Applying and Evaluating Functional Programming Paradigms and Techniques in Developing Games

City, University of London MSc in Computer Games Technology

Stuart Mathews

Project Report

25/09/2021

Supervised by Dr Chris Child

School of Mathematics, Computer Science & Engineering

Declaration

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed: Stuart Mathews

Abstract

It is suggested that Functional programming produces more robust, predictable and less cognitively complex code. We explore this by refactoring an existing imperative game (based on C# and MonoGame) by introducing new functional programming paradigms and techniques and explore its effect primarily on code complexity. The initial game code is split into two codebases, one representing the original imperative game code and the other the modified functional codebase. A static code analysis is run on both codebases to assess the complexity metrics which are then comparatively assessed to determine the impact that these changes have had on Cyclomatic Complexity, Maintainability Index, Class Coupling and Lines of Codes.

Initial results indicate that overall the functional codebase is larger, more coupled and more complex however through further analysis, improvements are seen in Cyclomatic complexity in existing areas of the game as opposed to newly introduced code which largely are more complex. These findings also suggest that traditional reported cummulative metrics such as Cyclomatic Complexity may be less objective in effectively representing code complexity in function-heavy codebases.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer game development is complex. It is time-consuming, requires a good understanding of the interplay between computer game technology, simulation, and logic, and often requires low-level performance optimisations.

The above, coupled with adherence to tight development schedules, means productivity is crucial during game development and makes dealing with complexity an increasing priority, particularly as "... games have ballooned in complexity". (Blow, 2004).

Game engines are often described as "... extremely complex piece[s] of software" (Maggiore, Bugliesi and Orsini, 2011) and it has been suggested that furthermore they are cognitively complex environments because of the diverse nature of the interactions and dependencies between underlying subsystems. (Sweeney, 2006)

1.1 Objectives

This research looks to explore the implications of taking an existing game written with the *MonoGame* game engine, using C# and re-factoring it to incorporate functional programming (FP) and associated hybrid strategies using the Language-Ext functional library (Louth, 2021).

It is suggested that these may reduce the cognitive complexity in reasoning about the game's internal logic and increase the reliability and predictability of its effects on co-participating systems. This is done with the view of refactoring as the "... process of changing the structure of a program without changing the way that it behaves". (Murphy-Hill, Parnin and Black, 2012)

The second objective is to illustrate and document the usage of the functional programming constructs provided by the Language-Ext library itself, both within the game but also in a generic fashion. To this end, along with the game source code, a comprehensive domain-agnostic tutorial is provided to support how these concepts and techniques can be used generally. This tutorial is provided as external reference and serves as an educational contribution and reference.

The final objective is to determine how these functional programming constructs have impacted the structural code complexity, as reported by traditional code complexity metrics, primarily using McCabe's Cyclomatic Complexity (McCabe, 1976). This is done by comparing the code before and after the refactoring process.

The following will outline the main components of the report.

We begin by framing functional programming in the context of game development and introduce Language-Ext.

We then explore the essence of how functional programming differs from traditional imperative programming and its main distinguishing qualities. We define the main paradigms and techniques used in functional programming and put these methods into context.

In doing so, we explore declarative programming style as exemplified by most functional programming languages to describe logic more simply and flexibly by presenting the concept of representing logic through functional composition. We then explore strategies behind implementing reliability by implementing side-effect-free 'pure' functions and briefly look into immutability.

An approach is described for ensuring functions always finish reliably, irrespective of the outcome and thus negating the need for exception handlers to deal with failures. We investigate lazy evaluation and the idea of automatic validation and short-circuiting through the usage of Monads and monadic functions by way of pipelining of functions that use Higher order functions (HOFs) to express logic as a series of expressions (FP) instead of as a series of statements as is characterised by traditional imperative programming (IP).

As previously noted, we explore these concepts and ideas and see how they are implemented in an existing prototypical game called MonoMazer (Mathews, 2021), a Pac-Man® style platformer. A strategic approach to adapting an existing codebase to its functional equivalent is also discussed.

We then analyse the changes to the before and after snapshot of the game's code by performing a comparative analysis of the impact these concepts have had on the complexity of the resulting codebase by assessing standard complexity metrics.

Finally, we discuss the results and their implications.

Chapter 2

Context

2.1 Functional Programming and Game Development

John Carmack, arguably one of the most famous game programmers suggests that "... a large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in", and goes on to say that "... programming in a functional style makes the state presented to your code explicit, which makes it much easier to reason about, and, in a completely pure system, makes thread race conditions impossible." He says further that there "... is real value in pursuing functional programming", and that "No matter what language you work in, programming in a functional style provides benefits.", and suggests that C++ does not prevent one in doing so. (Carmack, 2012)

There are games written entirely in purely Functional Languages however they are limited. Notable examples include '*Nikki and the Robots*' (a now open source and previously commercial game written in Haskell), '*Haskanoid*', '*Allure of the stars*' and those aimed at mobile platforms such as '*Magic Cookie'* and'*Enpuzzled'* (Zeller and Perez, 2019).

Tim Sweeney of Epic Games says, in a presentation named *'The Next Mainstream Programming Language – A game developer's perspective'*, that performance, reliability, concurrency, productivity, and modularity are key concerns for game developers. He suggests further that reliability and concurrency specifically are two major inherent failings in programming languages used in game development.

In a paper titled, "Monadic Scripting in F# for Computer Games", the authors introduce an approach which is used to reduce the complexity in game scripting which are exacerbated by scalability issues particularly as "... behaviours grow more complex" which makes existing approaches "... difficult to maintain". This suggests that FP in gaming is also being considered as an antidote to growing scalability concerns amidst complexity in game scripting scenarios (Maggiore, Bugliesi and Orsini, 2011).

Problems of reliability (safety) and uncertainty embedded in imperative style programming, which otherwise would be curtailed by functional designs, expose problems inherent to real-time systems such as concurrent and multitasking execution environments (including games) where concurrency and parallelism are important performance mechanisms (Murphy *et al.*, 2019) particularly as predictability is essential to reasoning about threading effects (Butcher, 2014).

In terms of complexity, Sweeney states that gameplay simulation is inherently imperative, and refers to the evolution of state-change in such objects as a 'Gratuitous use of mutable state', and that of these game objects, '10,000's [of] objects must be updated'.

He (Sweeney) also cites the high level of coupling between game objects, which contributes to the complexity of game development in this area and he suggests that with the increased dependencies, comes a higher rate of propagation of side effects throughout the codebase.

This suggests that a pursuit to reduce the impacts of unpredictability in game code and increasing its reliability would be welcome in the game industry.

Modularity is also cited as an important concern where often around 10-20 libraries are used per game. This is also an issue for the development of games developed using functional programming as cited by (Zeller and Perez, 2019) in reference to the poor representation of modular frameworks in FP.

Sweeney predicts that future game developers will want to write all numerical computations as 'referentially transparent' functions suggesting that this would be a benefit for concurrency and promote flexibility (future change) i.e. the ability to freely replace side-effect free code to reduce error-propagation. We will briefly explore this idea through the application of pure functions.

2.2 Language-Ext

Languages-Ext was developed by Paul Louthy and associated contributors and was primarily created to make programming existing C# code safer and more reliable such as preventing runtime errors and preventing null reference bugs for example, and to promote using functional techniques to improve the cognitively qualities over traditional object orientated techniques. (Louthy, no date).

When it comes to writing games in a pure FP language, and specifically using MonoGame, it is possible to do so entirely in F#. It is also possible to write entire mobile games using Haskell (Zeller and Perez, 2019).

However, like most purely FP languages, including Haskell and F#, the transition of many game developers from an imperative approach to a functional mindset is hampered (as indicated by the previous study by Zeller) by the specific lack of tooling and generally a significant investment in learning FP.

Such a transition is comparatively radical. This 'paradigm shift' instead of 'paradigm mix' suggests replacing existing code with new syntax (and effectively programming language) that deviates considerably from industry conventions i.e. imperative techniques and their established practise. Carmack for one, suggests against it. Finding experienced game developers who are adept in developing games entirely in FP is likely to be difficult, as is the re-training investment for new hires that would need to re-skill away from industry standards.

C# on the other hand, has improved tooling, a similar programming methodology to C++ (which is generally considered the industry standard and based on imperative underpinnings) and does have support for many functional programming techniques, albeit without a coherent set of functional programming frameworks and libraries to support these ideas and techniques. Languages-Ext is perhaps the most interesting in this role, and the focus of this research.

Furthermore, C# and Language-Ext provides a hybrid means for new or existing MonoGame or Unity developers to use an intermediate, interoperable language for realizing and implementing functional paradigms and techniques without replacing existing IP investment outright, which may not be practical for games with existing investment in skills. While a learning curve would still exist, it would likely be less steep and more manageable compared to the more radical investment required to transition to a purely functional environment.

C# thus represents a fertile foundation onto which to implement and test functional techniques without jeopardising existing developer experience, investment, and productivity.

To the author's knowledge there is no research using Language-Ext in computer games, nor its effect on code complexity or reliability and this suggests that further research could be useful.

One of the difficulties in approaching Language-Ext, is that it can be difficult to find examples of implementation where functional programming knowledge is not a prerequisite. This limitation is sought to be addressed through this research and the accompanying *"Understanding Language-Ext"* tutorial.

2.3 Imperative programming vs Functional programming

Programming can be considered as a means in which programmers instruct a machine to perform a series of fundamental tasks. In most cases, these tasks pertain to controlling the underlying mechanisms of the machine.

For example, a traditional view of a computer's functionality, is defined by its ability to perform computations. These can be viewed as an application of a series of interactions with its sub-components such as its memory, CPU, and disks etc in order to bring about some desired result or computation.

A computer-based computation is arguably an implementation of an algorithm reduced into its machine-dependant computations which then bring about the effects of the algorithm.

Because traditional programming began with the control of these machine sub-systems, most programming languages themselves have been expressed in such a way as to model logic and thinking about computation as the manipulation

of these sub-systems. This essentially is how we learn about programming but "... one must mentally execute it to understand it" (Backus, 1978)

In Assembly, for example, one expresses the idea that two positive numbers when added together result in a third larger quantity as sequences of 'mechanical' instructions where data is moved and manipulated in memory with the MOV instruction as an operand prior to being manipulated by the CPU to perform an ADD operation.

This imperative styled approach is due in large part to the work of Alan Turing and John Neumann around the conceptualization that computation can be performed by coordination of a physical machine's subsystems and can be represented through its consistent operation thereof. See (Backus, 1978; Hudak, 1989)

As these imperative programming (IP) languages have progressed to model more and more complex ideas, the abstraction of manipulating these machine instructions has increased which hide these specifics from the programmer, however, the underlying idea is still that they describe how or express programming solutions as conceptual ideas mapped to a series of machine operations that can represent those ideas.

In order to model thought on a Turing machine, it needs to be reduced to mechanical imperative operators such as STORE, LOAD, ADD, MOVE etc which has been argued is an "... intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the tasks at hand" (Backus, 1978). This suggests that creative expression of ideas and thought may be restricted by the limits of computational thinking enforced by imperative programming.

This is at the heart of imperative programming (IP) languages, which makes up most programming languages used today.

Furthermore, as the Turing machine and the Von Neuman architecture (Backus, 1978; Hudak, 1989) represented the early foundations of modern computing and because of its inherent ability to model ideas relatively simply, imperatively styled programming languages have dominated the gaming and computing industry, with the top five most popular languages all being IP languages, including C, Java, Python, C++, and C# (in decreasing order of popularity) according to the TIOBE Index ('TIOBE INDEX TIOBE - THE SOFTWARE QUALITY COMPANY', 2020)

Functional programming (FP) languages in contrast, have their foundations in an alternative, declarative (describing what instead of how) approach to modelling thought, based on the manipulation of ideas instead of on their mechanical representation, in the same way mathematics does. (Hudak, 1989)

Mathematics like FP for example, is an application of conceptual ideas, not concerned with explicitly storing intermediate values into variables and then manipulating them. This is true of summation ($\sum$) – it's a conceptual computational mechanism, represented by a sparser, more conceptual syntax.

In this way FP is designed to model ideas to reflect the way people think instead of relying on how the machine needs to represent computation as a series of data mutations. Here the "... term *mutation* indicates that a value is changed in-place..", i.e. "... updating a value stored somewhere in memory." (Buonanno, E, 2017)

This suggests arguably that FP is more concerned with how people think and reason about problems than traditional imperative programs are, and so are not bound to imperative notions of consecutive data mutations being central to representing it. For example, with functional programming, the programmer is not concerned (or burdened) with representing a summation by first assigning data to memory (variables), then by subsequently iteratively changing its state over time in a imperative looping fashion, before perhaps moving it out and then evaluating the result, or some other such requirement. This is how a computer works, not necessarily how a conceptual idea must be modelled syntactically in code.

Functional programming constructs provide an intermediate layer of expression in representing thought, which of course, will ultimately be represented into low-level computations, however without those computational mechanisms (and associated implications) getting in the way of describing what the code must do and diluting in the problem domain.

Imperative programming is however more prevalent when considering its provenance and the popularity of C, C++, and C# and other derivatives such as Java and Python.

2.4 Truth, predictability, and the implications of imperative programming

A fundamental concept shared by both FP and mathematics is data that represents an idea remains consistent and true or at least the idea holds true.

An example of an unchanging, predictable idea is that of a mathematical equation, or process, where when passed the same data, it will yield the same result over time without exception. The data is immutable. No changes to the state of the data is allowed. This is a fundamentally immutable idea that is corruptible when represented in an imperative sequence of steps and thus may violate the integrity of the idea itself over time. This typically manifest as a bug due to side effects of imperative computations.

This idea, of data not changing once its set, is a first class citizen of FP languages, but more work and diligence is required to achieve it in IP, where its mutable data's 'side-effects' may cause an idea or statement to become incorrect later in history when its state is usually expected to be independent of time. An example being a function result forming a larger compositional idea changing later in time.

A by-product of using imperative languages is that the data that represents ideas are as explained usually stored in variables (that reference memory location), which is itself subject to uncontrolled changes. This is fine when those subsequent operations will define and produce the desired computational result. However, using this imperative model means that this desired result is still subject to unexpected change (it's mutable) - even after it's perceived to be in its final state.

This is further complicated by complex problem domains such as game development and programming tasks such as concurrency (concurrent changes to state) which can increase the cognitive load on the developer to determine when and how these erroneous changes have occurred. This can also present difficulties when stepping through a debugger.

FP languages also exhibit a property borrowed from lambda calculus called "referential transparency" which means that functions that have no dependencies can be substituted with other like functions over time, perhaps improving or replacing the functionality but crucially it means that their execution can be done in any order as they don't depend on shared state which makes them good candidates for parallelism. These are usually referred to as 'pure functions' (Goldberg, 1996)

2.5 Benefits of functional programming

Research has shown that functional code is significantly more concise than procedural code and that "...shorter programs will tend to have fewer bugs.", which alone is a compelling reason to use functional programming, not least within games (Nanz and Furia, 2015). The evaluation of the impact that FP refactoring has on lines of code is analysed later.

FP languages help enforce data immutability and it focuses on describing logic as essential conceptual operations free from the machine-based notions and limitations which increases its expressiveness.

(MINSKY and WEEKS, 2008) highlights that the improvements that functional programming paradigms bring helps to promote a more robust, reliable and correct codebase which is more predictable.

The ability to predict outcomes based on declarative expression devoid of imperative side effects is what makes functional programs "...easier to reason about". (Wyatt, 2003)

This provides a multi-faceted view of complexity: Cognitive complexity can be
interpreted as the impact or ease at which one can understand or predict the effect code has, while code complexity, as measured quantitatively using metrics such as Cyclomatic complexity are derived from how the code is structured and how this can affect the comparative complexity of the function itself considering complexity as a function of the number of linear choices or pathways through the code. Both are useful perspectives of complexity within programming.

Functional programming, apart from its apparent expressiveness of logic also allows for predictable outcomes through its approach to reducing side-effects. Knowing that a result will not change in the future is an important idea that can aid determinism and fault finding.

Coupled with the advantages of immutable (unchanging) data i.e. lack of side effects, is that certain characteristics of techniques native to functional programming such as pure functions, allow operations to be easily parallelised: it

is often a particular pain-point in imperative programming, that the imperative structure of holding onto a lock and sharing mutable state(that routinely changes), can cause reasoning about multi-threaded parallel work to be error prone, difficult and non-deterministic.

Higher order functions (HOFs) and Lazy evaluation (which is the ability to delay running a given function parameter until its required) are heavily used in FP, and have even influenced IP languages such as C++ to allow for delayed execution (which has performance implications) and higher levels of abstraction and generalization of behaviour in code. This not only can improve the generality of code and reduce redundancy but also offers higher levels of abstraction to reduce complexity in code.

2.6 Limitations of Functional Programming

Research has shown that functional programming ideas are initially difficult to grasp, particularly when presented to students approaching declarative style (Lopez, 2001)

Functional programming does often present an otherwise unusual way of structuring thought which makes it foreign to most programmers, particularly within the game development discipline who rely heavily on the close relationships between hardware and computation through dominant IP languages such as C++/C/C#.

FP is often thought to be too high level in many respects due to its declarative approach, however as will be presented later on in this research, the need for abstraction in programming, particularly in game development, to reduce the complexity of application domains, can benefit from a higher, more declarative means of specifying logic and other FP paradigms and techniques.

Developing games in a pure functional language is rare, particularly because games rely on side-effects, interaction and other state changing scenarios including dependence on I/O.

In addition, there is a "... lack of libraries and frameworks" and "... poor integration with existing toolchains" where the main challenges are purity, compositionality, and abstraction. (Zeller and Perez, 2019)

Functional programming in its pure sense i.e. being strictly devoid of I/O (as this introduces side effects and unpredictable outcomes) and other state change, makes it difficult to develop entire games with, however examples do exist albeit limited in comparison to what is evidently possible today using imperative languages and frameworks. Most AAA games use C++ for example.

