from third-party libraries. Visual Studio's analysis is based on the intermediate language (IL) version of the code. IL is a lower-level language that C#, a high-level language, is compiled to upon building. This means the "Lines of Code" metric is not based on the actual source code, but can still be used for relative scale.

Another popular static analysis tool, NDepend, can generate numerous

Class	Maintainability Index	Cyclomatic Complexity	Class Coupling	Lines of Code
LevelManager	71	106	94	451
RotationHintSystem	66	93	68	448
Cube	67	404	60	1686
HighlightTargets	60	53	52	443
OrangeStickerPower	55	123	51	806
PurpleStickerPower	59	94	45	510
PermanentButton	58	31	44	252
LevelNameReveal	56	19	43	226
RedStickerPower	59	52	39	309
Platform	67	97	38	444

Table 5: Visual Studio Code Metrics results, sorted by Coupling

useful graphs, tables and other visualizations for analyzing the complexity of code. NDepend also analyzes the IL version of the program [48]. Here NDepend was used with its default settings. Figure 5 shows a treemap visualizing the number of instructions and cyclomatic complexity in the codebase. The size of each square represents its instruction count, while the color is scaled from green to red based on the instructions' cyclomatic complexity. Many of the same classes can be seen in the treemap as were highlighted by Visual Studio's analysis.

Next, NDepend was used to generate coupling statistics for the code. Table 6 lists all types with TypeRank of 2.00 or more. TypeRank is a metric developed by Google for ranking pages in their search engine and is calculated based on how many pages link to the page [46]. NDepend uses PageRank to indicate how interdependent the type is, and thus how difficult it would be to change and how dangerous bugs in it could be. Afferent coupling measures the number of types that directly depend on the type. Efferent coupling is the opposite, measuring the number of types directly depended on by the type. [49]

This analysis uncovers a few new types compared to the previous methods. For example, WorldFeature is coupled to 110 other types, mostly via inheritance coupling. The table also highlights non-class types such as IToggleable (an interface implemented by WorldFeature and thus coupled to most of the same types) and StickerType, which is a widely used enum containing the six

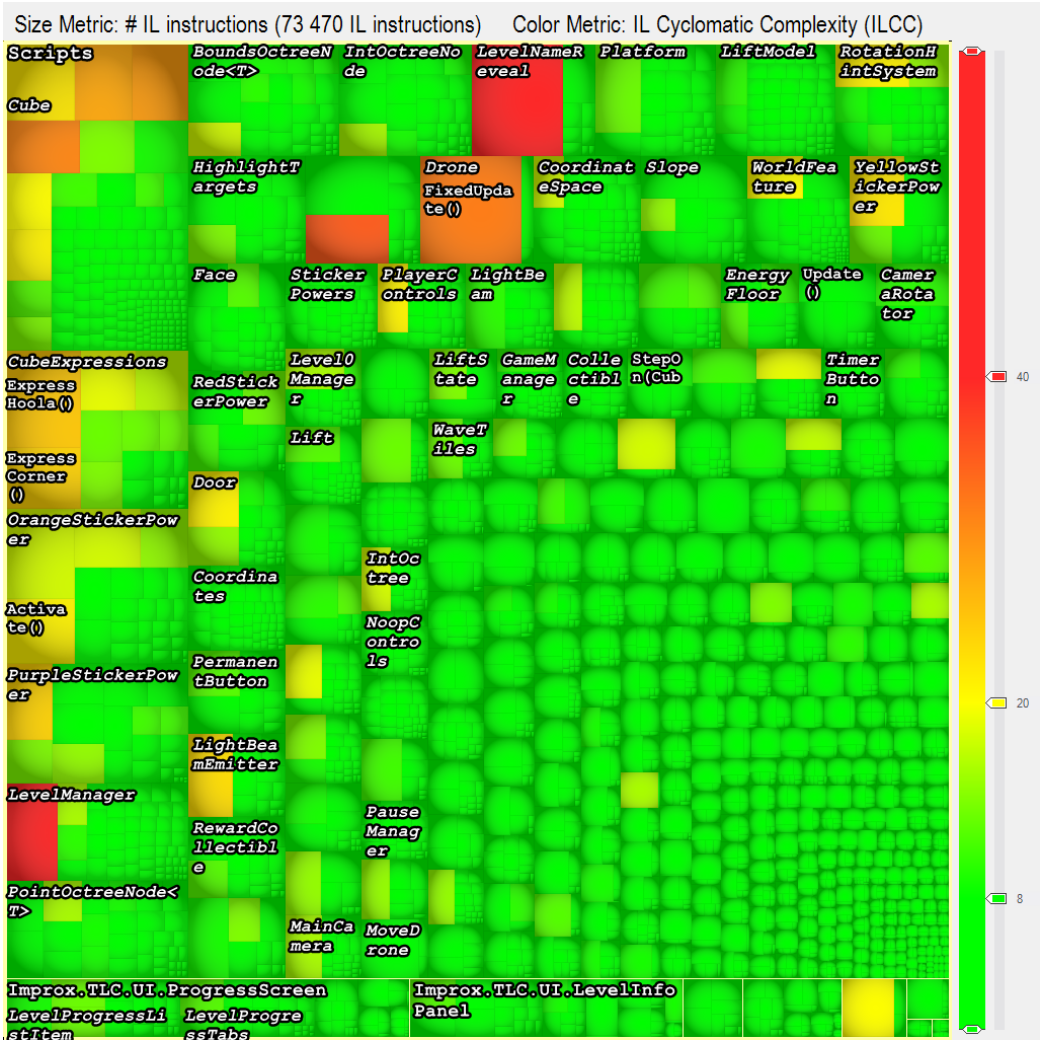


Figure 5: Treemap of IL instruction count and cyclomatic complexity in the code.

sticker colors. It is clear that any changes to these types would have to be cascaded to all types dependent on them.