Performance is a concern for many game developers because in most cases functional programming relies on copies of data (albeit short lived copies) to enforce immutability and due to this, garbage collection is usually a requirement for functional programming environments which can be comparably slow to C++ for example which does without one for performance reasons.

As presented earlier, these limitations i.e. I/O and other non-deterministic characteristics of mutable data, are inherently unpredictable and avoided in FP. These are however essential to most games.

A hybrid, non-strict approach is necessary.

2.7 Spoke-hub distribution model: A mixed hybrid implementation methodology

C++ and C# are IP languages routinely used in the game industry with the former being more popular than the later, however C# is increasing in popularity due to game engine frameworks like Unity, MonoGame and Stride.

With the increased abstraction required of imperative programming languages due to ballooning complexity requirements, implementing functional programming ideas to reduce complexity is becoming more compelling.

Both C++ and C# have introduced support for HOFs (lambda functions) and C# has introduced declarative syntax with LINQ, showing that "...functional concepts are being introduced into classic object-oriented technologies such as Java and .NET." and while this trend may not result in a complete paradigm-shift, "paradigm-mix" is already well established". (Turek *et al.*, 2018)

Its important to note that FP and object orientated programming (OOP) are not mutually exclusive - OOP serves as a structuring element for software designs in which a FP approach can be applied. They are interoperable.

However, implementing FP style in the face of an imperative-first style language like C++ or C# requires leveraging other ways to enforce these principles, such as programmer diligence (which is burdensome and unlikely to be robust) and using functional techniques and libraries such as Language-Ext, Immutable collections etc.

This represents an evolving 'mixed hybrid' approach, based on imperative foundations laid by the gaming industry and computing in general, where FP ideas such as describing logic declaratively and reducing side effects, can be strategically incorporated into existing code (as needed) to realise their benefits.

There is also a research effort in functional programming to "...increase the expressiveness of functional languages for applications in which state (through assignment) is seen as necessary in conventional programs". (Goldberg, 1996)

This need for middle ground indicates that it is necessary for some programs like games to continue to function in an imperative way and it is indeed more productive to do so, but with increases in adherence to functional ideas.

This is arguably true of many applications characterised by IP such as line of business applications and games for instance where side effects in some form is essential, however in a more controlled capacity.

A hybrid model that embraces both imperative and functional needs can be representing metaphorically using a bicycle wheel, which describes a dual natured approach, and compatibility with both FP and IP in the current world of the Von Neuman/Turing computing.

In FP this general idea is usually referred to as having a functional core and an imperative shell where the core aspects of logic be modelled centrally (hub) using FP ideas such as purity and immutability of data, while unpredictable areas such as I/O and imperative side effects and routines are relegated to the fringes or shell (the spokes):

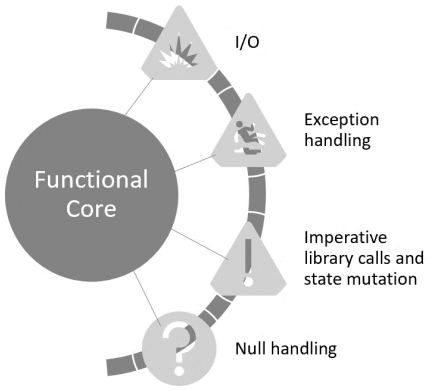

Figure 2.1: Hub-spoke hybrid model

In this way, the argument is not to use functional programming paradigms in a strict sense as to remove all possible side-effects or I/O, as this would be counter-intuitive and is likely scare away traditional programmers, but to combine the two approaches, taking advantage of both.

2.8 Complexity

2.8.1 Cyclomatic Complexity

Research has shown that mitigating complexity is crucial in writing computer programs, as "...complex code is hard to write correctly and hard to maintain, leading to more faults" (Jbara et al, 2013). This has been confirmed by further research which shows that in regards to defects, "...that complex code contained more than 40 percent of the problems" in contrast to "...simpler classes only about 12 percent of all defects". (Schroeder, 1999)

The McCabe Complexity metric (MCC) is a software complexity metric that determines the complexity of a function by the number of linear pathways through the code. It suggests that those that have an MCC value greater than certain threshold (typically between 10 and 15) are complex and should be split into smaller functions, effectively reducing its complexity.

For example, the methodology used by *SonarCube* is summarised as being calculated "... based on the number of paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each function has a minimum complexity of 1. This calculation varies slightly by language because keywords and functionalities do"

2.8.2 Maintainability Index

The Maintainability Index, as described by Microsoft Code Analyzer (MSCA) is "... an index value between 0 and 100 that represents the relative ease of maintaining the code." The following is indication of how it is calculated:

$$Max(0,\ \frac{(171 - 5.2 * \ln(\text{Haldsteam Volume}) - 0.23 * (\text{Cyclomatic Complexity}) - 16.2 * \ln(\text{Lines of Code})) * 100}{171}$$

This additional metric will be used to assess the effect on maintainability before and after the introduction of functional paradigms and techniques.

2.8.3 Class Coupling

Class coupling is an indication of how many classes one class uses.

According to Microsoft, Class Coupling "...measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration. Good software design dictates that types and methods should have high cohesion and low coupling. High coupling indicates a design that is difficult to reuse and maintain because of its many interdependencies on other types." (Mikejo5000, no date)

2.9 Functional Paradigms and Techniques

In order to drive the functional implementation that would result in functional paradigms and techniques being factored into MonoMazer, it is necessary to identify them and put them into context as they will form the basis of the main programming effort to be carried out.

For the purpose of this research, a paradigm is seen to represent an idea or concept whereas a technique is a practical application or methodology.

Reliability considers behaviour in the face on exceptional circumstances to be consistent. Predictability is concerned with how easily the effects of future outcomes are reliably reasoned about i.e. predictable.

For example, a square root function that does not change the value of its argument, will allow for predictable valuation of its argument, before, during and after the invocation of the function within the codebase. This is a predictable function which yields no side effects that could change its value or its input or output, and thus both are easily predictable. Furthermore, a function that always finishes and always returns a result, guarantees the prediction that the code thereafter will execute every time.

Table 2.1: Functional paradigms and techniques

Functional programming Concept	Category	Purpose
Declarative Style	Paradigm	Enhance readability & expressiveness
Higher-Order functions (HOFs)	Paradigm	Enhance genericity and re-use
Functional composition	Paradigm	Enhance flexibility, expressiveness
Lazy evaluation	Paradigm	Enhance performance
Immutability (and side effects)	Paradigm	Improve reliability/predictability
Monads	Paradigm	Improve reliability/predictability
Pure functions	Paradigm	Improve reliability/predictability
Optional Types	Technique	Improve reliability/predictability
Functions always finish	Technique	Improve reliability/predictability
Suppression of Exceptions	Technique	Improve reliability/predictability
Short-circuiting	Technique	Enhance performance
Smart constructors	Technique	Enhance reliability/predictability

In some instances, there may be overlap where an idea is the technique itself. For example, to ensure that functions always finish, requires implementation of a technique such as suppressing exceptions as well as retuning optional types.

2.9.1 Declarative Style

Declarative style is an approach to describing logic or computation that expresses the intent that the parts are intended to perform, i.e. one "... declares what is being computed, rather than instructing the computer on what operations to carry out in the computation", as would be done in an imperatively styled piece of logic. (Buonanno, 2017)

Procedural (imperative) style can be succinctly defined as describing the "how" while declarative programming describes the "what". (Lopez, 2001)

This layout of intent can be achieved by specifying a series of descriptive labels for logical operations such as function names which hide the implementation of those functions and provides an expression that is "... higher level, and closer to the way we communicate with other human beings." (Buonanno, 2017)

A study describing a declarative system to specify the requirements for input validation which were "... complex and tedious", suggests that using a declarative approach, can lead "... to a very concise maintainable implementation" and can "... release the application programmer from the tedious details" (Hanus, 2007).

This reductionist approach through abstractions therefore can thus help to simplify and reduce the cognitive complexity of interpreting and keeping track of large amounts of information in game code.

A study entitled "Declarative Game Programming" suggests that while "... performance requirements subject to resource constraints have often necessitated a low-level approach to implementation", referring to the prevalence of IP in computer games development, and that it is therefore less desirable to use a declarative approach in those cases, that "... implementing higher level parts" of a game using declarative styles is possible and useful. (Nilsson and Perez, 2014)

While this research is interested in the cognitive qualities of declarative style in code, performance is an important consideration in game development and further research has shown that "... declarative processing" of computation "... can improve the computational performance of scripted character AI by an order of magnitude" but more importantly, suggests that a hybrid approach to using declarative style is necessary and that it "... would be both difficult and unnatural to write all of a game in a declarative language". This fits well with the mixed-hybrid approach proposed in this study. (Sowell *et al.*, 2009)

Sweeney of Epic games says that as game developers, they "... will gladly sacrifice 10% [of] our performance for 10% higher productivity", particularly where this is gained through a reduction in code faults or in other words, code safety is important. (Sweeney, 2006)

Popular game engines like Unity, Stride and MonoGame are well placed to benefit from the cognitive benefits of declarative programming due to the first-class support for specifying declarative expressions through C# language features such as LINQ. This offers game developers a wider reach, where using declarative style in game engine subsystems such as gameplay simulation can directly benefit from reducing the cognitive complexity associated with these areas.

Another important aspect of declarative style, is that intent in logic can be expressed "... without specifying the exact side-effects in-between", and thereby limiting the dependence on imperative constructs to represent these logic declarations. (Sowell *et al.*, 2009)

2.9.2 Higher-Order functions

Most "... functional languages support functions that operate over functions... ", meaning that Higher Order Functions (HOFs) can be thought of as data and used as input arguments to other functions, including being returned from them (Goldberg, 1996).

HOFs are also used to simplify code and it has been suggested that they are "... essential for generic programming" (Lincke and Schupp, 2009).

Further research which expands on this and provides qualification suggests that they can increase not only generality and but also reduce redundancy in code. This is typically achieved by allowing functionality or behaviour to be

passed into an otherwise fairly independent and now more generic function in order to benefit from the specified behaviour defined within the incoming function argument (Xu, Jia and Xuan, 2019). This practise can be referred to as "factoring out" a function. (Kühne, 1994)

HOFs have also been attributed to facilitating "... the separation of concerns", referring to the factoring out of behaviour from otherwise monolithic logic functions and they also contribute "... to code conciseness". (Buonanno, 2017)

From a maintenance perspective, by allowing generic functions to differ only by the variety and behaviour of their function parameters, limits the need to constantly change the enclosing functions code and prevents the propagation of change applicable to routine manipulation. This is particularly useful if the enclosing function is a core, critical or already a well-tested function.

Increasingly more imperative programming languages are supporting this FP paradigm including C# and more recently C++ through lambda functions.

2.9.3 Functional Composition

Functional composition is the reduction of larger logic into combinations of discrete logic components or functions, each which performs a unique computation that contributes to the overall computation. In FP these functions are pure and thus are reusable, predictable and replaceable.

However, a key aspect of this composition is the role that HOFs play, often themselves being embedded into input and output of other components that make up the larger logic circuitry. HOFs form a key part for example, in performing data transformations such as those seen in Language-Ext's implementation of Monad types (introduced later).

The organisation of functions into a composed whole also contributes to declaratively outlining the intent of the logic by chaining the names of the functions to declaratively express the "what" of the logic, as opposed to detailing "how" the logic will perform the computations.

This also makes it easier to make changes to logic by replacing a component or adding a new step into the logic workflow, which "... leads to greater flexibility". (Buonanno, 2017)

2.9.4 Lazy Evaluation

Lazy evaluation specifies that the evaluation for a result, usually the execution of a function parameter may occur in the future, or not at all such that "... no computation need occur until it becomes-directly or indirectly – essential to the next visible (output) behaviour of the program" and "... postpones that initial evaluation until first use" (Wise, 2003). This means that encountering the evaluation source itself (the function) does not necessarily mean that its evaluation will be performed until later.

This is a language feature inherent to non-strict functional programming languages like Haskell, which allows the execution of the function being passed in to be under the control of the higher order function who received the function. This idea means that function evaluation might not need to be executed.

This has a performance benefit because HOFs need not run/evaluate if conditions suggest that this is not required.

2.9.5 Immutability & Side effects

Immutability refers to the prohibition of data to change once it is set. This is at odds with traditional imperative programming which models computation around state change.

For example, this concept can be articulated thusly: "In an imperative language, the first call to f might change the value of a variable used by the second call to f and thus represents a side-effect which impacts the second call to the function f.". (Goldberg, 1996)

2.9.6 Pure functions

Pure functions are functions that are isolated, independent functions that do not propagate downstream changes in data they use. This is largely achieved by relying on FP languages to enforce immutability of data and can be implemented in IP languages by copying data to avoid inducing side-effects of manipulating the data, which restricts accessing shared data.

Pure functions don't perform any operation that can yield a different result over time, such as manipulation of I/O or access to shared state.

There are inherent benefits to pure functions due to their adherence to the principle of immutability. For example, they are automatically parallelizable due to the fact they have no side effects. The latter also contributing to the predictability of such functions.

2.9.7 Monads

Mathematically a monad "... is defined as a triple composed by a functor and two natural data transformations acting on it" (Bonelli *et al.*, 2014) however for the purposes of implementing this concept programmatically, it can be seen as a storage datatype that defines 2 transformations functions. These two functions are implemented as Map() and Bind() and are explored in detail in "Understanding Language-Ext" which accompanies this study.

The transformation functions follow a convention on how the transformation function should work, what the general format of the return type must be and declaration of the argument - a user defined function that defines the transformation itself.

This transformation is enforced by convention by ensuring that a map function and additionally a bind function is available for Monad types. Monads thus primarily hold and transform data and the manipulation of these transformation can be used to create a single expression that represents a pipeline of a series of processing steps by chaining Monads together. These pipelines reduce the need to perform conditional expressions in the code as conditions are built into how the transformation functions work by default. This is expected to contribute to the reduction of linear pathways through functions and therefor reduce the cyclomatic complexity.

FP is said to be distinct from IP where "... imperative code relies on statements; functional code relies on expressions". Monads help to chain the evaluation of expressions. (Buonanno, 2017)

Monads are natively supported in most FP languages however more effort is required to implement them in IP languages such C# and C++ (Teatro, Mikael Eklund and Milman, 2018)

2.9.8 Using optional types

Optional data types are Monads which allow one type to represent two possible states however not as the same time and supports transformations into and out of these states. This allows a function to return a single type which then can be inspected by the caller to determine which state the result is in as an indication to the outcome of the function.

In Language-Ext, Option is useful in removing NULL from the codebase, which is the source of many unpredictable failures (see Sweeney), by encapsulating the concept of NULL as an optional type having a value of *None* (representing NULL or nothing) or *Some* value – meaning these this type can be used uniformly as a return type irrespective of its state.

These types are usually represented in FP as Monads and in Language-Ext as represented by the Option<T> and Either<L, R> monad:

```
// a NULL component is turned into a Option<Component>.None:
public Option<Component> FindComponentByType(Component.ComponentType type)
=> EnsureWithReturn(() => Components.SingleOrDefault(o => o.Type == type))
    .Map(component => component ?? Option<Component>.None)
    .IfLeft(Option<Component>.None);
```

A key aspect of optional types is that in order to access the underlying value contained withing them, both eventualities of the state must be catered for in the calling code, reading a value directly is not possible otherwise. In Language-Ext, the Match() function is used to do this.

When a Monad is asked to transform its current state/value, such as in pipelines, they by default only operate only on valid states/values such as Some state (Options) or Right state (Either), while invalid values (unhappy path values) are also evaluated and factored into the behaviour of the pipeline, by causing short-circuiting to occur.

2.9.9 Predictable Functions

Reliability is a key concern for game developers (Sweeney, 2006)

Functions that are unable to return a value or finish due to an exception having been thrown do not and cannot reliably guarantee to fulfil the function's specification. However, a function that always finishes provides a predictable flow of program logic.

To guarantee that a function will finish, exceptions need to be suppressed and this means that whatever the outcome of the function is, it needs to be able to be communicated back to the calling function: Either it needs to signify why it failed (but still return) or the result of its success should be returned instead.

This dual natured return value can be represented by an optional construct such as Either<L, R> or Option that can hold either an error/invalid state or a success state. In this way a function that returns an optional type is consistent - it always returns something, and this introduces predictability into the codebase.

2.9.10 Suppressing Exceptions

Exceptions are an issue in game development and share the category of concerns classed as dynamic or runtime failures which can affect the reliability and/or predictability of outcomes and can lead to wasted effort in finding trivial bugs. (Sweeney, 2006)

Some examples include accessing arrays out-of-bounds, dereferencing NULL pointers, integer overflow, access to uninitialized variables and so on. Sweeney says hunting down these bugs severely limits game development productivity.

The nature of the way in which these problems are detected (usually via throw construct) and then handled, is what is called exception handling and traditionally requires the location in which the issue occurs in to be halted in-place and control to be given to an upstream function that will deal with the problem.

The main issue with this approach is that it introduces side effects, and removes predictability of what functions will or won't end up doing i.e. "...they disrupt normal program flow" (Buonanno, 2017) - they might work but they might experience a problem and fail, with the resolution usually being handled by a different area of code. Thus, looking at a function, it become difficult to reason in a predictable way what happens next and for accurate determination if proceeding code will be reached or not.

2.9.11 Automatic validation and Short circuiting (HOFs, Lazy evaluation)

As briefly highlighted earlier, in order to access the contents of a Monad, both scenarios/states (valid or invalid, left or right, some or None) that its data may be in must be catered for in the calling code when querying its value.

Automatic validation is the built-in conditional control in Monadic transformation functions such as map() and bind() that prevent the execution of the transforming function (when data is in an invalid state) and can signal to the caller through its return state that it was not run because the condition was not met.