NDepend also searches the code for issues based on a provided ruleset, looking for violations of principles of clean code, for example, deep inheritance trees and unused code. Overall, NDepend finds 3294 issues and estimates that fixing all of them would take 47 days of development time.

Type name	TypeRank	Afferent coupling	Efferent coupling
WorldFeature	9.29	110	42
Cube	7.12	100	65
IToggleable	7.09	86	1
Coordinates	4.81	68	25
StickerType	4.6	61	4
CoordinateSpace	4.56	60	38
CubeTime	2.95	44	5
MainCamera	2.63	40	29
FloorType	2.02	28	4

Table 6: NDepend code quality metrics. Types with type rank >2.00.

4.1.2 Duplication

To uncover code duplication and violations of DRY, a duplication finder utility was run on the codebase. The utility used was dupFinder by Jetbrains [50], which is a free command-line tool. The tool produces an XML report, which can then be prettified into HTML. A portion of the analysis is shown in Table 7, in a simplified and prettified form. The table excludes duplication found in third-party code.

The results of the dupFinder analysis show that most of the duplication found in the source code is found in tests. This is because many of the tests make nearly identical assertions, but are based on different initial arrangements. For example, the first four rows of Table 7 are from tests that are nearly identical:

```
var mockControls = Substitute.For<IControls>();
cube.Controls = mockControls;
powers.Controls = mockControls;

mockControls.UsePowerButtonDown += Raise.Event<System.Action>();

yield return null;

mockControls.BlueRotateLeftDown += Raise.Event<System.Action>();

yield return new WaitForSeconds(0.5f);

// Assert
```

Cost	Lines	File
366	58-76	Tests/BluePowerTests/BluePowerTests.cs
	91-107	Tests/BluePowerTests/BluePowerTests.cs
	122-140	Tests/BluePowerTests/BluePowerTests.cs
	155-173	Tests/BluePowerTests/BluePowerTests.cs
357	75-97	Scripts/Effects/MainMenuTiles.cs
	80-102	Scripts/Effects/ReactiveEmissionWaveTiles.cs
327	128-153	Tests/PermanentButtonTests/PermanentButtonTests.cs
	137-162	Tests/TimerButtonTests/TimerButtonTests.cs
293	158-175	Tests/LiftTests/LiftTests.cs
	275-292	Tests/LiftTests/LiftTests.cs
293	77-92	Tests/LiftTests/LiftTests.cs
	103-118	Tests/LiftTests/LiftTests.cs

Table 7: Simplified, partial output of dupFinder using default settings.

```
Assert.AreEqual(cube.GetFace(Vector3.forward).GetSticker(),
    StickerType.None);
Assert.AreEqual(cube.GetFace(Vector3.back).GetSticker(),
    StickerType.None);
Assert.AreEqual(cube.GetFace(Vector3.down).GetSticker(),
    StickerType.None);
Assert.AreEqual(cube.GetFace(Vector3.right).GetSticker(),
    StickerType.None);
Assert.AreEqual(cube.GetFace(Vector3.up).GetSticker(),
    StickerType.Blue);
Assert.AreEqual(cube.GetFace(Vector3.left).GetSticker(),
    StickerType.Blue);
```

This test asserts that upon receiving a “rotate left” input during the usage of the blue sticker’s power, the cube, in fact, rotates left. Another test makes sure that the rotation does not happen if a square next to the cube is blocked. The assertions are not exactly the same, but dupFinder still flags them as duplication. This is because dupFinder by default finds similar structures even if the variables and methods used are slightly different. The cost of duplication in the Cost column is calculated using a syntax tree and is similar to cyclomatic complexity. [50]

While tests should be held to the same standards as production code [1],

these cases of duplication are probably harmless. They are short and clear blocks of instructions and assertions. The alternative would be to create small functions for sending input to the cube and asserting its position and stickers. This would mean the tests are no longer encapsulated in single functions. On the other hand, Martin [1] suggests doing exactly that: having an evolving, "domain-specific" testing API for making the tests more clean and readable. Furthermore, the duplicated code is not cohesive enough to group into a single function. Following Martin's advice, the assertions could be abstracted into a helper function:

```
void AssertSticker(Cube cube, Vector3 faceDirection, StickerType
    sticker) {
    Assert.AreEqual(cube.GetFace(faceDirection).GetSticker(), sticker);
}

...

AssertSticker(Vector3.forward, StickerType.None);
AssertSticker(Vector3.back, StickerType.None);
AssertSticker(Vector3.down, StickerType.None);
AssertSticker(Vector3.right, StickerType.None);
AssertSticker(Vector3.up, StickerType.Blue);
AssertSticker(Vector3.left, StickerType.Blue);
```

This simplifies the assertion code considerably, and the abstraction moves the test code's coupling with Cube's functions into a single function. After this refactoring, dupFinder still reports the same lines as duplicates, but with a lower cost of 270.