This allows chained/pipelined monadic operations such as Map and Bind to simulate a short-circuiting scenario where all subsequent calls to Map/Bind (upon receiving the invalid state) will also cancel their execution of the transformation function which was provided by the programmer.

The following depicts how this automatic validation can be used, along with lazy evaluation and HOFs to implement short circuiting behaviour which prevent subsequent functions from running if one of the steps fails (causes a result to be in an invalid state).

Note that a Monad can be interpreted as a container, to which transformations of its contents can be performed:

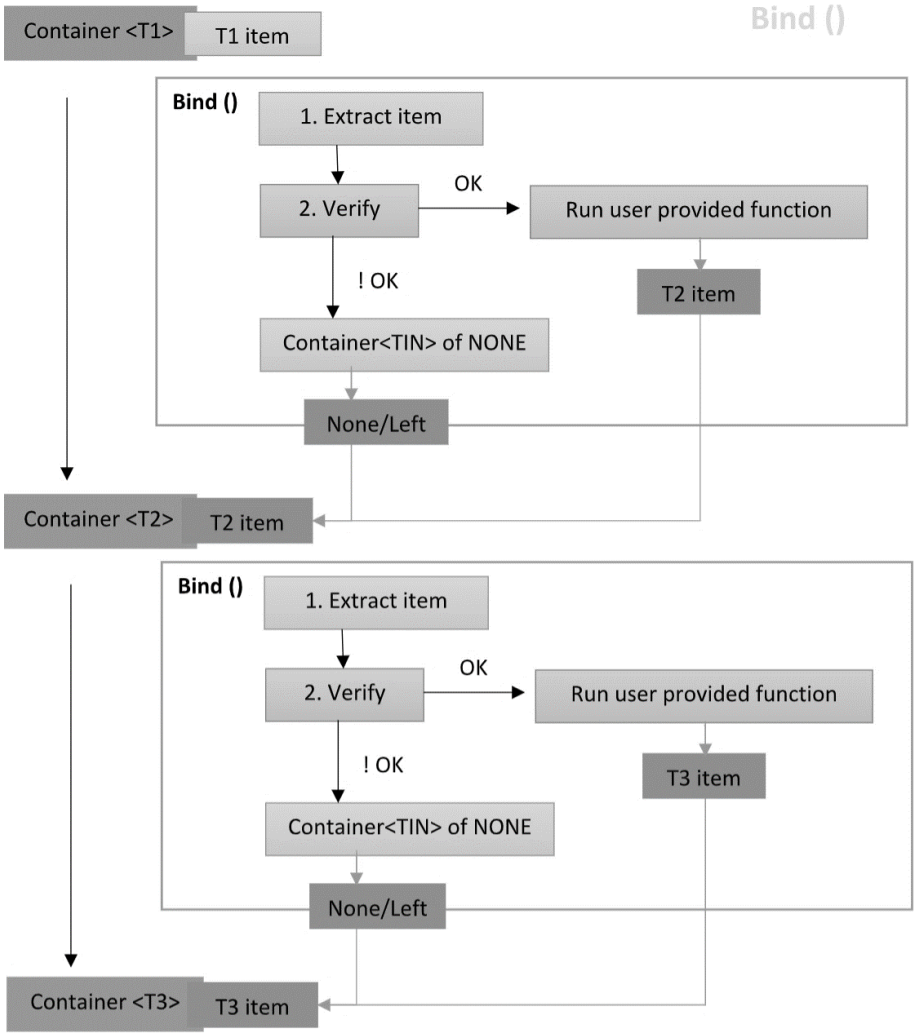

Figure 2.2: Automatic validation: Each bind point is a point that short circuiting can occur

The above shows that all subsequent transformations, T2 & T3 (HOFs), on the Monad container (typically a *Either<L,R>*) may not be performed (during subsequent binds) if the validation of the item within the Monad is determined not to be valid.

This behaviour can be seen when using pipelining transformations:

```
// File: Mazer.cs
// Declarative pipeline chaining monad transformations
 protected override void LoadContent()
=> GameContentManager
    .Bind(contentManager => contentManager
                            .TryLoad<SpriteFont>("Sprites/gameFont"))
    .Bind(font => SetGameFont(font))
    .Bind(contentManager => GameContentManager)
    .Bind(contentManager => contentManager.TryLoad<Song>("Music/bgm_menu"))
    .Bind(music => SetMenuMusic(music))
    .Bind(unit => LoadGameWorldContent(_gameWorld, _currentLevel,
        _playerHealth, _playerPoints))
    .ThrowIfFailed();
```

In the code above, a function that uses short circuiting will cancel all remaining steps in an initialization sequence, if one fails (is in an invalid state). This behaviour can be changed by manipulating the pipeline, based on the state of the pipeline at predefined steps if the programmer wishes it using Match()(not shown).

Each line is effectively an execution of a Monads transformation HOFs that describes declaratively what transformation that step does (e.g. SetGameFont, LoadGameWorldContent). The result can be a failure or a successful result, which is then fed into the next step or bind transformation.

When this layout of logic, and the resulting data from each step is coupled with each step producing and enforcing immutable data as input to subsequent steps, this represents a fundamental motivation for incorporating FP pipelines like this to produce side effect free logic processing.

2.9.12 Smart Constructors

Smart constructors is a technique that avoid exceptions in constructors, where such runtime failures are often critical.

It's mechanism is based on the construction of objects by first requiring a validation of the input to be performed before returning a new instance of an object. In this way it is less likely that invalid data will cause an exception to be thrown when constructing the objects with the data.

This relies on the idea that data might not be correct, and as such relies on representing either/or situations, i.e. the mainstay of optional datatypes.

This can be seen in creating *Room* objects in *MonoMazer*, an example of which is shown later in the describing the methods used. Usually the constructor is private and creation is delegated to a intermediate validation function which uses the constructor if the validation of arguments/data is valid and which then returns an optional type depending on and indicating if the construction succeeded or not, usually as an Either<IFailure, TSuccess> type.

This can be used to improve the reliability and predictability of object construction within game code.

Note that constructor is private and that the *Either<IFailure,Room>* forces the caller to have provision made for both results before whatever result can then be used. This, like Option<T> ensures that there is provision for the alternative state made by the caller by default, i.e. when the object cannot be constructed due to invalid data and when it can.

Chapter 3

Methods

3.1 Objective 1: Refactoring MonoMazer using FP paradigms and techniques

The re-engineering of code to incorporate the FP paradigm and techniques followed an iterative development lifecycle, but one which could be identified as clearly defined stages of incremental progress as shown in the following model:

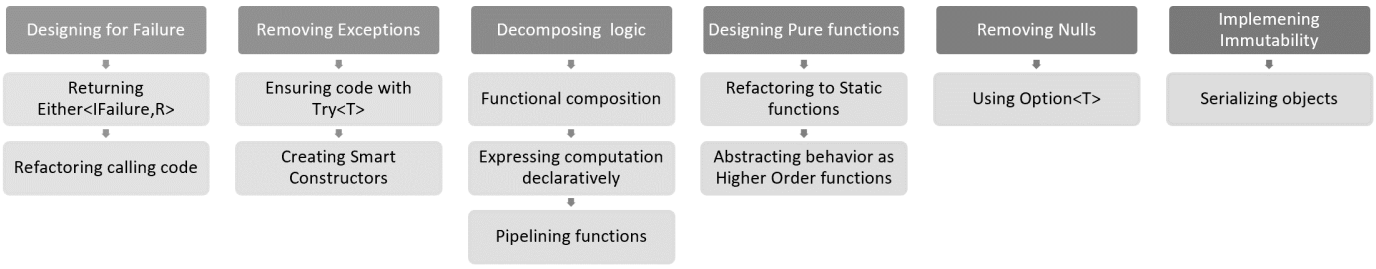

Figure 3.1: Stages of refactoring an imperative codebase into a functional derivative (Left-to-right)

Broadly, the process is carried out from left to right, however these can be revisited and can be done out of order. The left to right process described above was used to approach the refactoring effort in *MonoMazer*:

1. Designing for Failure: incorporates reliable code techniques centred around representing functions as returning Monads that encapsulate the duality of successful or unsuccessful function calls.
2. Removing Exceptions: uses an exception supressing Monad type (Try<>) to reduce exceptions to "left" failure types using Eithers so they can be reliably reasoned about in the return values of functions.
3. Decomposing logic: describes the process of simplifying the expression of logic as well as reducing the complexity by using higher level declarative constructs.
4. Designing Pure Functions: deals with the reduction of change propagation, side effects and re-use of code.
5. Removing NULLs: is related to *Removing Exceptions*, as it too serves to remove runtime failures when encountering NULL in source code.
6. Implementing Immutability: looks at eliminating the impact of change in imperative code.

These were loosely organised as source code tasks in https://stumathews.atlassian.net/jira/software/projects/FP/boards/1 (project planning software) and labelled in the source code commit history included in the appendix:

Table 3.1: Types of source code refactoring task mapped to the stages of refactoring

Task	Description	Stage
FP-9	Return Eithers	Designing For Failure
FP-10	Remove NULLs	Removing NULLs
FP-11	Avoid Exceptions	Removing Exceptions
FP-12	Implement Immutability	Implementing Immutability
FP-15	Refactoring/Misc.	NA
FP-16	Create pure functions	Designing Pure Functions
FP-17	Create utility functions	NA
FP-18	Fix Bugs	NA
FP-19	Make code more declarative	Decomposing Logic
FP-21	Pipelining code	Decomposing Logic
FP-22	Make more readable/understandable	Decomposing Logic
FP-23	Use adapters - ThrowIf(x) to maintain compatibility	Decomposing Logic

3.1.1 Integrating Language-Ext with MonoGame

MonoGame provides 4 main entry points in which game code implementation can be placed:

1. *LoadContent*() to load game level elements and resources
2. *Initialize*() to setup game infrastructure
3. *Update*() to progress game state, and lastly
4. *Draw*() to paint onto the screen

Each can be represented using Language-Ext as a pipeline by implementing the logic within as a series of composed functions.

This is immediately natural when considering how the graphics pipeline is traditionally represented.

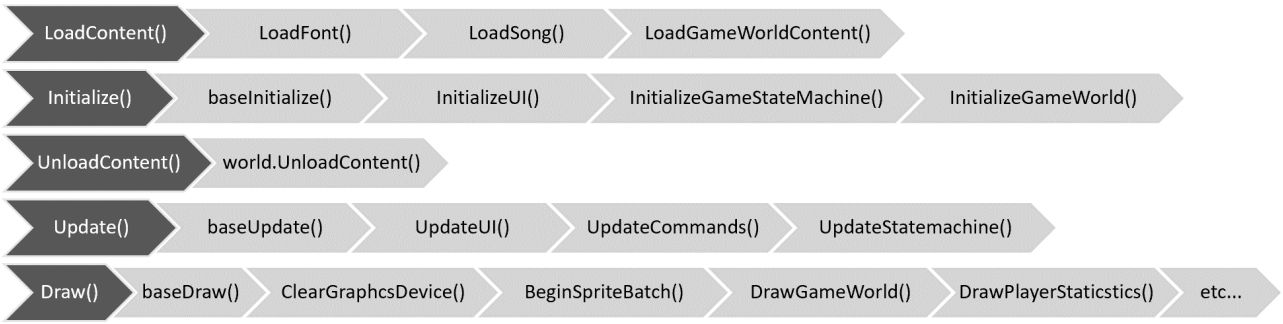

Figure 3.2: Representing the entry points and Pipelines using Language-Ext, as used within MonoMazer

All functions in the pipeline are represented with a specific function prototype that declares that each function will either return a failure type, or the successful result of the function.

This is typically represented using Either<L,R> or Option<T> return types. This configuration also brings together the ideas of functional composition through pipelining, which allows the evaluation of a series of expressions instead of statements, as would be the case in an imperative styled piece of logic. The implementation of these elements is declaratively laid out in C# using LINQ.

The following shows an implementation of the *Update*() pipeline, where a functional composition of the update pipeline is created:

```
// Execute the update pipeline
protected override void Update(GameTime gameTime)
```

```
=> Ensure(() => base.Update(gameTime))// Execute the update pipeline
    .Bind(unit => gameUserInterface
                    .Bind(UserInterface => UpdateUi(gameTime, UserInterface)))
    .Bind(unit => SetGameCommands(_gameCommands, gameTime))
    .Bind(unit => _gameStateMachine
                    .Bind(gameStateMachine =>UpdateStateMachine(gameTime, gameStateMachine)))
    .ThrowIfFailed();
```

This forms the basis of the proposed refactored code architecture, where Language-Ext is used to implement the logic using the functional paradigms and techniques, expressed declaratively and as a series of function expressions. These are also documented in the "*Understanding Language-Ext*" and discussed earlier.

3.1.2 Implementing Reliability

In reasoning about the effects of a function, if a function does not produce the same result consistently, specifically when the same inputs are used, then it is not reliable and the flow of code is not easily predicted, and as such is not easily reasoned about at face value.

To ensure functions are made reliable i.e. they always return a value irrespective of the outcome of the function, including under exceptional circumstances, an important first step is to have all functions return Optional data types where the result was either a failure indicator or the result of the function.

```
public Either<IFailure, Unit> MoveInDirection(CharacterDirection direction, GameTime dt)
{
    switch (direction)
    {
        ...
        default:
            // Return a Failure instead of exception
            return new InvalidDirectionFailure(direction);
    }
    return SetCharacterDirection(direction);
}
```

In the code above, InvalidDirectionFailure will cast into Either<IFailure,T> return type.

This is achieved by ensuring function prototypes were changed to return *Either<L,R>* or *Option<T>* data types provided by Language-Ext. The result of this refactoring task required also ensuring that the calling code of these functions was appropriately refactored to handle the new function return types.

The return type is an Either Monad describing an interface type that describes the nature of the failure if there was one (IFailure), or the successful value of the function, as illustrated in the figure as Unit.

To improve upon the reliability of the codebase, additionally, exception handlers were removed and replaced with a mechanism that turns any exceptions into failure states for above optional types (None or Left), which are required return types of all functions (See code commits labelled as FP-9). Exceptions are a means of execution flow that can cause the function not to maintain its contract with the caller i.e. to not always return consistently and thus it represents a side effect.

This involves having all functions that might throw exceptions to instead return an 'ensuring' HOF which executes the original composed function and returns any exceptions as failures, as shown in the figure below.

```
Either<IFailure, Song> LoadLevelMusic() => EnsuringBind(()
    => MaybeTrue(()=>!string.IsNULLOrEmpty(LevelFile.Music))
      .ToEither()
      .Bind<Song>((unit)=> ContentManager.Load<Song>(LevelFile.Music)));
```

In the code above, the LoadLevelMusic function returning an "EnsuringBind()" HOF to supress exceptions which may occur in impure code such in 3rd party objects as ContentManager.

Incorporating these techniques with functions that are themselves pure, i.e. return immutable data, and that do not mutate any parameters it receives improves overall effect.

In many cases incorporating multiple functional ideas this way, compounds the benefits of applying functional programming paradigms and techniques in code.

Next, in line with the goal of making code more reliable, the possibility of invalid construction of data was considered. This entailed implementing a validation mechanism as part of the construction of objects to ensure that the objects constructed are valid.

This is done by ensuring constructors are publicly inaccessible, and that construction of objects is delegated through "smart constructors" such as Create() which perform validation first:

```
public static Either<IFailure, Room> Create(int x, int y, int width,
       int height, int roomNumber, int row, int col)
   =>  IsValid(x, y, width, height, roomNumber, row, col)
       .Bind(unit => EnsureWithReturn(() => new Room(x, y, width, height, roomNumber, row, col))
       .Bind(room => InitializeBounds(room)));
```

In the code above an object using a smart constructor that validates construction parameters and then returns an indication of the result using the Either<L,R> Monad optional datatype.

Once the first step of the pipeline is performed (*IsValid*) and is validated to yield a non-failure state, then the rest of the pipeline is executed, i.e. the room object is constructed, calling the private constructor, initializing its bounds and sending back the fully constructed object as a Either.

3.1.3 Reducing Cognitive Complexity (Logic)

In terms of reducing the complexity of logic, functional composition was used to reduce constituents of large pieces of logic into smaller constituent functions, aptly named to indicate what the component logic pieces do. Then these are connected together to form the original expression of the original, monolithic description of the logic out of the composition of these constituent functions.

The expression of this composition is done declaratively using LINQ to expose the constituent functions in a declarative manner i.e. in a way that names what the functions do, as opposed to the original layout of the logic which is an expression of all the computational details of how the computation is performed. The following figure exemplifies this.

```
// A composition of multiple functions to express logic (MazerStatics.cs)
public static Either<IFailure, Unit> StartLevel(int level, Either<IFailure, IGameWorld> theGameWorld, ...)
=>      .Bind(unit => theGameWorld)
        .Bind(gameWorld => gameWorld.UnloadContent().Map(unit => gameWorld))
        .Bind(gameWorld => gameWorld.LoadContent(level, overridePlayerHealth, overridePlayerScore)
                                    .Map(unit => gameWorld))
        .Bind(gameWorld => gameWorld.Initialize())
        .Bind(unit => StartOrContinueLevel(isFreshStart, theGameWorld,
                setMenuPanelNotVisibleFunction, setGameToPlayingState,  setPlayerHealth,
                setPlayerPoints, setPLayerPickups));
```

The code above is an initialization function using declarative style in expressing high-level construction of logic. This uses the LINQ fluent syntax.

This allows the expression of the logic to be represented as a series of independently named functions that describe what will be done. The instantiation of this mechanism is through pipelining these component functions together.

3.1.4 Implementing Predictability

In composing functions as described previously, it is important to look for opportunities to make those constituent functions independent and isolatable as much as possible: this is so that they to not depend on the caller or other state that might render those functions specific or locked into their original locations i.e. non-generic or difficult to re-use elsewhere.