To better analyze production code, dupFinder was run once more with the option -e="**Tests.cs", making it exclude any files ending with the characters "Tests.cs", which applies to all test files. Partial results of the analysis have been collected in Table 8. The filtered results reveal duplication in scripts handling graphics settings, lifts and "wave tiles". The duplication in the graphics settings is caused by a refactoring of the class during which the old file was mistakenly never deleted. LiftStateExtending and LiftStateRetracting are state-machine states which contain the exact same code, which should be refactored, for example to their base class LiftState. Wave tiles are tiles that can move in choreographed patterns or in reaction to a major game event. The movement code in the two classes is very similar and uses complex mathematical calculations, and should thus be refactored into a base class or a utility class.

Cost	Lines	File
357	75-97	Scripts/Effects/MainMenuTiles.cs
	80-102	Scripts/Effects/ReactiveEmissionWaveTiles.cs
244	19-38	Scripts/Essential/GraphicsManager.cs
	43-62	Scripts/UI/Settings/GraphicsOptionsMenu.cs
170	25-46	Scripts/UI/Settings/Graphics/DisplaymodeSetting.cs
	63-84	Scripts/UI/Settings/Graphics/DisplaymodeSetting.cs
170	57-71	Scripts/WorldFeatures/Lift/LiftStateExtending.cs
	55-69	Scripts/WorldFeatures/Lift/LiftStateRetracting.cs
168	44-63	Scripts/Essential/GraphicsManager.cs
	68-87	Scripts/UI/Settings/GraphicsOptionsMenu.cs

Table 8: Simplified, partial output of dupFinder, excluding test files.

4.1.3 Code style

ReSharper is a tool made by JetBrains for code analysis and refactoring. It integrates into Visual Studio as an extension and is free-to-use for student work. ReSharper offers dozens of functionalities, one of them being the inspection of issues in the project.

Running ReSharpers "Code Issues in Current Project" option on The Last Cube's source code produces a list of 4512 issues, 3192 of them having the severity of "warning" and the rest being suggestions. Figure 6 is a screenshot of these results. The issues have been grouped by their category.

The "Clean Code" category contains mostly warning about overly complex expressions and excessively deeply nested blocks of code. It also finds places where there are too many chained references, violating the Law of Demeter [26], and correctly identifies functions with flag arguments (boolean arguments that change the behavior of the function [1]) that break the Single Responsibility Principle.

The "Common Practices and Code Improvements" identified statements where static members were accessed via a derived type: places where GameObject.Destroy was used unnecessarily instead of the shorter Destroy. "Constraints Violations" found violations of ReSharper's default naming rules, asking for example that fields that are to be serialized in the Unity editor be written in lowercase without a preceding underscore, which is not in line with

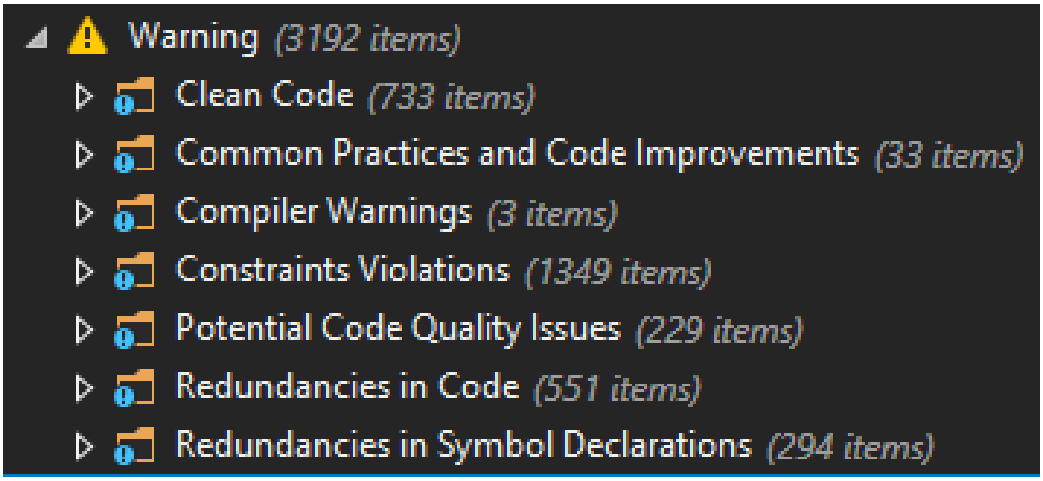


Figure 6: List of code issues in the source code with the severity of "warning".

the team's agreed-upon naming conventions.

The "Potential Code Quality Issues" category contains issues such as fields that are never assigned to and should thus be removed and places where a null reference is not handled and are thus vulnerable to errors. The "Redundancies in Code" category found unnecessary import statements and values that are never used. Finally, the "Redundancies in Symbol Declarations" found fields that were being initialized with their default values such as float Foo = 0, which is unnecessary since 0 is the float type's default value [51].

4.2 General analysis

The source code of The Last Cube follows around half of the aforementioned agile programming practices. It follows a consistent style thanks to the automatic formatting functionalities of IDEs. It partially follows the principles of TDD. Naming rules, such as functions starting with a capital letter, have been agreed upon and are followed consistently. The SOLID principles, however, are mostly ignored: most functions and classes are long and do more than one thing, and interfaces and other abstractions are only utilized in a few places. Thus, coupling is rampant since classes directly reference each other.

The fact that the WorldFeature class is used as a base class for over 70 other components severely breaks the Interface segregation principle [12]. Many of the functions found in WorldFeature have been raised there to be

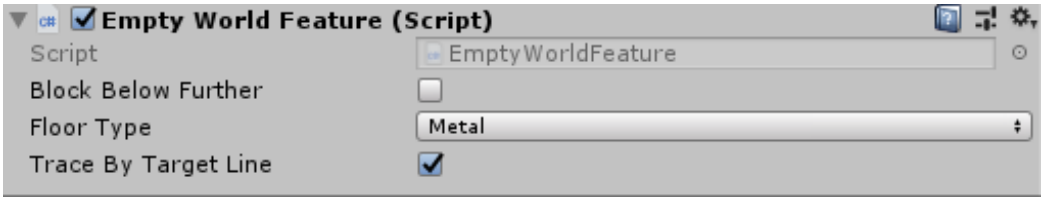


Figure 7: An empty component inheriting from WorldFeature.

shared by some of the child classes, but most do not need them. This becomes even more evident when viewing a component inheriting from WorldFeature in the Unity editor as in Figure 7. The component is an empty C# script that inherits WorldFeature. WorldFeature pollutes its interface with public variables such as FloorType (used for determining the sound played when a cube steps onto the object), which is useless for many WorldFeatures.

One Unity-specific problem is the abuse of script execution order. Unity allows developers to override the otherwise undefined order in which scripts are executed. Normally, objects are initialized in the Awake lifecycle method, which happens for all objects before the Start lifecycle method is called. Start can then be used to establish relationships between GameObjects, knowing that they have had time to initialize. For some situations, this is not enough, and a programmer has to make sure one script has fully initialized before another can access it. The Script Execution Order window inside the Unity editor (Figure 8) allows specifying an order of execution. The number shown is relative and used for sorting the list.

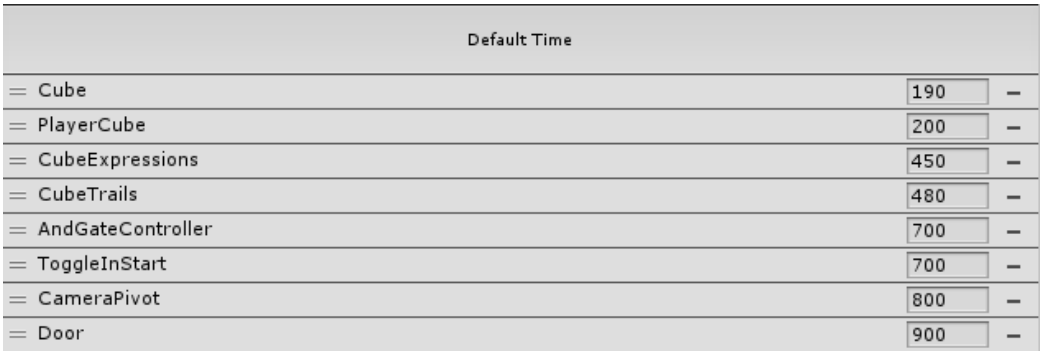


Figure 8: Partial Script Execution Order for The Last Cube.

This kind of forced time-relationship is an extreme form of temporal coupling – the modules must be executed in a specific order to function.

Furthermore, that order is enforced not via code, but through the game engine. While this is a quick way to correct the problem, it often causes more errors than it fixes. Now that one script has been set to execute before any others, the others have to wait for it, making start-up times slightly longer. Then, when another script needs to execute before any others, it too must be added to the Script Execution Order, further locking down the initialization order. Ideally, problems with execution order would be handled by calling the scripts in the specific order from an initialization function, perhaps by utilizing a factory pattern [7, 23]. That way, the order is much more flexible and visible to developers.

4.3 Case: LevelManager

The LevelManager class handles loading and unloading game levels. It consists of two main functions: LoadSceneAsync and ClearLevel. The class also contains multiple small functions for calculating information about levels, such as their index and theme. This section focuses on the ClearLevel function as it is the main source of coupling in the class. Isolated results for the function using Visual Studio's metrics can be seen in Table 9.

Member	Cyclomatic Complexity	Coupling	Lines of code
ClearLevel	1	44	33

Table 9: Visual Studio's code metrics [47] for the ClearLevel function.

ClearLevel is a large static function that was written to clear static state throughout the game's code whenever the scene changes. It is called in LoadSceneAsync after all current scenes have been unloaded and before the loading of the new scene starts. Many of WorldFeature's subclasses follow the ideology of having a static list inside them that instances of the class get added to. This static list enables quickly fetching a list of all instances of the class instead of using Unity's built-in functions (such as FindObjectsOfType), which are often slow [52] and cause duplication. Since these lists are static, they are not cleared when the level changes or when the game shuts down within the Unity editor, and must thus be cleared manually:

```
public static void ClearLevel()
{
        BeamHole.ClearAll();
```

```
        BeamMirror.ClearAll();
        BreakableBox.All.Clear();
        ButtonFeature.Buttons.Clear();
        CoordinateSpace.Reinitialize();
        Cube.Cubes.Clear();
        PlayerCube.PlayerCubes.Clear();
        Cube.ResetPlayerCube();
        CubeTime.Reset();
        FullResetSquare.All.Clear();
        GameManager.ResetRewardsCollected();
        GreenCubeKillGate.All.Clear();
        IceSquare.All.Clear();
        LightBeamEmitter.ClearAll();
        LightBeamReceiver.All.Clear();
        PhysicsObject.All.Clear();
        RewardCollectible.All.Clear();
        RewardCollectible.CollectedThisFrame.Clear();
        WallHole.ClearAll();
        WaterSquare.All.Clear();
        WaveTiles.Tiles.Clear();
        WaveTiles.Waves.Clear();
        WhitePlane.WhitePlanes.Clear();
        WorldFeature.ClearAllFeatures();
        WorldSticker.All.Clear();
        Slope.All.Clear();
        PuzzleBounds.All.Clear();
        SwitchPlayerCubeUI.Reset();
        RotationHintSystem.EnableHints = true;
}
```