As part of that effort, making constituent functions "pure" and independent of environmental shared state and instead only dependant on data provided to functions via function parameters, pure functions can be used that are more reliable as they do not mutate data that is shared.

An early indicator that a function can be made pure is when the function can be made static, i.e. it has access only to static data (but does not make use of any) and its own function parameters. Making existing functions static is another technique used throughout the codebase, and which is depicted in the figure that follows.

```
// Note: basic types used as input parameters are immutable by default
public static List<Room> CreateNewMazeGrid(int rows, int cols, int RoomWidth,
                                           int RoomHeight)
{
    var mazeGrid = new List<Room>();

    for (var row = 0; row < rows; row++)
    {
        for (var col = 0; col < cols; col++)
        {
            mazeGrid.Add(Room.Create(x: col * RoomWidth,
               y: row * RoomHeight, width: RoomWidth, height: RoomHeight,
                roomNumber: (row * cols) + col, row: row, col: col).ThrowIfFailed());
        }
    }
    return mazeGrid;
}
```

In the code above, a pure function which relies only on its input arguments which are immutable and produces a predictable result every invocation using those parameters.

3.1.4.1 The NULL issue

Many characteristics of NULL lead to unpredictable behaviour to occur within functions, particularly when functions do not expect NULL, rendering those functions unreliable when encountering unexpected NULL values. It is therefor important to eliminate NULL by representing it in a way that it is usable and consistently reliable, and therefore predictable (i.e., consider using Option).

This is done by replacing use of NULL in the codebase and using Option<T> datatypes instead to ensure that when data is NULL, it is easily accountable as Option.None instead. The Option<T> datatype makes it impossible to use NULL without making provision for the possibility of the main scenario being false such being NULL without having to interpret the NULL in the code. This is typified in the figure below:

```
public static Option<T> SingleOrNone<T>(this List<T> list,
    Func<T, bool> predicate) => EnsureWithReturn(()
  => list.SingleOrDefault(predicate))
      // Below the NULL is re-interpreted as a None state
      .EnsuringMap(item => item != NULL ? item : Option<T>.None)
      .IfLeft(Option<T>.None);
```

In the above, the function represents NULL as an Option of <T> such that a NULL is represented as a typed result of type Option<T>.None

This optional type can be either in two states: the data of the original value, or 'None' state if a NULL was encountered here or somewhere else in the pipeline for example.

Generally, incorporating these techniques in a layered approach is the goal: functions need to return reliably; they should be pure and side effects minimized. This, coupled with declaratively composed logic and the flow of data through the simulation of a pipeline yields, many if not all, the goals for using these functional paradigms and techniques.

3.2 Objective 2: Producing *'Understanding Language-Ext'* – a Tutorial

The secondary research objective was to establish a practical means to both understand and try the functional programming techniques and paradigms primarily through the Language-Ext library which was used in MonoMazer.

While the majority of the effort in this research has been focused on implementing functional programming within game code, the majority of the Understanding Language-Ext tutorial was written before undertaking this research. It however has been continuously updated with new contributions throughout the refactoring of MonoMazer as necessitated. This includes new additions to aspects of the Try Monad that is used to great effect in reducing run-time failures by supressing exceptions. Also, effort was made to improve the readability of the commentary used in the tutorial.

An approach of creating individual, runnable code samples linked to the tutorial and forming part of a larger suite of compliable code was taken. This is in the form of a Visual Studio project. A central code repository containing both the tutorial and the associated code was used with additions being made throughout the duration of the research timeframe.

To start the documentation process, the functional paradigms and techniques were identified, and these determined the individual code projects that needed to be created and implement. This provided a roadmap of implementation.

The tutorial is divided into parts: Part I starts with basic functional topics, including Monads, functional composition and pipelining. The later parts culminating in specific usages of Language-Ext constructs such as *Either<L, R>*, *Option<T>* and *Try<T>* to bring about a more orchestrated and practical application thereof.

Each implementation provides a simple example with full commentary to help guide the programmer to understand the utility in the code that is presented.

The fundamentals behind Monads and their specific representation and implementation in Language-Ext is outlined without necessitating prior functional programming knowledge, with each sample presenting a new paradigm or technique that builds on previous illustrations. This is done by threading a continuous commentary-style narrative moving through each tutorial.

Care was taken not to restrict commentary as to produce cleaner code, but effort was given to provide more explanation to the point of repetition to encourage continuity of previously learned concepts.

"*Understanding Language-Ext",* the tutorial as well as its complementary source code are released together and are open source and available online and is free to use and consume at https://github.com/stumathews/UnderstandingLanguageExt

3.3 Objective 3: Analysing Code Complexity

To evaluate the results of incorporating the above functional programming paradigms and techniques, the resulting codebase is compared with the unmodified codebase.

Two branches of the MonoMazer codebase were used to represent the original unmodified code (before) and the newer modified version housing the newly introduced functional programming paradigms and techniques (after) as previously introduced.

Visual Studio's CodeAnalyzer (VSCA) was used to run a static analysis on the source files to provide a report on the effects of the structural changes that have occurred and provide metrics to inform what those effects mean in terms of Code Complexity.

Three primary measures of complexity were used: McCabe Cyclometric complexity (MCC), Maintainability Index (MI) and Class Coupling (CC).

These generally provide a numeric value for functions in the source code, indicating its value-rating. An overall value is used which represents a cumulative aggregation representing all the code.

Only the files within *MazerPlatformer* project namespace was analysed as this represents the main source code and excludes 3rd party code.

A comparison of the complexity metrics both before and after is discussed in the Results.

Chapter 4

Results

4.1 Objective 1: Refactoring MonoMazer using FP paradigms and techniques

The refactoring process was carried out on a separate code branch to the original implementation of MonoMazer.

Play-testing and later unit-testing (introduced in the beyond branch) was used to validate that the game's functionality and overall behaviour and gameplay remained unchanged.

There are 3 points of comparison: before the FP paradigms were introduced (represented by the before branch), directly after they have been introduced (after branch) and beyond, which represents a continual effort to incorporate the FP paradigms into the codebase (beyond branch). The after branch is a snapshot in time but as time has progressed more refactoring has been done and this is now best represented by an extension of it called the 'beyond' branch.

For ease of comparison the before branch is used and beyond branch is used in place of the after branch when considering 'before' and 'after' comparisons. The beyond branch is considered a matured version of the after branch, as it contains further progress in incorporating FP into the codebase

The source code for MonoMazer is open source and is made available at https://github.com/stumathews/MonoMazer.

The before and beyond branches represent the imperative and the functional approaches respectively for comparative purposes.

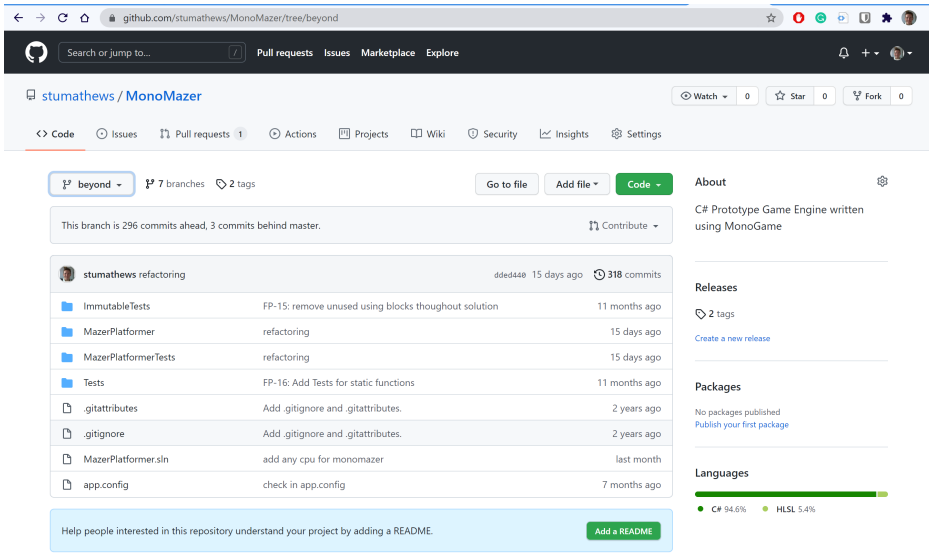

Figure 4.1: Location of the MonoMazer code in GitHub

The refactoring of MonoMazer is discussed later.

4.2 Objective 2: Understanding language-Ext – Tutorial

The tutorial is open source and is made available at: https://github.com/stumathews/UnderstandingLanguageExt

The tutorial has helped reduce the scope of this research immeasurably and has helped focus on the code implementation of the concepts previously discussed as this is where most of value is, and the primary focus of this project.

Many concepts were revisited and edited during the refactoring and the tutorial is seen as an accompaniment to the main research.

The code samples, numbered as tutorials are listed below:

Table 4.1: Code samples referred to in the tutorial

Tutorials
Tutorial01 - Introduction to the Box type (a Monad)
Tutorial02 - Shows you how to use Map and Bind (also construction of a Monad Type)
Tutorial03 - Shows how Bind is used to create a pipeline of function calls
Tutorial04 - using Map() and Bind(), Select() and introducing SelectMany()
Tutorial05 - More examples of performing operations on a Box
Tutorial06 - This tutorial shows you how pipelining is used using LINQ Fluent and Expression syntax
Tutorial07 - Shows you when to use Map() and Bind()
Tutorial08 - transition from Imperative style coding to Declarative style coding with an example
Tutorial09 - Shows you that pipelines include automatic validation
Tutorial10 - Expands on Tutorial09 to show that transformation function always return a Monad
Tutorial11 - Shows how monad built in validation, affords short-circuiting functionality.
Tutorial12 - Composition of functions
Tutorial13 - Pure Functions - immutable functions with now side effects i.e. mathematically correct
Language.Ext : Either<Left,Right> Basics:
Tutorial14 - Shows the basics of Either<L,R> using Bind()
Tutorial15 - Using BiBind()
Tutorial16 - Using BiExists()
Tutorial17 - Using Fold() to change an initial state over time based on the contents of the Either
Tutorial18 - Using Iter()
Tutorial19 - Using BiMap()
Tutorial20 - Using BindLeft()
Tutorial21 - Using Match()
Operations on Lists of Either<left, Right>:
Tutorial22 - Using BiMapT and MapT
Tutorial23 - Using BindT
Tutorial24 - Using IterT
Tutorial25 - Using Apply
Tutorial26 - Using Partition
Tutorial27 - Using Match
Option Basics:
Tutorial28 - Introduction to Option
Tutorial29 - Basic use-case of Option
Tutorial30 - Using Option in functions (passing in and returning)
Tutorial31 - using IfSome() and IfNone()
Tutorial32 – Full pipelining example
Tutorial33 - Using ToEither<>
Tutorial34 - Using BiMap() - see tutorial 19
Custom Specific
Tutorial35 - ThrowIfFailed() and introducing Either<IAmFailure, Option>
Tutorial36 - Using custom extension method FailureToNone()

Tutorials
Tutorial37 - Using pure functions to cache
Tutorial38 - Changing state of an object over time using FP
Tutorial39 - Immutability
Tutorial40 - Try - Supressing Exceptions
Extra:
TutorialA - Partial Functions
TutorialB - Threading and parallelism benefits
TutorialC - Hub-and-Spoke distribution model
TutorialD - Custom useful Monad Extensions

These are included within the *"UnderstandingLanguageExt"* Visual Studio solution:

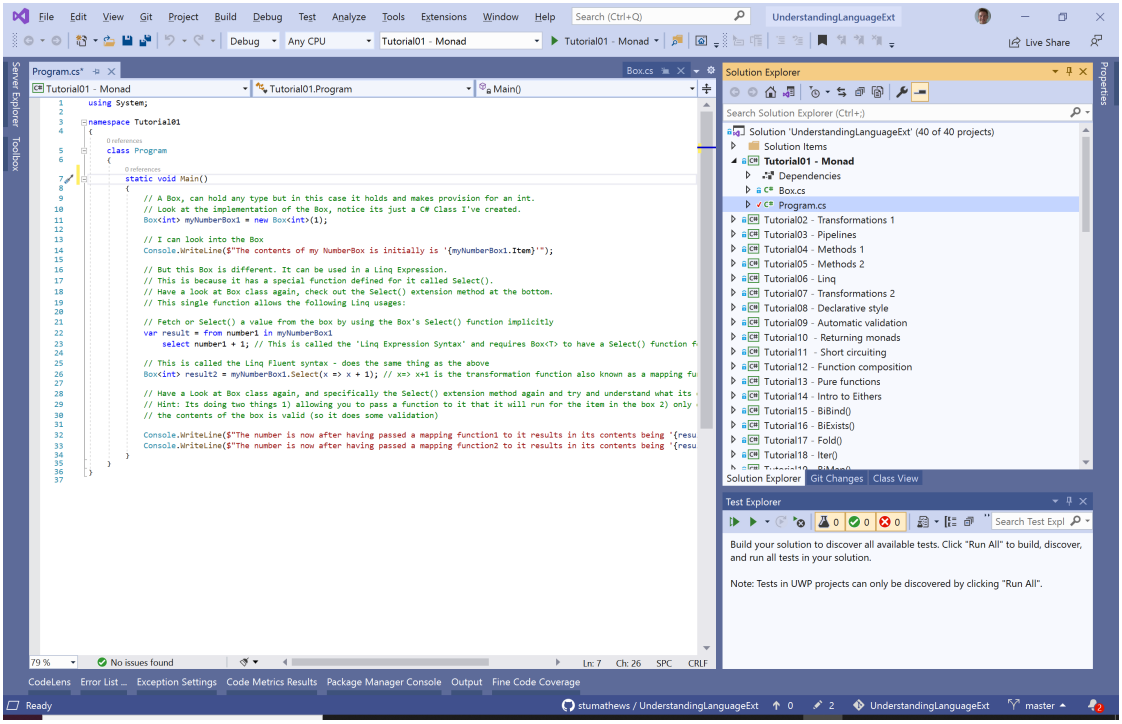

Figure 4.2: Individual projects housing executable code demonstrating functional paradigms and techniques covered in tutorial and used in MonoMazer

4.3 Objective 3: Code Complexity

Code Complexity is primarily measured using Cyclomatic Complexity however measures including Maintainability Index, Class Coupling are also gathered for analysis. Additionally, the number of Lines Of Code (LOC) including and excluding whitespace is analysed to investigate the effect FP has on code conciseness both per function and per type. Executable Lines of Code (ELOC) does not take whitespace into consideration.

Only the main "MazerPlatformer" C# namespace is considered when calculating the metrics.

There are 3 main areas of comparative distinctions that are relevant: comparing changes in original data, the contribution of new data and finally the analysis the contribution across all the files, i.e., the effect of both original and new data combined.

4.3.0.1 Original data

In analysing the changes in the original data, it is possible to perform a like-for-like comparison of the effect that the FP refactoring has had before and after the FP changes have been introduced. Original data represents the main aspects of the game code and include the key areas of the game such as the game loop, gameworld and level code etc.

4.3.0.2 New data

Analysing new data provides a current and specific view of how new code affects the existing codebase. This is code that enhances the original code and is purely a consequence of the refactoring effort.

4.3.0.3 Future data

Recognising that in the future the codebase is likely to contain variances in the statistics that are difficult to predict now, analysing the average of all the data provides a reasonable general estimate of the effects on the future of the codebase, had it to continue being refactored using functional paradigms and techniques in the way proposed in the methods. This is achieved by regularizing current variances (outliers) and treating all data as equally important.

This provides a valuable generalized indication of how the effects may or are generally applicable to any future code that is yet unwritten and is introduced and therefore provides a relatively useful insight into a reasonable future trajectory of the effects of the FP approach moving forward.

4.3.1 Overall cumulative complexities

The main results of the static analysis using VSCA (Visual Studio Code Analyser) on the before and after (beyond) branch are summarised below:

Table 4.2: Percentage difference in metrics as reported by VSCA on 'Before' vs 'After' (beyond) branches

Overall cumulative metric B	efore A	fter D	ifference C	hange in %
Maintainability Index	91	91	0	0 %
Cyclomatic Complexity	578	877	+ 299	+ 51.73 %
Class Coupling	133	218	+ 85	+ 63.91 %
Lines Of Source Code	2762	4345	+ 1583	+ 57.31 %
Lines of Executable Source Code	911	1919	+ 1008	+ 110.65 %
Functions	312	1124	+ 812	+ 260 %
Classes	45	88	+ 43	+ 96 %

These initial results are discussed later in the 'Discussion' section that follows later.

4.3.2 Comparing average effects per function and per class (type)

Apart from assessing the metrics as reported by VSCA, the effect on the average function or typical type is also considered. The primary reason for this is that it became clear that with a 260% increase in function count in the functional codebase, that overall cumulative measures such as Cyclomatic complexity appears to be disproportionately correlated to the increase in number of overall functions. This makes sense because each function is automatically given 1 point plus whatever the underlying complexity of that function is, i.e. the number of non-linear pathways it has. This effectively appears to cumulatively penalise codebases with more functions.

This means codebases with more functions (such as a functional codebase) will yield a large overall cumulative cyclomatic complexity, while traditional imperative codebases will have a more modest rating.

For this reason, assessing the effect on the 'average' function in both codebases is done to investigate a potentially less biased measure of cyclomatic complexity in these circumstances. Performing a further analysis will also help reveal the nature of the underlying VSCA data.

In this way we consider metrics that represent the 'average' function as well as the average type. Here type is merely a localised aggregate of functions that appear within it. Considering the type allows us to evaluate the impact of change in specific areas of the game as each type usually represents a specific aspect of the game. This also allows for seeing where data varies and the relative contributions it makes.