There are many problems with the current implementation. Usage of the ideology is inconsistent: the lists do not have a common, clear interface. Some classes use the name All for the list, some use a term relevant to the context such as WaveTiles.Tiles, WaveTiles.Waves, PlayerCube.PlayerCubes. This more descriptive naming allows for multiple lists in the same class, such as in the case of WaveTiles. The accessibility level of the list is not consistent either: BeamMirror's list is not public and is cleared using a helper function instead of directly calling Clear on the list. The functions also violate the Dependency Inversion Principle [13]; LevelManager is a high-level class, but the function accesses dozens of lower-level functions in lower-level classes. Ideally, the classes should be separated by abstraction, and the dependency should be inverted.

There are multiple ways to fix these problems. First, LevelManager could have a public event that it invokes when all levels have been unloaded. Interested clients, such as the current classes listed in ClearLevel, could then subscribe to the event and reset themselves. The subscriber class would have

to make sure to only subscribe to the event once and to eventually unsubscribe at a proper time.

Second, the classes could implement the resetting in the OnDestroy function. OnDestroy is a built-in function in Unity that gets called on components whenever they are destroyed. Again, this function is called on every instance of the class, so the implementation would have to be written in a way that does not proceed with the resetting if another instance has already reset the state. Alternatively, the classes that only require clearing of the list to be reset, the instances could remove themselves from the list in OnDestroy. This, however, would cause the list to be accessed for every object in it, possibly causing performance problems. Another problem with this solution is that OnDestroy is only called on classes that inherit from Object, which not all of the classes in ClearLevel do.

Third, the classes could implement a common interface, named for example "IResettable". The interface would declare a (possibly static) Reset function, which could then be called to reset the state. This would hide the resetting behind an abstraction, which is desirable from the point of view of the Dependency inversion principle [13] as mentioned above. The problem with this solution is the difficulty of calling the Reset function. The most straightforward solution would require the classes to add themselves to a list of IResettables in LevelManager, which could then be iterated over in ClearLevel. IResettables could also have their own static manager class, containing a static list of them along with functions for adding and removing them and resetting them.

A fourth option could be implementing the Publish-subscribe pattern [53]. This would involve creating a central messaging channel through which events can be listened to and sent. This would allow nearly full decoupling. The pattern would, however, introduce more problems than it fixes. It pushes the messaging into a public channel, completely breaking encapsulation and introducing a large error vulnerability. The pattern is also a massive system that, after this refactoring, is only used in one situation whereas everywhere else other methods of communication are used. A larger refactoring would be required to bring everything under the common messaging channel.

Out of these options, the first seems the best. It offers a reversal of dependencies where the higher-level function does not concern itself with lower-level classes. It allows hiding the resetting functions as private functions, stream-lining the class's interfaces. Further, it follows a pattern used throughout the codebase: events. In fact, the LevelManager class already has multiple events that are triggered throughout the scene loading process:

```
public event Action<string> OnLoadSceneStartedPreFade;
public event Action<string> OnLoadSceneStartedPostFade;
public event Action<string> OnLoadSceneFinishedPreFade;
public event Action<string> OnLoadSceneFinishedPostFade;
public event Action<string, string> OnSceneLoaded;
```

The names of these events do not match Microsoft's naming guidelines [54], nor does the way in which they are declared and invoked follow Microsoft's suggested pattern [55]. Using Action instead of the suggested EventHandler pattern means that whenever the parameters of the event change, all listeners must be manually modified to match the new parameters. This severely violates the Single responsibility principle [8]. The usage and naming conventions of events in the codebase are heavily inconsistent, and refactoring them is out of the scope of this section.

To perform the refactoring, a sixth event is added: OnAllLevelsUnloaded. The event does not need parameters. All 25 scripts referenced in ClearLevel must then be modified to subscribe to the event with a resetting function. For example, the Cube script is modified thusly:

```
private void Awake()
{
    ...
    LevelManager.Instance.OnAllLevelsUnloaded += Reset;
}

private void Reset()
{
    Cubes.Clear();
    LevelManager.Instance.OnAllLevelsUnloaded -= Reset;
}
```

Awake is one of Unity's built-in lifecycle functions [56]. It gets automatically called when the game launches and is used instead of the constructor for initialization. When the game starts, the script subscribes to the OnAllLevelsUnloaded event with its Reset function, and unsubscribes once its state has been reset. Removing the subscription is important because Awake will be called again once the new level loads. Note, that since Awake is called on every component inheriting from Object, the above code is executed for every Cube in the game. The Cubes list is therefore cleared multiple times. This can, however, be ignored, since clearing an empty list is a trivial operation.

Components that do not inherit from MonoBehaviour do not have access to Unity's lifecycle functions. An example of this is the static CubeTime class, a custom time implementation. To refactor it, a static constructor is

added:

```
static CubeTime()
{
    LevelManager.OnAllLevelsUnloaded += Reset;
}

private static void Reset()
{
    ...
}
```

In C#, static constructors are automatically called before any static members of the class are called [51], ensuring that the subscription always happens in time. As both the event and the class are static, unsubscribing is not necessary since the constructor will not get called again until the static context gets reset when the game is shut down.

The thorough collection of tests set up for The Last Cube made this refactoring trivial. After changing all classes, all tests were run, revealing errors in multiple places where the resetting function did not reset all necessary variables. Those errors were fixed and the tests ran again until they all passed without errors.

Once all dependent classes have been refactored using the same pattern, the ClearLevel function can be removed completely. Instead of calling it, the OnAllLevelsUnloaded is simply invoked, informing all of its subscribers and causing them to reset.

	Old score	New Score
Maintainability	71	74
Cyclomatic Complexity	106	106
Class Coupling	94	52
Lines of Code	451	419

Table 10: Visual Studio's code metrics [47] for LevelManager before and after refactoring.

After performing this refactoring, the metrics for the codebase were recalculated. The results are shown in Table 10. As expected, the amount of class coupling decreased by nearly 50 %. At the same time, the maintainability index of the class improved slightly, and the amount of code in the class was reduced.

Chapter 5

Discussion

As explained in Chapter 2, software development is all about optimizing time spent writing and reading code versus the results achieved. The goal of many of the methodologies discussed in this thesis is to spend more time designing and refactoring segments of code which in the end are efficient to expand upon, instead of sloppily writing code that quickly accumulates technical debt and becomes impossible to manage. This is very similar to traditional ideologies such as the "waterfall model", which is based on the fact that fixing mistakes in the early design stages is much easier and thus cheaper than in later stages of production [57, 58].

As attempting to follow many of the methodologies discussed in this thesis can initially slow down development, some developers opt to ignore them. Even within the rules, there are some discrepancies and caveats. The rule of DRY teaches (if taken literally) to never repeat anything, while Fowler's Rule of Three [24] downplays its importance, claiming that a single repetition is still acceptable. This is an attempt to prevent programmers from wasting time abstracting away chunks of code that are only reused once – a refactoring that will probably take more time than it is worth.

Similarly, the rules can be ignored in situations where the time saved by them does not get to come into play, such as in game jams (events where developers make games in a short timeframe) or when making prototypes. In such situations where the end product is more important than the disposable codebase, the front-loaded cost of having to design and write clean code is too great. By focusing on rapid prototyping instead of clean software design, programmers are able to achieve more in the budgeted time.

While the results of Chapter 4 will be taken into consideration in the future of Improx Games, refactoring the codebase to correct the issues and to match the rules presented in this thesis would take too long. Existing code will probably not be changed, except while refactoring it for another

reason. The lessons learned have however been shared amongst the developers and will be considered while writing additional code. The testing workflow discussed in Chapter 3 has thus far been a tremendous help in refactoring and will be expanded upon by following TDD where possible.

One of the most important lessons the author learned during this research was the meaning behind the SOLID ruleset. SOLID is often touted as the cornerstone of clean code, but not until reading into it deeper did the rules' importance become clear. Specifically, the Liskov substitution and Interface segregation principles were unfamiliar but turned out to be great lessons and things to consider while writing code programs in the future. They offer solutions for situations which in the past seemed wrong but had no clear answer. The current code of The Last Cube violates them repeatedly. On the other hand, the Single-responsibility principle and DRY are rules that were relatively familiar to the development team but were still disregarded in the favor of quicker prototyping. The prototype code was never refactored and causes code that was built to extend it to have to break the same rules. A better alternative, perhaps, would have been to either plan everything thoroughly before starting (something which is rare and difficult in game development), or to restart from scratch once the gameplay of the rough prototype was satisfactory.

Chapter 6

Conclusions

Many people have opinions on what constitutes as good, clean code, and hundreds of rules, guidelines and methodologies have been discussed and developed to help produce code deserving of that label. The general consensus, however, seems to be that code should first and foremost be flexible, and easy to extend and to work on. While some of the rules listed in Chapter 2 cause development to slow down temporarily, they also allow it to continue sustainably and to keep the cost of adding or changing features relatively constant. Following the principles of agile software development is suitable for long-term and larger-scale projects, where the timeline of the project is long enough for the benefits to start outweighing the downsides.

The game under examination in Chapters 3 and 4, The Last Cube, has been in development since 2017, and its developer Improx Games has grown considerably in that time. Due to a lack of code quality assurance and overall skill early in its development lifetime, modern additions to the codebase have to be built on out-of-date and overly complex solutions. Static code analysis in Chapter 4 found dozens of violations of basic rules, for example duplication of whole functions, unnecessary coupling and a general lack of use of interfaces and other abstractions. Moreover, some of the violations found could be fixed relatively effortlessly, as seen in Section 4.3, where a function for clearing game state was broken up and delegated to its clients using an event, reducing class coupling by nearly half. Had the aforementioned methodologies been considered while writing the code, these situations could have been avoided and the accrued technical debt could have been managed better.

While the clean code methods are not suitable for every project, and are rarely fully followed, they are crucial to keep in mind while writing new code. The examples and analysis of game code in this thesis show the effects of not utilizing them to their fullest and act as motivators for the developers to maintain good code quality. Especially in productions with multiple

programmers, the clarity of code not only facilitates reuse and refactoring but also eases communication and reduces time spent parsing the code.

Bibliography

[1] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2009, 431 pp., ISBN: 978-0-13-235088-4.