4.3.2.1 The average existing, new and overall complexity

In considering the above metrics for the average function or type, various perspectives are useful:

- After vs Before: represents a comparison of the impact on the average existing function/type before and after refactoring, i.e. the changes in the shared code before and after.
- New vs Before: represents a comparison of the impact on the average new function/type (New) compared to existing or old functions/types (Before).

- Before vs Now (overall): represents a comparison of the impact on the overall average function/type (Now) code compared with the unmodified existing code (Before).

4.3.2.2 Maintainability Index

In assessing the impact to existing functions i.e., those present both before and after the refactoring, the Maintainability Index (MI) has increased slightly by 0.12% (0.11 points) with a median change of 0% (or 0 points) which suggests technically that generally on average main areas of the game's functions are slightly more maintainable. This on average however is not significant at less than a percent, but there are specific cases where maintainability has changed:

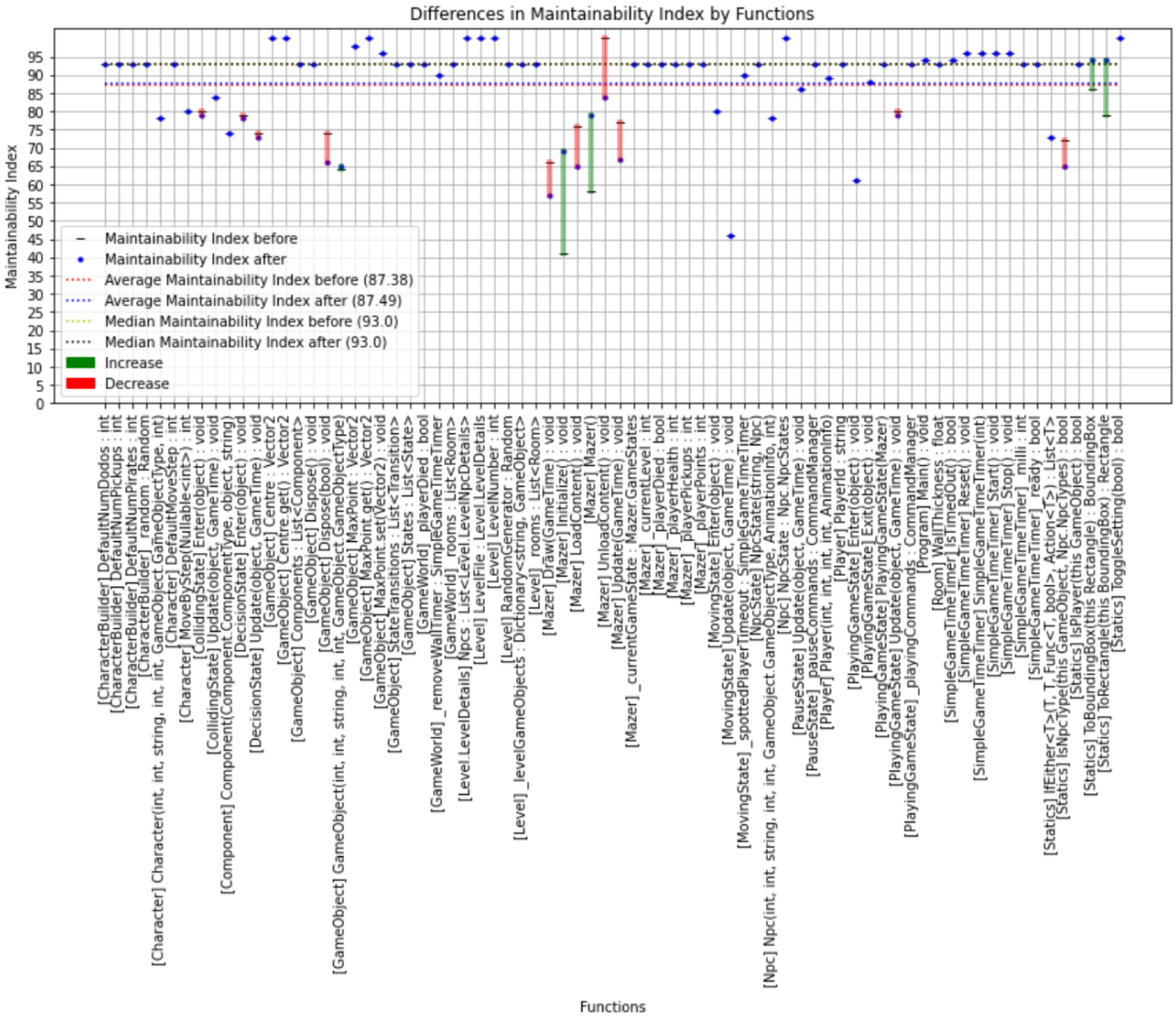

Figure 4.3: Changes in Maintainability Index in original functions

When considering the average function based on all functions, i.e. both existing and newly introduced, there is only a marginal decrease in MI on average of 1.57% or 1.37 points and a median decrease of 3.23% or 3 point per function, technically suggesting that the overall impact on average is slightly worse. This, like the impact on the average existing function is similarly negligible:

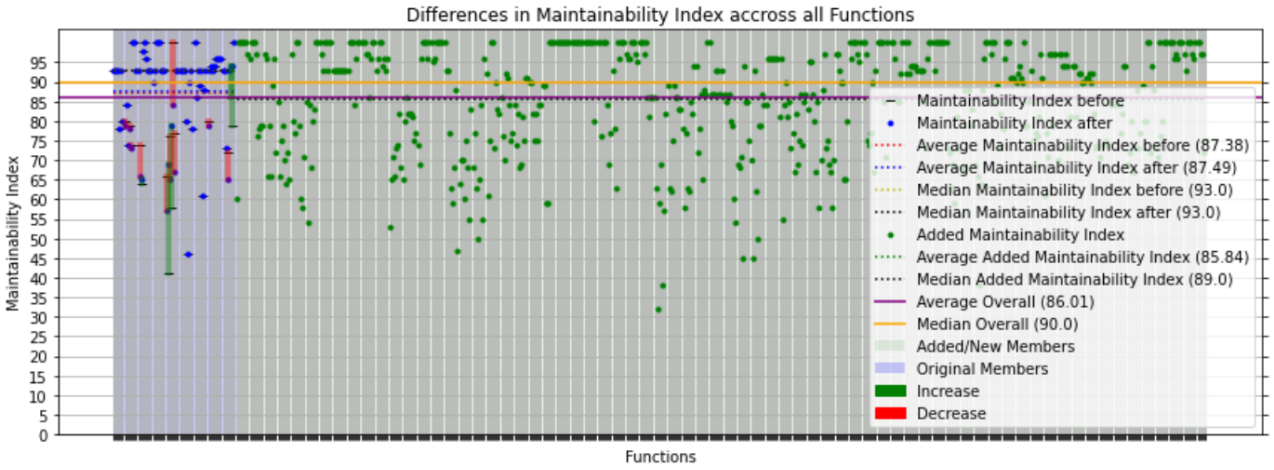

Figure 4.4: Maintainability Index in functions

When considering MI of the average existing type, the results show that in existing types on average the MI has decreased slightly by 0.11% or 0.1 points and a median decrease of 11.1% or 1 point. This too suggests that the main types of the game have also seen a slight reduction in maintainability. Again, this average reduction is a negligible amount and we discount its significance.

Considering the average type across existing and new types, MI has increased on average by 4.94% or 4.31 points and a median increase of 5.56% or 5 points in new types. This suggests that new types are more maintainable on average.

The overall average type, considering all types in the game, has increased MI by 2.91% or by 2.54 points and an increase in median of 4.44% or 2.54 points. This suggests that generally the codebase has become slightly more maintainable as more types have been added to the game.

These results overall suggest that Maintainability Index impact on the average function is negligible ($< 5\%$). The approximately 3% increase in average type overall is also considered to be comparably low, however it is improved over existing types:

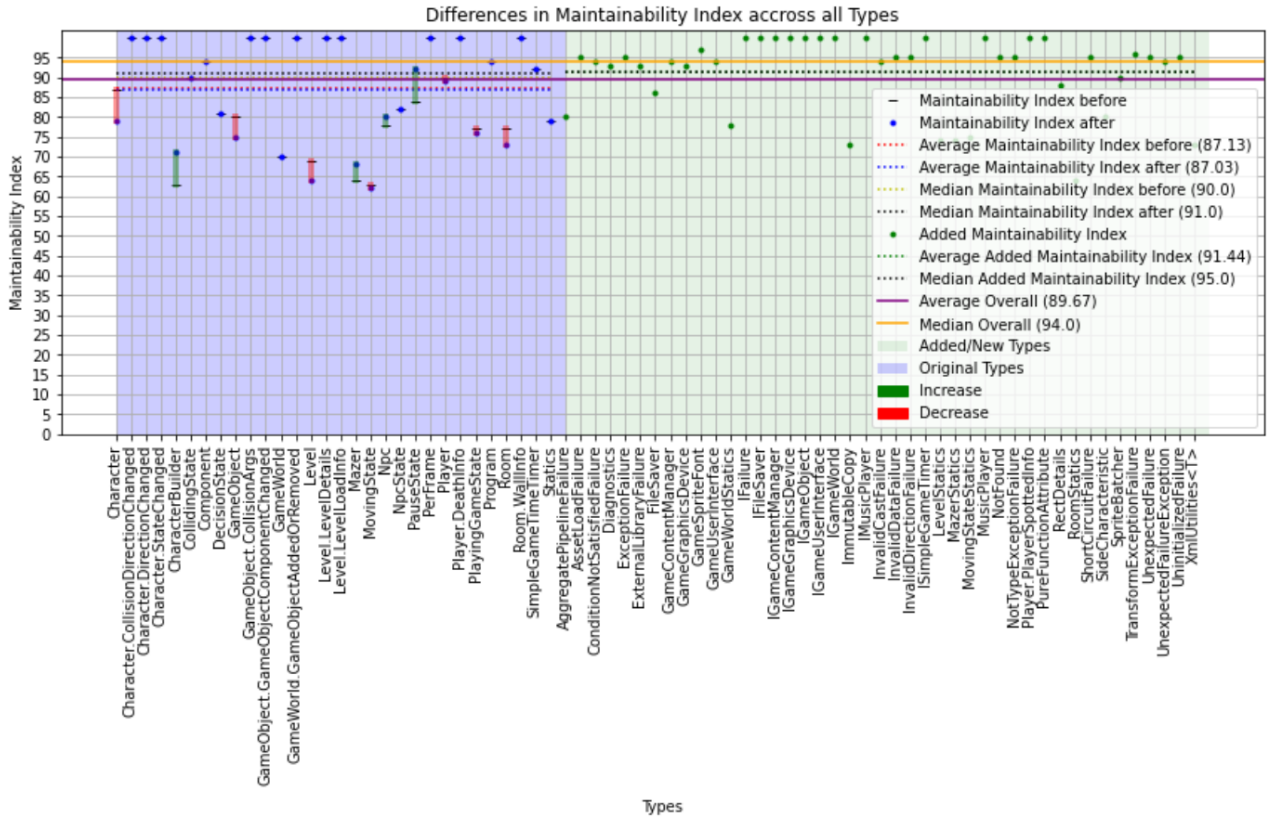

Figure 4.5: Maintainability Index in Types

4.3.2.3 Cyclomatic Complexity

The average function across existing functions show a decreased in Cyclomatic Complexity(CC) by approximately 21% or by 0.22 points (0% median change(1.0)). This suggests that FP refactoring on the main parts of the game functions on average have actually contributed positively.

This means that the main areas of the game have become less complex on average in terms of the number of linear pathways through the code and this represents an improvement to the main aspects of the game.

The existing areas of the game that showed the biggest improvements were in the update loops in the GameObject and the Moving state, code that disposes the game object, the main drawing function in Mazer and the code that determines the type of NPC type. The median perspective discounts this however, potentially owing to outlier influence however reductions (improvements) can be clearly seen in the figure below:

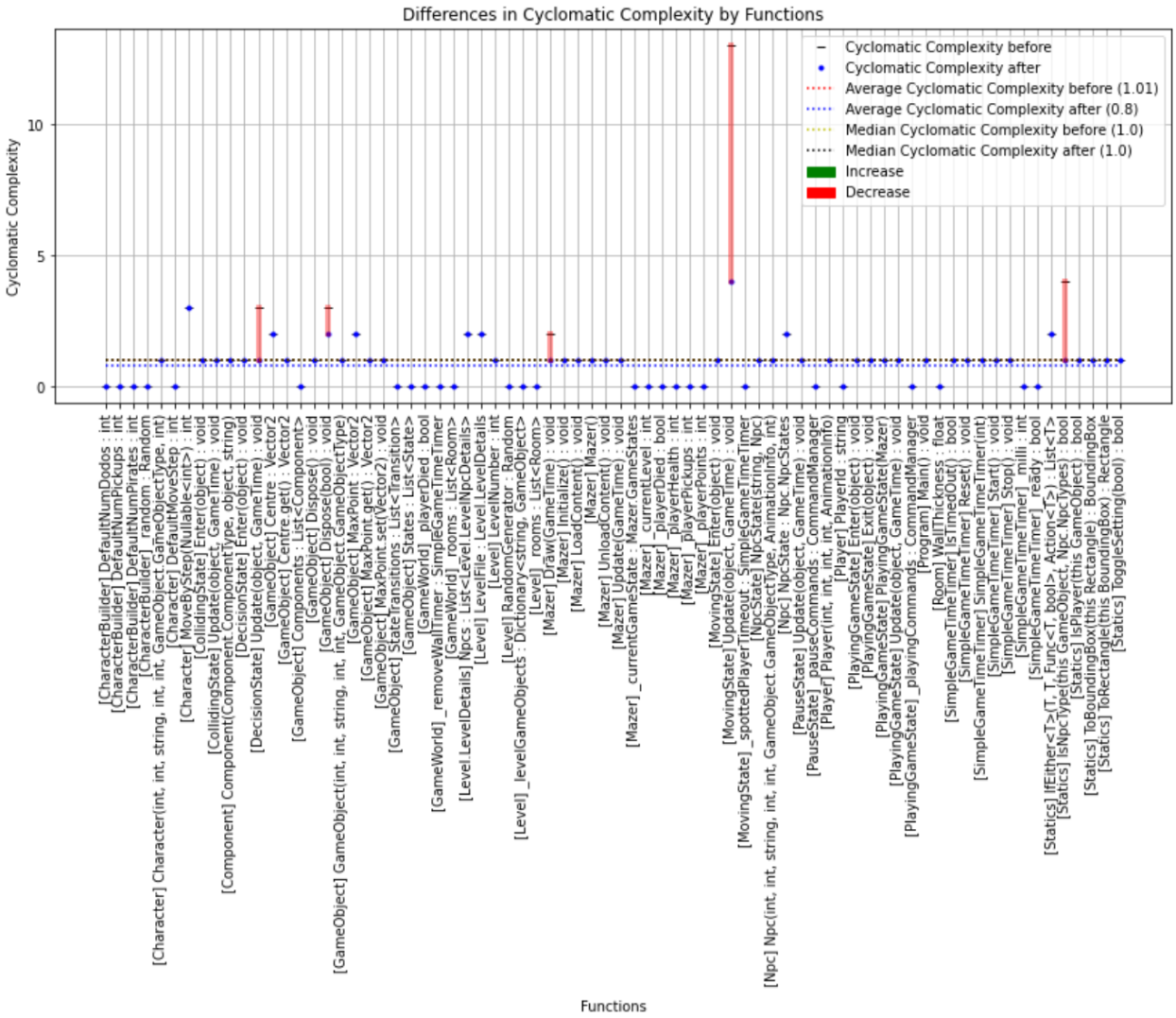

Figure 4.6: Changes in Cyclomatic Complexity in the existing functions only

The average new function has increased by 28% or 0.28 points (1.01 -> 1.3) and no change in the median when compared to existing functions. This suggests that new functions on average are created with 28% higher complexity than the existing game code.

It is unlikely that the increase in the number of total functions would contribute to the higher average across new functions because as an average this figure should decrease as there are more divisors to regularize the impact. It is therefore more likely that the actual functions are indeed more complex as the metric suggests. The median perspective suggests that there is no change.