[2] K. Beck, *Test-driven development: by example*, ser. The Addison-Wesley signature series. Boston: Addison-Wesley, 2003, 220 pp., ISBN: 978-0-321-14653-3.

[3] R. C. Martin, "Professionalism and test-driven development", *IEEE Software*, vol. 24, no. 3, pp. 32–36, May 2007, ISSN: 0740-7459. DOI: 10.1109/MS.2007.85. [Online]. Available: http://ieeexplore.ieee.org/document/4163026/ (visited on 11/27/2019).

[4] R. C. Martin, *The clean coder: a code of conduct for professional programmers*. Upper Saddle River, NJ: Prentice Hall, 2011, 210 pp., ISBN: 978-0-13-708107-3.

[5] G. Booch, Ed., *Object-oriented analysis and design with applications*, 3rd ed, The Addison-Wesley object technology series, OCLC: ocm80020116, Upper Saddle River, NJ: Addison-Wesley, 2007, 691 pp., ISBN: 978-0-201-89551-3.

[6] H. G. Koller, "Effects of clean code on understandability: An experiment and analysis", Master's Thesis, University of Oslo, Oslo, 2016, 100 pp.

[7] R. C. Martin, "Design principles and design patterns", p. 34, 2000.

[8] R. C. Martin, *Agile software development: principles, patterns, and practices*, ser. Alan Apt series. Upper Saddle River, N.J: Prentice Hall, 2003, 529 pp., ISBN: 978-0-13-597444-5.

[9] R. C. Martin, "The open-closed principle", *The C++ Report*, Jan. 1996. [Online]. Available: https://web.archive.org/web/20150905081105/http://www.objectmentor.com/resources/articles/ocp.pdf (visited on 11/14/2019).

[10] B. Liskov, "Data abstraction and hierarchy", in *OOPSLA 1987*, 1987.

[11] R. C. Martin, "The liskov substitution principle", *The C++ Report*, Mar. 1996. [Online]. Available: https://web.archive.org/web/20150905081111/http://www.objectmentor.com/resources/articles/lsp.pdf (visited on 11/14/2019).

[12] R. C. Martin, "The interface segregation principle", 1996. [Online]. Available: https://web.archive.org/web/20150905081110/http://www.objectmentor.com/resources/articles/isp.pdf (visited on 11/14/2019).

[13] R. C. Martin, "The dependency inversion principle", Jun. 1996. [Online]. Available: https://web.archive.org/web/20150905081103/http://www.objectmentor.com/resources/articles/dip.pdf (visited on 11/14/2019).

[14] E. Gamma, Ed., *Design patterns: elements of reusable object-oriented software*, Addison-Wesley professional computing series, Reading, Mass: Addison-Wesley, 1995, 395 pp., ISBN: 978-0-201-63361-0.

[15] J. Eder and M. Schrefl, "Coupling and cohesion in object-oriented systems", May 23, 1995.

[16] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring - improving coupling and cohesion of existing code", in *11th Working Conference on Reverse Engineering*, ISSN: 1095-1350, Nov. 2004, pp. 144–151. DOI: 10.1109/WCRE.2004.33.

[17] K. Lieberherr, I. Iioll, and A. Riel, *Abstract Object-Oriented Programming: An Objective Sense of Style*.

[18] D. Bock, "The paperboy, the wallet, and the law of demeter", [Online]. Available: https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf (visited on 12/16/2019).

[19] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Reading, Mass: Addison-Wesley, 2000, 321 pp., ISBN: 978-0-201-61622-4.

[20] L. Briand, J. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems", *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, Feb. 1999, ISSN: 00985589. DOI: 10.1109/32.748920. [Online]. Available: http://ieeexplore.ieee.org/document/748920/ (visited on 11/24/2019).

[21] L. Constantine, "Structured design", *Ibm Systems Journal*, 1979. [Online]. Available: https://www.academia.edu/7878229/Structured_Design (visited on 11/26/2019).

[22] M. McShaffry, *Game coding complete*, 4th ed. Boston, MA: Course Technology, Cengage Learning, 2013, 911 pp., ISBN: 978-1-133-77657-4.

[23] J. Kerievsky, *Refactoring to patterns*, ser. Addison-Wesley signature series. Boston: Addison-Wesley, 2005, 367 pp., ISBN: 978-0-321-21335-8.

[24] M. Fowler and a. O. M. C. Safari, *Refactoring: Improving the Design of Existing Code*. 2018, OCLC: 1099553884. [Online]. Available: `https://www.safaribooksonline.com/library/view/-/9780134757681/?ar` (visited on 10/30/2019).

[25] S. McConnell, *Code complete*, 2nd ed. Redmond, Wash: Microsoft Press, 2004, 914 pp., ISBN: 978-0-7356-1967-8.

[26] E. H. Sibley, V. R. Basili, and B. T. Perricone, *Software Errors and Complexity: An Empirical Investigation*.

[27] R. Selby and V. Basili, "Analyzing error-prone system structure", *IEEE Transactions on Software Engineering*, vol. 17, no. 2, pp. 141–152, Feb. 1991, ISSN: 2326-3881. DOI: `10.1109/32.67595`.

[28] D. Thomas and A. Hunt, "Mock objects", *IEEE Software*, vol. 19, no. 3, pp. 22–24, May 2002, ISSN: 1937-4194. DOI: `10.1109/MS.2002.1003449`.