The average function from all available functions (new and old), shows that there has been an increase by a comparatively similar amount on average by 25% (1.01 -> 1.27) and again, no change in the median:

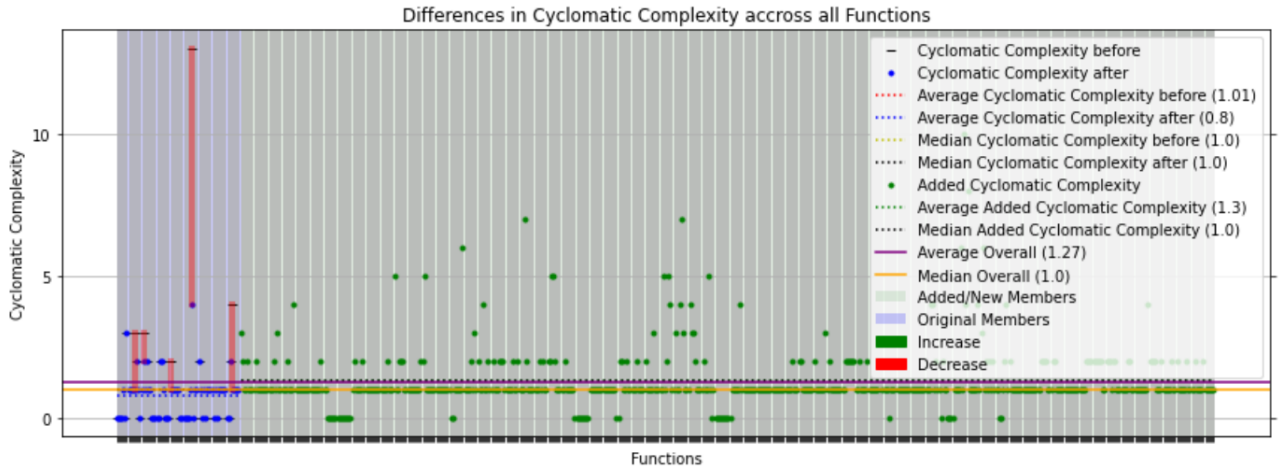

Figure 4.7: Changes in Cyclomatic Complexity in functions

When considering the complexity of the average existing type, the results show that there has also been an decrease in complexity, and that they are 6% or 1.17 points less complex (18.13 to 19.97) but an decrease in the median of 10% or 0.5 points :

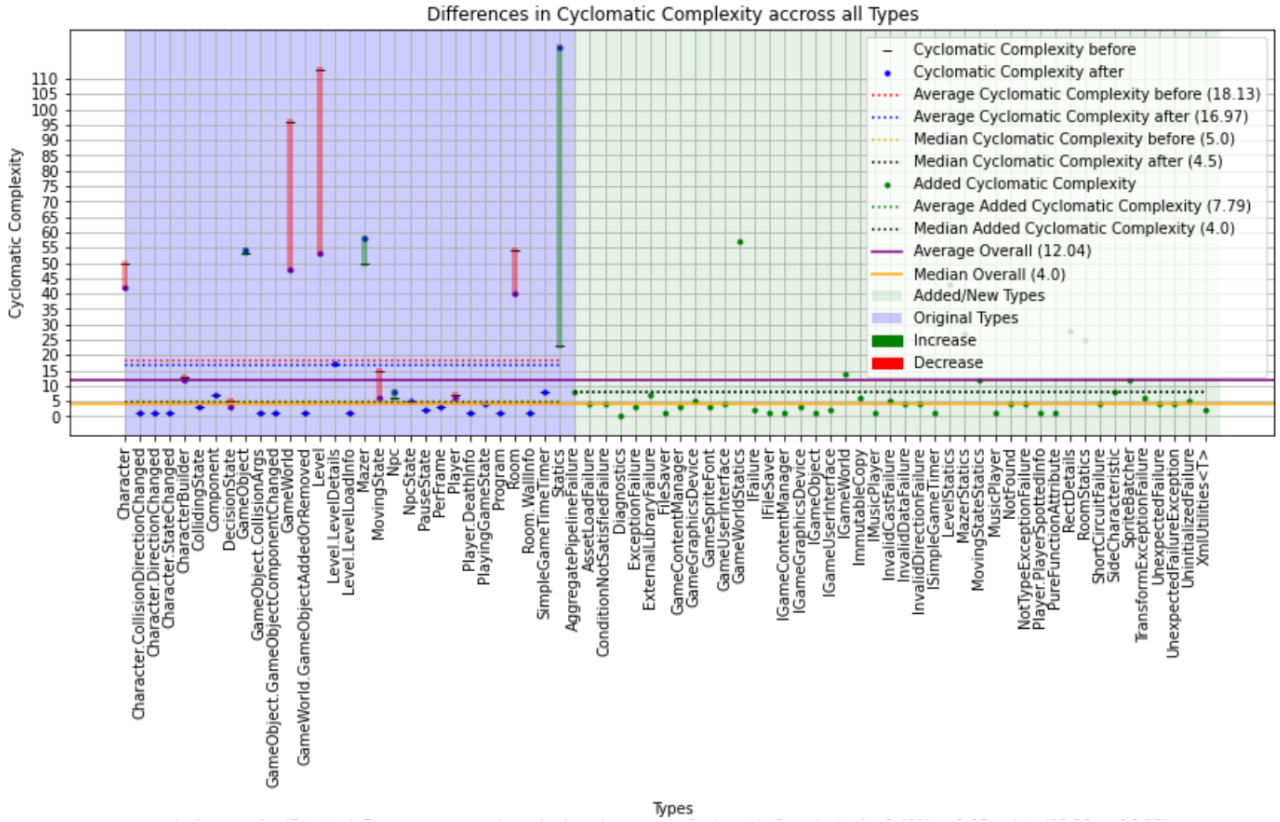

Figure 4.8: Changes in Cyclomatic Complexity in types

Also, the average new type has decreased in Cyclomatic Complexity by 57% or by 10.34 points (18.13 -> 7.79) and a 20% or 1 point decrease of the median, with the game world, level code and room logic has seen a significant decrease in average complexity, while the statics class has a significant increase in complexity.

The statics type is where all the pure functions are located in. The static types also contain the most number of functions and therefor show an disproportionate increase in cumulative complexity. They are outliers.

The average new type shows that there has been an overall 34% decrease in CC (18.13 -> 12.04) and associatively a 20% decrease of the median (or 1 point).

This suggests on average future classes/types are likely to be less complex, despite the overall average function being more complex. This is expected as types are an aggregate measure of the containing functions which reduce the average by increasing the divisors (number of functions).

It does however also show that pure functions that are moved to a single type such as Statics are likely show a sharp increase in complexity as the number of functions in the type increase.

4.3.2.4 Class Coupling

The data shows that there is an increase of 24% or a 0.59 point increase in Class Coupling in the average existing functions (from 2.43 to 3.03) and a 50% increase or 0.5 points of the median, indicating that the FP refactoring has made the main functions of the game more coupled, however the Mazer's initialization and construction code has improved. The effects can be seen below:

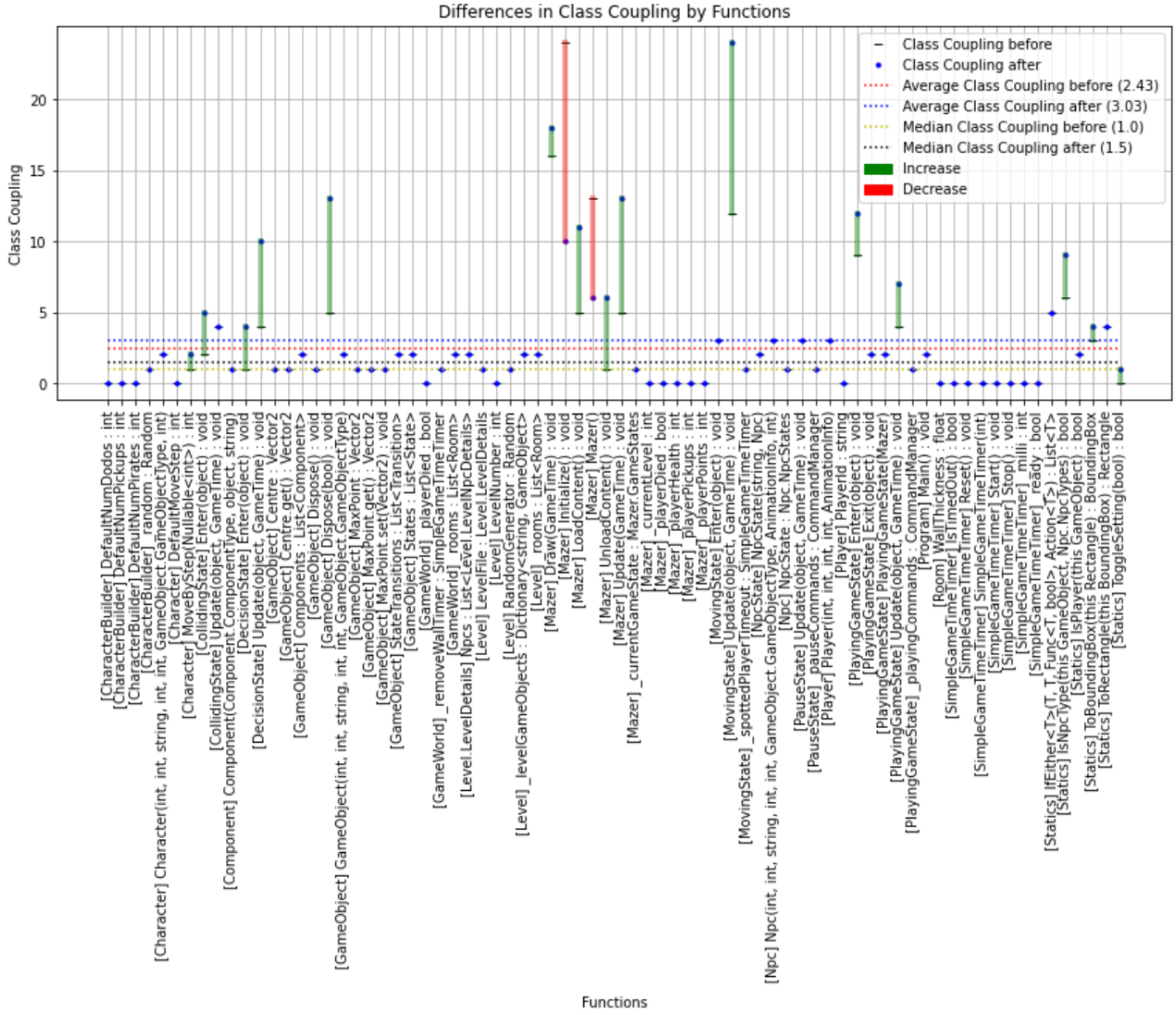

Figure 4.9: Changes in Class Coupling in existing functions

This increase in coupling is more noticeable when considering the average function across new functions which show an average increase of 101% when compared to original functions (2.34 to 4.9) and a 300% increase or 2 points of the median.

Consequently, this influences the average function across all available functions which shows an increase of 90.04% or 2.19 points (2.43 to 4.62) and a 200% increase or 2 point increase of the median, and shows equally that new functions house most of the increased class coupling.

Future functions are thus reasonably likely expected to have a higher class coupling around at approximately 4.62.

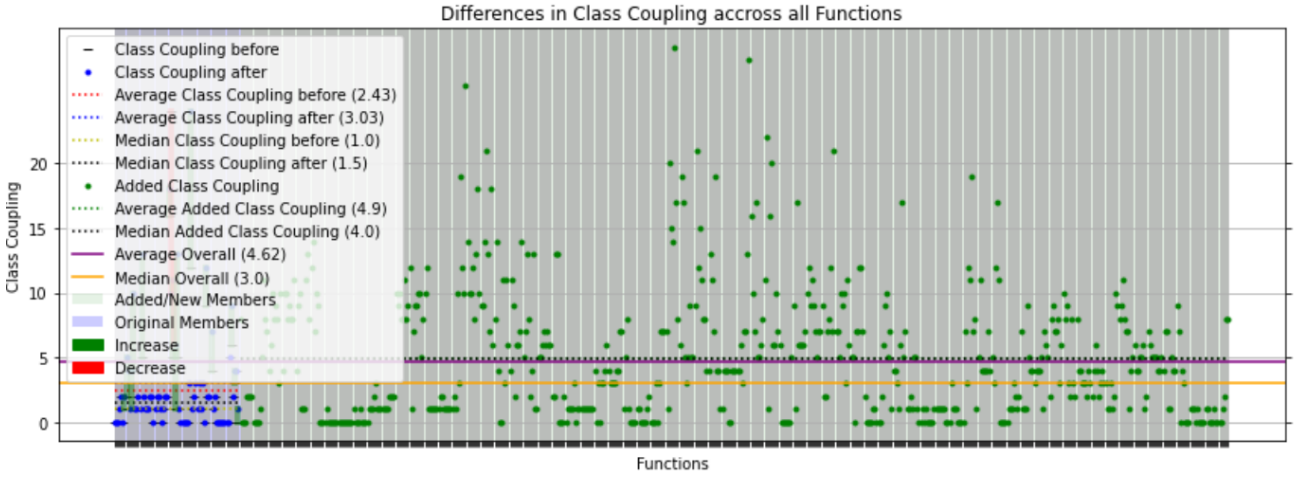

Figure 4.10: Class coupling in functions

Considering the average existing type, it has also increased, by 59% or by 7 points (11.9 to 18.9) and an increase of 87.5% or 3.5 points of the median, suggesting that refactoring existing types has increased the class coupling as is the case with existing functions. An increase in coupling here is consistent with what the average existing function experiences:

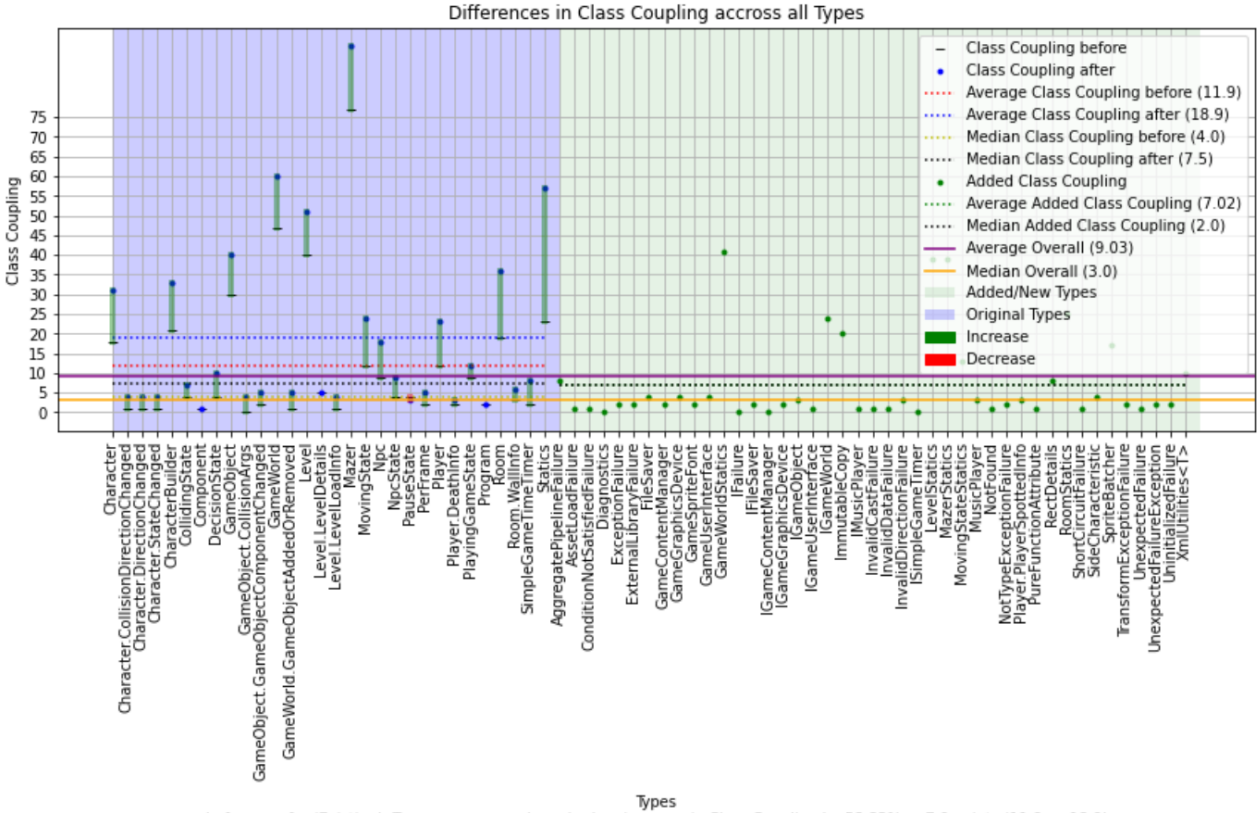

Figure 4.11: Class Coupling in types

Also, the average new type introduced shows a lower class coupling by an average of 5 points, a reduction of approximately 41% when compared to the average of the average existing type of 11.9. (11.9 -> 9.03) and a decrease by 50% or 2 points of the median.

This influences the overall average type which shows that for class coupling across all types (new and old), it sees a 24% or 3-point decrease and a 25% reduction or 1 point of the median(4 pts -> 3pts).

Noticeable is the significant increases in all the new types that house the new pure functions, i.e. the static types.

Considering areas of the game: there is a specific and significant Increases in Statics, moderate increases in Room, Level, GameWorld, Character, CharacterBuilder, Player, Npc, MovingState.

Minor increases occur in most existing Types, while no change in Component, LevelDetails and Program.

4.3.2.5 Lines of Source Code (LOC)

There is approximately a 10% or 0.49 line decrease LOC (4.99 -> 4.5) in the average existing function (main areas of the game) but a 50% increase in the median by 0.5 points , while new functions see a 0.89% or fractionally small 0.04 line increase (from 4.99 to 5.03 lines) but which increases the median by 1 point(1pts -> 2pts), which in turn influences the overall average function's LOC to increase by 0.79% or 0.04-lines (499 -> 5.03), equally seeing a median increase by 1 point (2pts -> 1pts).

This suggests that the average existing function does not show a significant increase or decrease in lines of code, or at most 1 extra line when considering the median of the existing functions. There is however a specific and significant decrease in the initialization routine as well as the constructor in Mazer.cs:

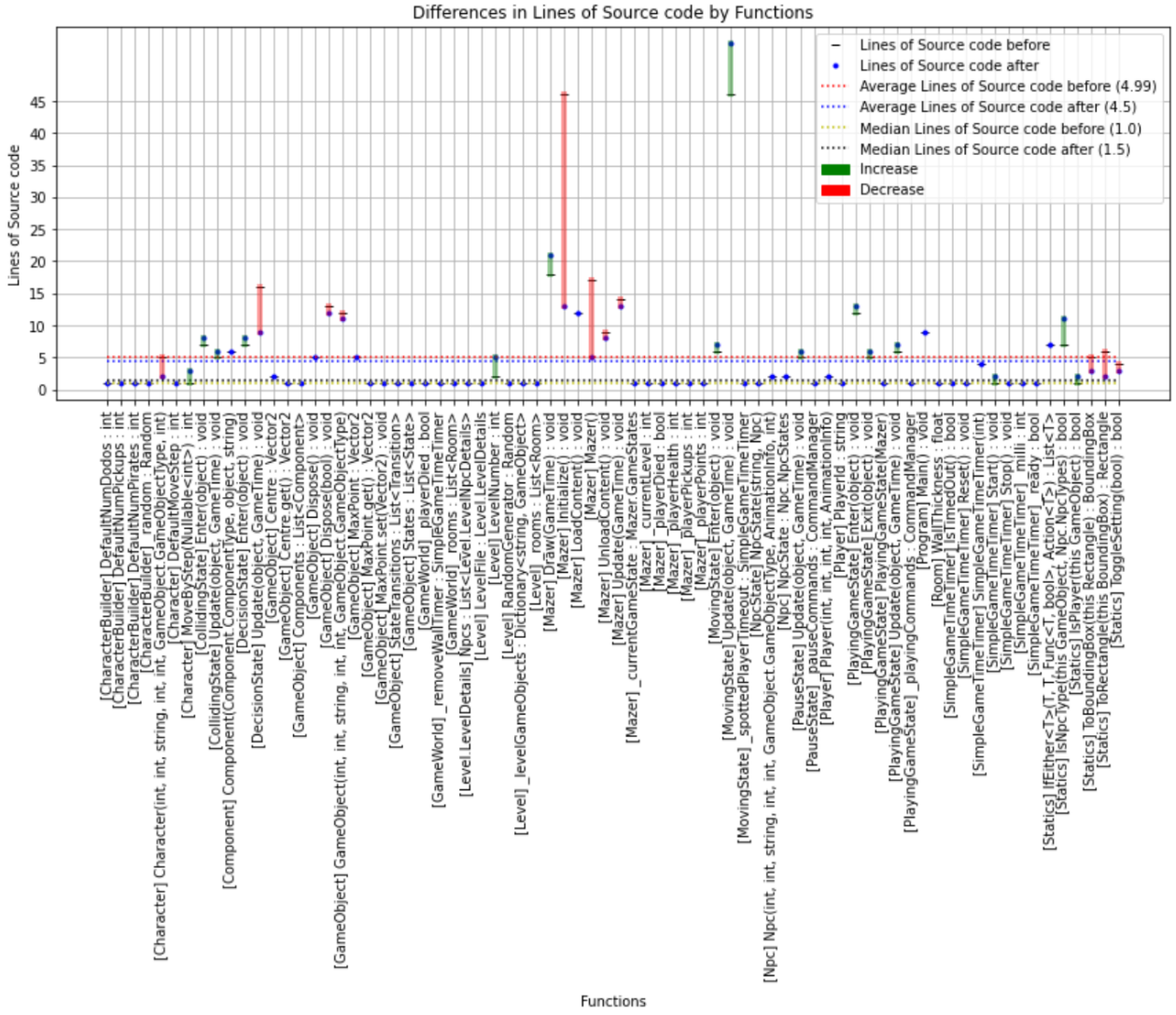

Figure 4.12: Lines of Source Code in existing functions

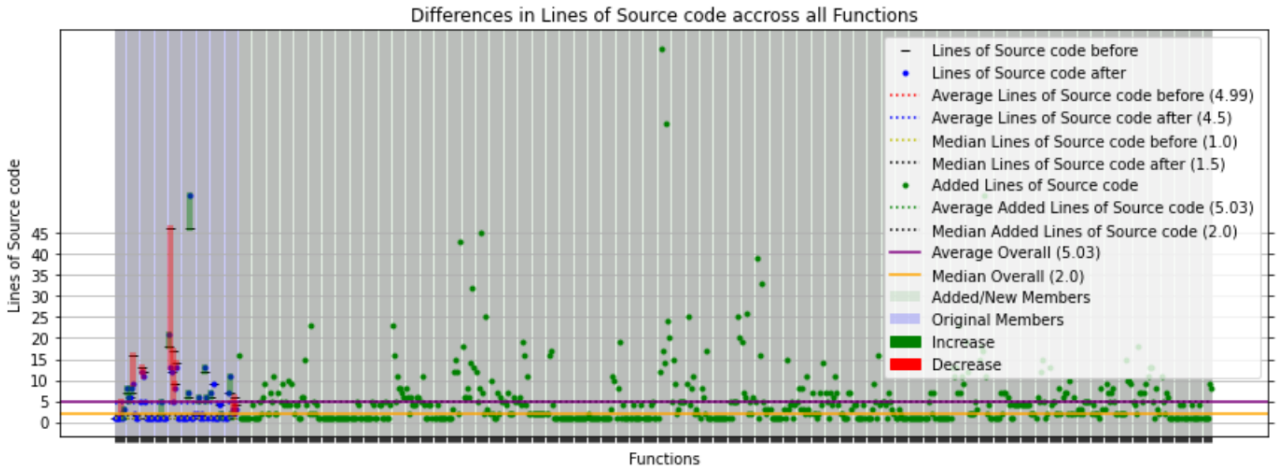

Figure 4.13: Lines of Source Code in Functions

The average existing type however shows an increase of 8% or 17.17 lines (90.57 -> 97.73) and an unchanged median,

suggesting that the FP refactoring on average has added more lines to the main areas of the game instead of reducing them and when considering the median suggests that there has been no improvement at all.

The average new type contribute significantly lower lines of source code than existing types by 68% or 61 lines (90.57 -> 29.33) and a reduction of the median by 56.52% or 13 points (23pts -> 10 pts).

Across all types (new and existing), the average type shows an overall decrease in LOC by approximately 40% or by 36 lines (90.57 -> 54.49) and respectively a 34% reduction or 8 points in the median.

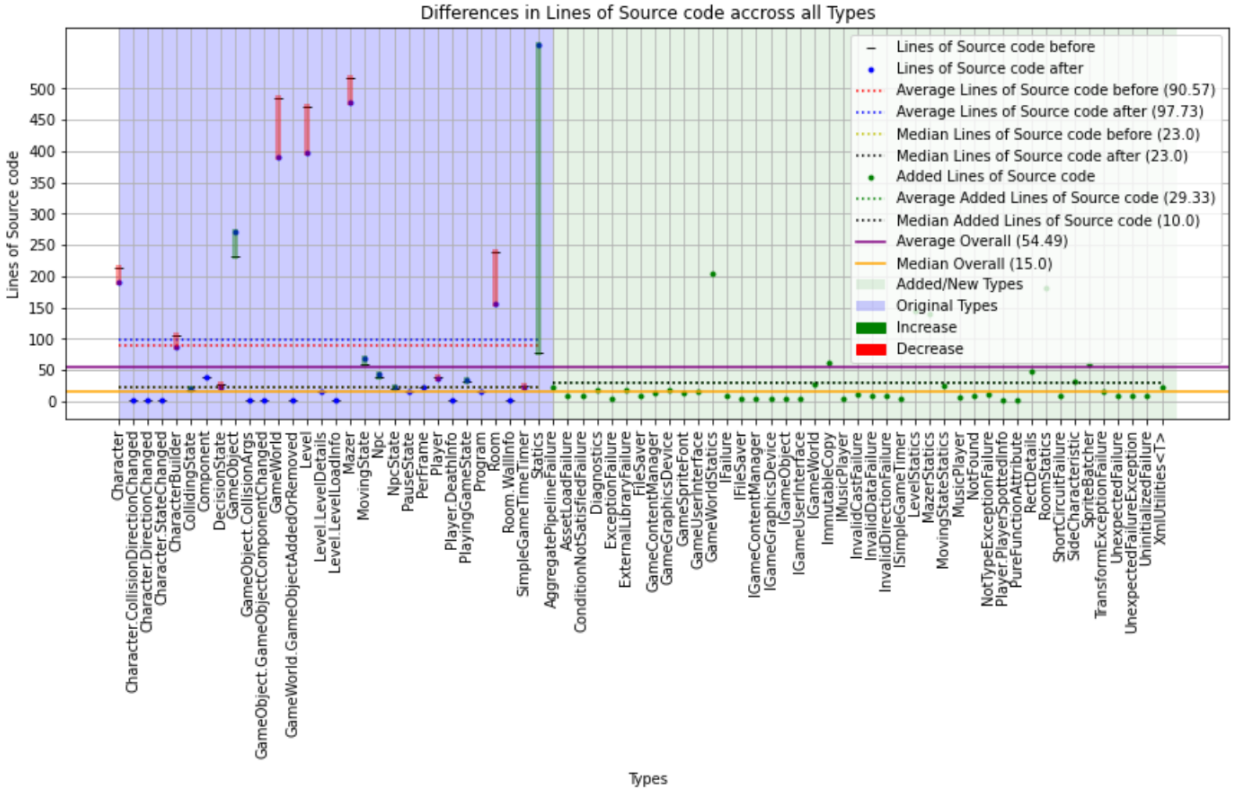

Figure 4.14: Lines of Source Code in Types

4.3.2.6 Executable Lines of Code (ELOC)

There is an 15% decrease or 0.45 lines on average in existing functions (3.02 -> 2.58). This shows that the decrease in average existing function has not decreased significantly. There has however been, as seen with LOC a significant and similar reduction in ELOC compared to LOC the Mazer' initialization and construction routine:

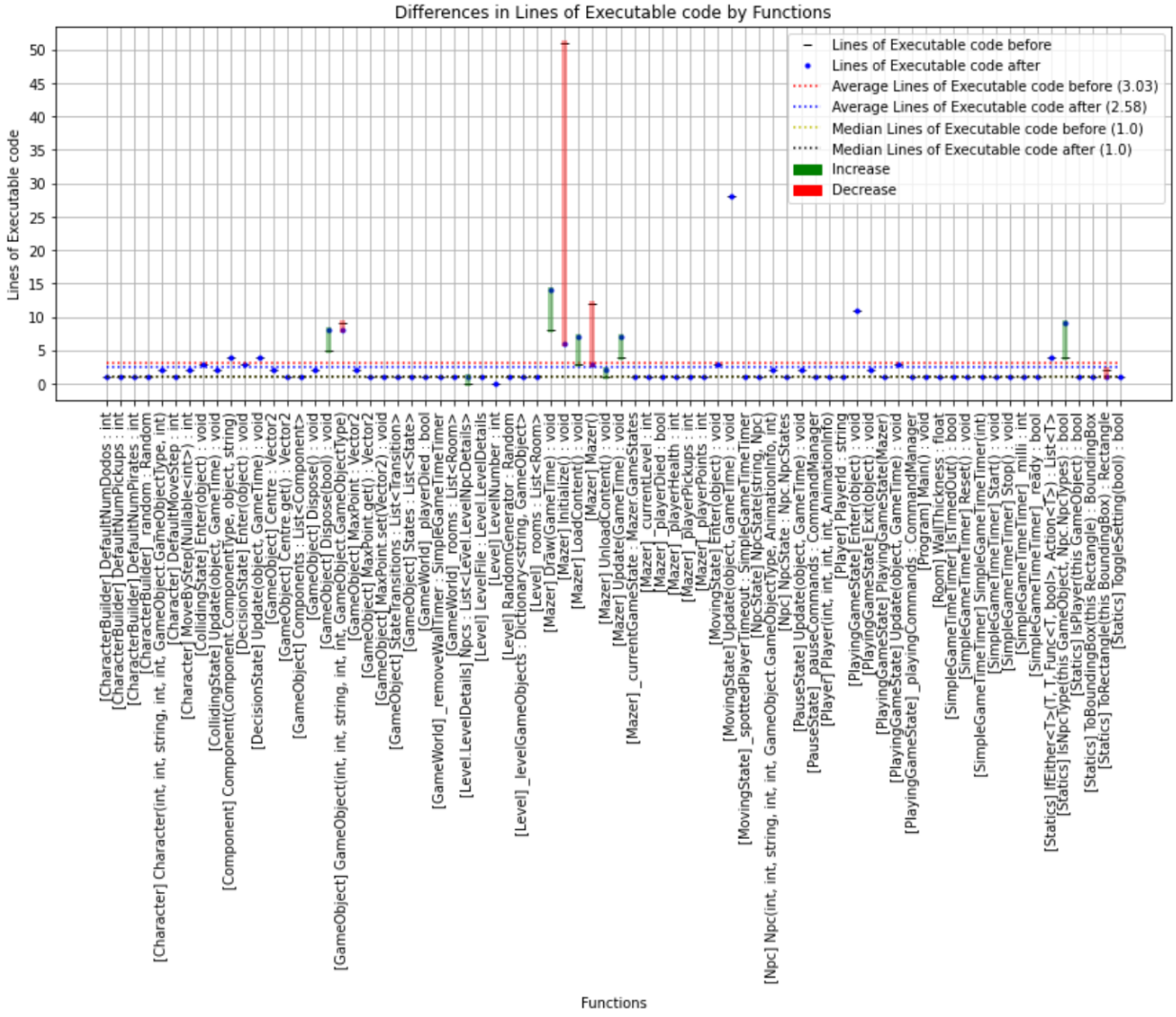

Figure 4.15: Changes in Executable Lines of Source Code in original functions

The average new function shows a 0.16% which is virtually a 0 line decrease in ELOC in existing types (3.03 -> 3.02). There is similarly a near zero decrease or 0.14-lines on average overall across all classes (3.21 -> 3.35).

This shows that on average new functions (and the median) of the game are not significantly impacted in ELOC due to the FP refactoring, and suggests that the overall increase as indicated by the cumulative results (110%) do not carry over to the view of the average function which might be more influential within a function-heavy functional codebase:

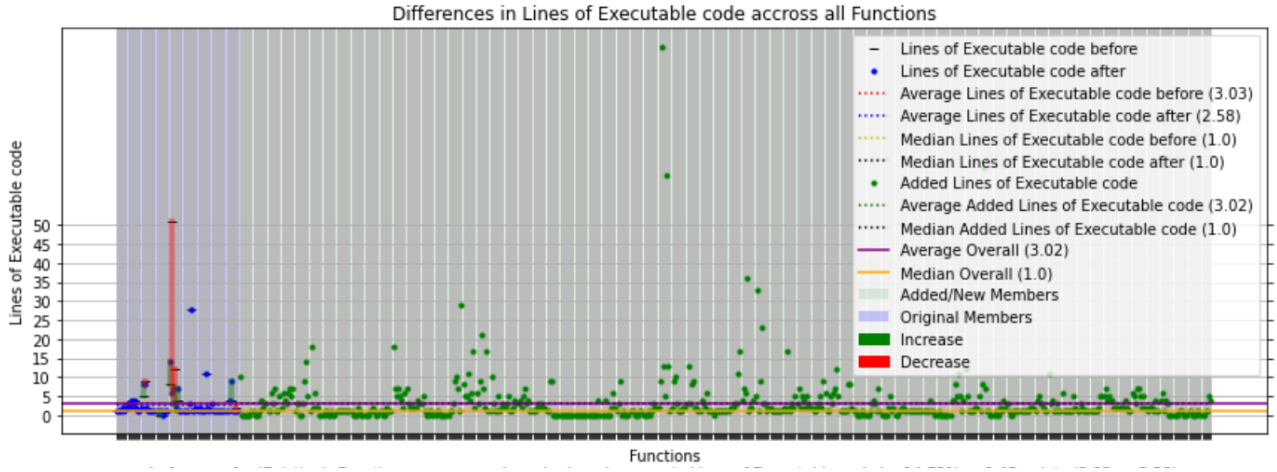

Figure 4.16: Lines of Executable Code in Functions

Considering the average type and like the average LOC for existing types, the average existing type increased in ELOC by 55 % or 17 lines (30.23 -> 46.8), showing again that the FP refactoring has increased the amount of code, particularly in the existing types in the game.
This is expected likely due to the increase in the number of functions within types.

The main areas seeing significant more code is in GameWorld, Level, Mazer, Statics and a moderate increase in GameObject. There are minor increases in Character, MovingState, Room. This can be seen in the figure below.

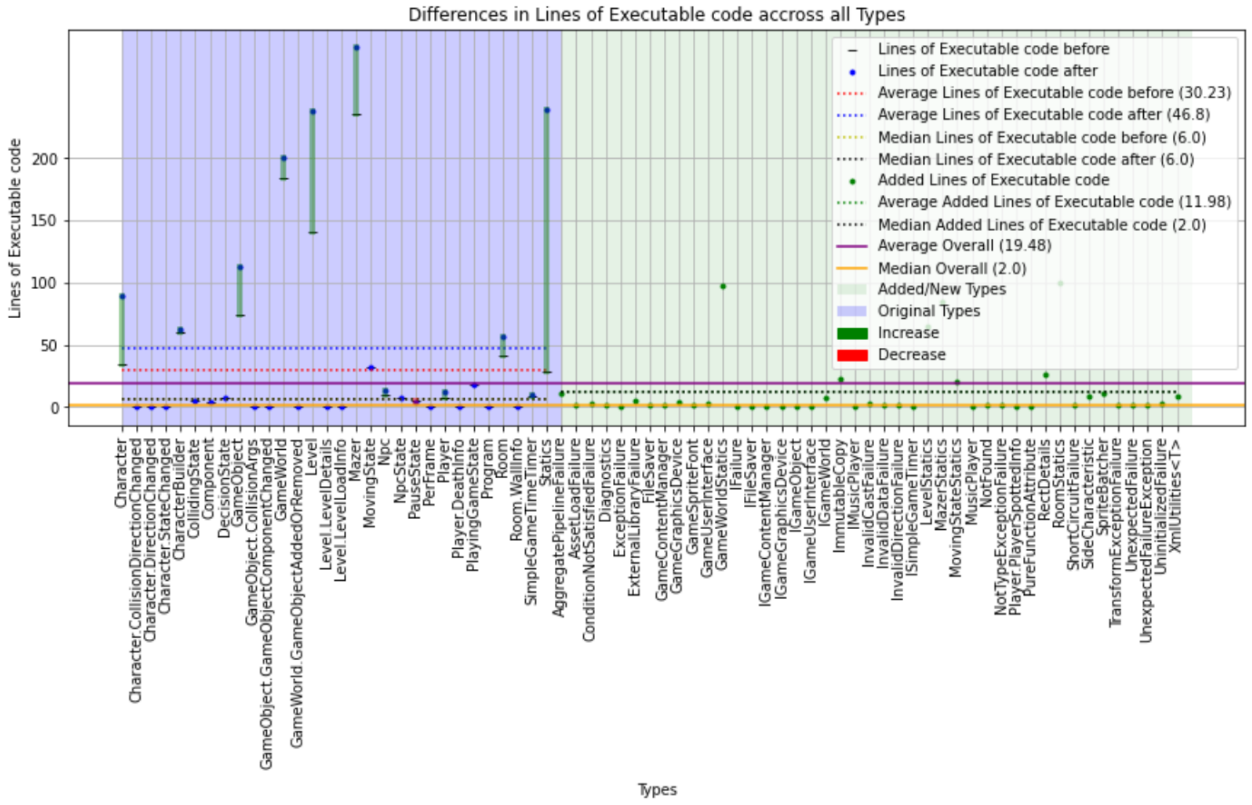

Figure 4.17: Lines of Executable code in Types

Also, new types on average are 60% or 18 lines lower in ELOC than existing types (30.23 -> 11.98) and 66% or 4 points lower in the median (6pts -> 2pts) and the average type across all types, also sees a decrease in ELOC but by

35 % or 10 lines (30.23 -> 19.48) which corresponds to a 66%.67 reduction in the median by 4 points (6pts -> 2pts).