[29] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: Unit testing with mock objects", 2001.

[30] T. McCabe, "A complexity measure", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, ISSN: 2326-3881. DOI: `10.1109/TSE.1976.233837`.

[31] KrzysztofCwalina. Capitalization conventions, Microsoft Docs, [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/capitalization-conventions` (visited on 11/24/2019).

[32] S. P. Reiss, "Automatic code stylizing", in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*, Atlanta, Georgia, USA: ACM Press, 2007, p. 74, ISBN: 978-1-59593-882-4. DOI: `10.1145/1321631.1321645`. [Online]. Available: `http://portal.acm.org/citation.cfm?doid=1321631.1321645` (visited on 11/26/2019).

[33] G. E. Jones, A. J. Owen, and M. Whitehead, "Automatic formatting of computer program source code", U.S. Patent 9557987B2, Jan. 31, 2017. [Online]. Available: `https://patents.google.com/patent/US9557987B2/en` (visited on 11/26/2019).

[34] D. Wells. (2009). Extreme programming rules, [Online]. Available: http://www.extremeprogramming.org/rules.html (visited on 10/30/2019).

[35] K. Beck, *extreme programming eXplained: embrace change*. Reading, MA: Addison-Wesley, 2000, 190 pp., ISBN: 978-0-201-61641-5. [Online]. Available: https://learning.oreilly.com/library/view/extreme-programming-explained/0201616416/.

[36] E. Ammerlaan, W. Veninga, and A. Zaidman, "Old habits die hard: Why refactoring for understandability does not give immediate benefits", in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, ISSN: 1534-5351, Mar. 2015, pp. 504–507. DOI: 10.1109/SANER.2015.7081865.

[37] W. J. Brown, R. C. Malveau, H. W. M. Iii, T. J. Mowbray, J. Wiley, R. Ipsen, and T. Hudson, "Refactoring software, architectures, and projects in crisis", p. 157,

[38] M. Alshayeb, "Empirical investigation of refactoring effect on software quality", *Inf. Softw. Technol.*, vol. 51, no. 9, pp. 1319–1326, Sep. 2009, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.04.002. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2009.04.002 (visited on 11/26/2019).

[39] K. Buchin, M. Buchin, E. D. Demaine, M. L. Demaine, D. El-Khechen, S. P. Fekete, A. Schulz, P. Taslakian, and C. Knauer, "On rolling cube puzzles", p. 30,

[40] D. Meagher, *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. Oct. 1, 1980.

[41] S. Narasimhan, R.-p. Mundani, H.-j. Bungartz, and T. U. München, *An Octree- and A Graph-Based Approach to Support Location Aware Navigation Services*.

[42] Unity Technologies. Unity - manual: Unity test runner, Unity, [Online]. Available: https://docs.unity3d.com/2019.1/Documentation/Manual/testing-editortestsrunner.html (visited on 11/29/2019).

[43] NSubstitute. NSubstitute: A friendly substitute for .NET mocking libraries, NSubstitute: A friendly substitute for .NET mocking libraries, [Online]. Available: https://nsubstitute.github.io/ (visited on 11/29/2019).

[44] Gitlab. Merge requests, [Online]. Available: https://docs.gitlab.com/ee/user/project/merge_requests/ (visited on 01/21/2020).

[45] Jenkins. Jenkins, Jenkins, [Online]. Available: https://jenkins.io/index.html (visited on 11/29/2019).

[46] B. Wichmann, A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W. R. Marsh, "Industrial perspective on static analysis", *Software Engineering Journal*, 1995.

[47] jillre. Calculate code metrics - visual studio, Microsoft Docs, [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values (visited on 11/28/2019).

[48] gewarren. What is managed code?, Microsoft Docs, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/managed-code (visited on 11/28/2019).

[49] NDepend. Code metrics definitions, NDepend, [Online]. Available: http://ndepend.com/docs/getting-started-with-ndepend (visited on 11/28/2019).

[50] Jetbrains. dupFinder command-line tool, Resharper, [Online]. Available: https://www.jetbrains.com/help/resharper/dupFinder.html (visited on 12/04/2019).

[51] B. Wagner. Static constructors - c# programming guide, Microsoft Docs, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-constructors (visited on 12/14/2019).

[52] Unity Technologies. Unity - scripting API: Object.FindObjectsOfType, Unity, [Online]. Available: https://docs.unity3d.com/ScriptReference/Object.FindObjectsOfType.html (visited on 12/13/2019).

[53] G. Hohpe and B. Woolf, *Enterprise integration patterns: designing, building, and deploying messaging solutions*, ser. The Addison-Wesley signature series. Boston: Addison-Wesley, 2004, 683 pp., OCLC: ocm52901145, ISBN: 978-0-321-20068-6.

[54] KrzysztofCwalina. Names of type members, Microsoft Docs, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-type-members (visited on 12/14/2019).

[55] gewarren. Handling and raising events, Microsoft Docs, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/events/ (visited on 12/14/2019).

[56] Unity Technologies. Unity - manual: Order of execution for event functions, Unity, [Online]. Available: https://docs.unity3d.com/Manual/ExecutionOrder.html (visited on 12/14/2019).

[57] S. McConnell, *Rapid Development*. Microsoft Press, 1996, OCLC: 939254863.

[58] B. Boehm and P. Papaccio, "Understanding and controlling software costs", *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1462–1477, Oct. 1988, ISSN: 2326-3881. DOI: 10.1109/32.6191.