Chapter 5

Discussion

5.1 Objective 1: Refactoring MonoMazer

5.1.0.1 Refactoring large portions of unsafe code to return Eithers

It is time consuming to wrap every function within a Try<A> Monad statement (to avoid exceptions)

An approach was used to wrap unsafe code i.e. that could throw exceptions into a version which suppressed them, as well as returning an Either type as indication of the nature of the outcome - crucially without changing the structure of the original function itself, except for modifying its return value to that of Either<IFailure, Result>.

This is useful because it allows for quick refactoring and serves as both a means to identify unsafe code (such as code that needs to be refactored to supress exceptions and return Monads) and at the same time partially remediates them.

In the process this then necessitates the need to change the calling code to now deal with Monad results reliably and predictable by the caller, without needing to refactor the called function yet. This enables accomplishing an iterative strategy of converting all functions to return Monads (Either<IFailure, Result>) in a expedient fashion without needing to go further (at that stage) to refactor each function internally, for example, to use pipelining until later in the refactoring process. This allows the architecture of the code to change quickly and incrementally without the code necessarily breaking because the functions themselves internally are changed also.

To do this, a family of routines was created by implementing a HOF that could be returned from any function, and which encapsulates/envelopes the entire function contents within it. It leverages the Try<A> Monad behind the scenes to result in an Either<L,R> return type.

Returning this function, and by composing in the original function as a input parameter, will encapsulate exceptions and return either the result of the function or an exception using an Either<L,R> return type:

```
// File: Statics.cs
// Info: This function will turn any exceptions into Either<IFailure, T>
public static Either<IFailure, T> TryThis<T>(this Func<T> action)
    => new Try<T>(() => action())
        .Match(
            unit => unit == NULL
                ? new NotTypeExceptionFailure(typeof(T))
                : unit.ToEither(),
            exception => new ExternalLibraryFailure(exception));
```

A convenience function, *Ensure()* which calls *TryThis()* behinds the scenes is used throughout the codebase:

```
// File: Level.cs
// Info: The function is unchanged except for returning calling Ensure()
public Either<IFailure, Unit> PlaySong() => Ensure(() => // <-- here
{
    MediaPlayer.IsRepeating = true;
```

```
    MediaPlayer.Play(_levelMusic);
});
```

This has the pleasing result that the unchanged function body looks the same (as mentioned previously, i.e. the contents do not change) with the exception of the additional *Ensure()* statement which is in effect a wrapping function call, passing in the original function inline and executing it within a Try monad.

Related to the incremental need to refactor code to be used in declarative pipelines which rely on Monad returning functions, is a similar incremental approach that allows newly refactored functions (who's internals as well as function signature have changed) which return Monads signatures to present their original return types by using an extension method for Either and Option monad types, *ThrowIfFailed()*.

This works somewhat in reverse: Refactor a function fully to return eithers and internally perhaps implement pipelining, but can now delay the refactoring of the calling code (which would need to be refactored to deal with the new return type of the function) to keep the existing code operational.

Again, this helps by going through the existing codebase in a slow, measured approach which helps to prevent large breakages which aren't easily testable if otherwise small changes caused larger propagating changes that would need to be refactored at the same time.

ThrowIfFailed() or *ThrowIfNone()* provides a gradual stop-gap measure to allow existing imperative code to continue to use refactored functions that have had their signatures changed to return monads. This allows underlying functions to be refactored this way without the referring calling code having to reflect using these new signatures (typically returning Monads now).

This means existing upstream imperative code can append ThrowIfFailed() or ThrowIfNone() to newly refactored function to simulate the old behaviour but with the caveat that this is a temporary measure which will allow old calling code to work but which will throw exceptions if the underlying monad is in an invalid or failed state.

This is achieved by inspecting the contents of the Monad and extracting the value that the original code is expecting, if however there is a failure, then throw an exception.

These occurrences are then progressively removed as the game is refactored, and play-tested. Remove these extension methods to fully utilize and integrate with the return type of monad, for example by refactoring the older calling imperative code to make use of the Monad return types as part of its own declarative pipeline:

```
// File: Level.cs
List<Room> MakeLevelRooms()
  => MakeRooms(removeRandSides: Diagnostics.RandomSides)
      .ThrowIfFailed(); // Returns a List<Room> instead of Either<>
      // Removing it will yield a Either<IFailure, List<Room>> return
      // type and the calling code will need to deal both
      // its Either<L or R> states (which is preferable and creates
      // more robust code)
```

Here, MakeRooms() returns Either<L,R> but existing imperative code still expects it to returns (R)ight type, ie a List<Room> not a Either as its implementation does.

5.1.0.2 Treating void as a type

A benefit of returning an optional type instead of void is you are conceding that a function could fail, a good assumption to make when considering robustness and this specifies that possible failure must be explicitly dealt with by the caller.

Additionally, returning Monads from functions allows the function to contribute within a larger logic pipeline, which can then be further defined using declarative syntax for computation of that logic.

During the refactoring process a large effort was made to convert void returning functions to Either<IFailure, Unit> as this provides more information about what happened in the function and the nature of the failure when it occurs.

The Unit type, used as a "right" hand side type of the Either<L,R> can be used to represent Nothing or void but unlike void, its strongly typed and can be used as a result, for example to represents the execution of an operation with lack of a failure (represented as Unit), instead of using void which provides no indication:

```
// File: Mazer.cs
 private Either<IFailure, Unit> InitializeUi()
    => Ensure(() => UserInterface.Initialize(Content, BuiltinThemes.editor))
        .Bind(unit => CreatePanels())
        .Bind(unit => SetupMainMenuPanel())
        .Bind(unit => SetupInstructionsPanel())
        .Bind(unit => SetupGameOverMenu(_gameOverPanel))
        .Bind(unit => AddPanelsToUi());
```

Here, operations that should not fail but do not return values can return Unit instead to indicate that there was no failure in the pipeline for that step and additionally as is the norm, if there was a failure, this is captured in the left type of the Either.

Unit is a type in Language-Ext that represents nothing. In MonoMazer, this type is aliased as a type of *Nothing*.

Alternatively using Option can also be used to represent the lack of a failure, represented by its Some() state or None to represent failure. This however does not contain information as to why its in the None state, i.e. why it failed. This can be easily remediated by using Either<TFailure, TSuccess> instead.

5.1.0.3 Debugging

As MonoMazer is re-designed to avoid throwing exceptions, it can be difficult to identify the source of the failure when it occurs as you only are aware of the failure at the point in which the result/failure is interpreted, not the point in which the failure occurred. This can be mitigated by inspecting the failure and tracing the failure to the point in which it is defined and used in the code.

This can be problematic and counterproductive and is a distinct disadvantage of using pipelining, which makes it difficult to debug at times.

However, when failures are triggered by exceptions and when all exceptions types are turned on under the debugger the initial exception can be isolated when it is first raised. Equally having distinct and meaningful *IFailure* types that represent distinct failures help to isolate there failure occurred, especially if the failure is unique and thus is uniquely located in the part of the code that represents where that failure must occur.

For example, a failure of type *InvalidCollision* can be isolated to the collision code that uses that type in returning a failure in the collision detection code. This is similar to having meaningful exception types in traditional imperative programming circles.

5.1.0.4 Optional Types

The question of when to use Option<T> over Either<Failure, T> is important. Option<T> for example, provides no reason why you have None, while Either<L,R> can encode this into the LHS type to indicate why. It is best suited for handling NULLs.

At best, you typically can convert an Option<T> to an Either<L,R> when you know what the reason is, so can then specify the nature of the context as an Either failure type depending on the program context:

```
// File: GameWorld.cs
private Either<IFailure, Unit> PlayerOnOnCollision(Option<GameObject> thePlayer,
    Option<GameObject> otherGameObject)
    => from player in thePlayer
             .ToEither(NotFound.Create("Player not found"))
        from gameObject in otherGameObject
            .ToEither(NotFound.Create("Other not found"))
        from isNpc in Must(gameObject, () => gameObject.Type == GameObjectType.Npc,
            "Must be NPC")
        from npcComponent in gameObject.FindComponentByType(Component.ComponentType.NpcType)
            // Convert Option<T> to Either<L,R>:
            .ToEither(NotFound.Create($"Could not find component of type
            {Component.ComponentType.NpcType} on other object"))
        from npcType in TryCastToT<Npc.NpcTypes>(npcComponent.Value)
```

```
        from collisionResult in ActOnTypeCollision(npcType, player, gameObject)
        select collisionResult;
```

This provides context to an Option using .ToEither() to represent a None condition that represents a failure, using an associated IFailure type that conveys the reason for the failure.

Note: In this example, LINQ's query syntax is used in place of its fluent syntax which implicitly calls Bind() in the statement on the right of the 'in' statement. This syntax is useful when extracted transformation values are referred to later within the expression, where otherwise using the fluent LINQ syntax's explicit bind() function only provides scope to the current statement only. This is explained in the tutorial with reference to how Select() is implemented for Monad types.

5.1.0.5 Pipelining

The benefit of returning Monads is that you can pipeline the functions, which benefits from additionally the inclusion of declarative layout of logic (using LINQ) and the use of Monad short-circuiting behaviour, not to mention the expression of functions instead of imperative statements.

However, returning Monads and providing transformation functions on monad data does not prevent exceptions from occurring themselves within those transformations. This can be a side-effect inducing problem.

The provided transformation function needs to be free from exceptions, preferably pure or otherwise needs a mechanism to not throw them and guarantee that no code will throw exceptions. This is what makes using a mechanism as described earlier such as *Ensure()* or in these cases *EnsuringMap()* or *EnsuringBind*() custom extensions useful as they protect these transformations if they are not inherently pure or are in the process of being made pure.

Removing logic from or adding it to a pipeline is simply a matter of removing, replacing, or adding functions. This adds flexibility and maintainability benefits. For example, each constituent function in theory reduces the risk of error propagation, and isolates the functional changes to specific functions within the overall larger composition.

This also makes testing complex logic easier as more of the constituent parts of the logic can be individually tested as individual functions. This is explored separately later.

5.1.0.6 Pipelining and decomposition

This is key aspect of pipelining a large procedure to break the procedure into smaller function expressions.

This is beneficial because it isolates problems to specific expression segments, segments which can then be switched out very easily without affecting the rest of the code and can be refactored and tested independently.

This means each segment can now be made more robust using Either<L, R> return types and when combined with ensuring capabilities will always finish.

Moreover, these isolatable functions can now be the subject of unit tests, without the inherent problems of needing to mock out deep dependencies, as they don't have any.

5.1.0.7 Problems and limitation of Pipelining

Pipelines are best at representing computations that either succeed or fail entirely.

It is clear however that trying to pipeline everything is not always easy or useful. Some functions that inherently do not return anything aren't easy candidates for pipelining but can benefit from returning Nothing and indicating success or failure through retuning optional types.

Functions whose primary purpose is to change underlying state (in a hybrid imperative capacity), and therefore do not return any data appear not to benefit from attempting to pipeline it. This is the case with some classes that are designed to modify their own state and are not good candidates for pipelining.

Furthermore, if you want to avoid the typical "all-or-nothing" behaviour of pipelines (it either succeeds for fails) and inspect results "mid-flow" within the pipeline, checking a value for conditions and then resuming the pipeline, depending on the result is possible. This does however requires adding more checks using Match() in the pipeline which can appear to complicate the pipeline, but might be essential to replicate the required logic:

```
// File: Level.cs
public static Either<IFailure, Player> MakePlayer(Room playerRoom,
LevelDetails levelFile, IGameContentManager contentManager) => EnsureWithReturn(()
    =>
  (from assetFile in CreateAssetFile(levelFile)
  from texture in contentManager.TryLoad<Texture2D>(assetFile).ToOption()
  from playerAnimation in CreatePlayerAnimation(assetFile, texture, levelFile)
  from player in CreatePlayer(playerRoom, playerAnimation, levelFile)
  from InitializedPlayer in InitializePlayer(levelFile, player)
  from playerHealth in GetPlayerHealth(player)
      // Match allows for inspecting the pipeline and altering it
      .Match(Some: (comp)=>comp, // found an existing component
             None: ()=> AddPlayerHealthComponent(player)) // not found, add it
  from playerPoints in GetPlayerPoints(player)
      .Match(Some: (comp)=>comp,  // if in Some State, do ..
             None: ()=> AddPlayerPointsComponent(player)) // if in None state do..
  select player).ThrowIfNone());
```

In the above code, inspecting the value of Monads requires provision/code to cater for both states of the Monad, ie Some and None states. This makes the code far more robust than assuming that only the valid or happy path/state is likely.

It also means using side effect inducing exception handling to cater for the invalid state is now unnecessary.

5.1.0.8 Exposing new issues

As soon as *Ensure()* with *Either<L,R>* return types were used as return types over imperative code, exceptions that were not being cared for (being ignored/swallowed) were now causing code to fail as any exception was being converted to failure instead of being ignored. This forced the caller and dependant functions to deal with the problem, making the problem definable and the code more robust. This could however be an potential pain-point when refactoring initially.

For example, an issue not accounted in the original version of MonoMazer (double escape in game) was fixed in Commit #73 (See commit history in the appendix)

5.1.0.9 Declarative Syntax

With the reduction of control statements such as if, for, while etc, in my opinion the code does appear cleaner, less complicated, more specific, descriptive and more concise when using the LINQ query syntax and combined with pipelining Monads.

This, I attribute mostly to the separation of concerns into smaller pieces, as described by (Buonanno, 2017), that are labelled by descriptive, declarative function names that are expressed in a pipeline which will either succeed or fail at particular stages:

```
// File: GameWorld.cs
// Info: Functional way: describing what needs to be done instead of how
private Either<IFailure, Unit> RemoveGameObject(string id, Level level)
  =>  from gameObject in GetGameObject(GameObjects, id)
      from notifyObjectAddedOrRemoved in
        NotifyObjectAddedOrRemoved(gameObject, GameObjects, OnGameObjectAddedOrRemoved)
      from isLevelPickup in IsLevelPickup(gameObject, level)
         .IfSome(unit => RemoveIfLevelPickup(gameObject, level)).ToEither()
      from removePickup in RemoveIfLevelPickup(gameObject, level)
      from isLevelCleared in IsLevelCleared(level)
         .IfSome(unit => NotifyIfLevelCleared(OnLevelCleared, level)).ToEither()
      from deactivateObjects in DeactivateGameObject(gameObject,GameObjects)
      select Nothing;
```

Readability of declarative logic is dependent on providing descriptive naming of the composed functions and providing

informative and descriptive names of functions can help when finding them within *Bind()* statements and can make interpreting those statements more obvious. In contrast, poorly named functions can impair the readability of declaratively specified logic.

5.1.0.10 Performance

Some performance 'jitter' in the game was observed during the refactoring process.

There was a new periodic lag that caused a slow-motion effect for short segments of time, which then stopped and allowed the game continues running normally.

At first this was thought to be caused by garbage collection due to its periodic nature however it appears that the problem was due to the introduction of the *.ToOption<T>* and *EnsureWithReturn()* used in a light loop in the collision code.

EnsureWithReturn<T> was used to protect against exceptions within the collision code.

It is not immediately clear what the remediation is here other than not using EnsureWithReturn(), meaning that there might be a performance implication in tight loops where the Try<> Monad is called repeatedly for example and more investigation and research would need to be done to quantify this, which is beyond the scope of this project.

The following shows how this was rectified by providing **Fast* alternatives:

```
// File: GameWorldStatics.cs
public static Option<int> ToRoomColumn(GameObject gameObject1, int roomWidth)
 => EnsureWithReturn(()
    => ToRoomColumnFast(gameObject1, roomWidth)).ToOption();

public static Option<int> ToRoomRow(GameObject o1, int roomHeight) => EnsureWithReturn(()
    => ToRoomRowFast(o1, roomHeight)).ToOption();

public static int ToRoomColumnFast(GameObject gameObject1, int roomWidth)
    => roomWidth == 0 ? 0 : (int)Math.Ceiling((float)gameObject1.X / roomWidth);

public static int ToRoomRowFast(GameObject o1, int roomHeight)
    => roomHeight == 0 ? 0 : (int)Math.Ceiling((float)o1.Y / roomHeight);
```

In the above, a possible performance issue with ToOption<T> required a unsafe faster method.

5.1.0.11 Testing logic

Another important and useful aspect of breaking down monolithic functions into pipeline segments is that it becomes easier to write tests for the separate functions that made up the composition of logic.

To test this, pure functions were put into static types that represented the general area that they were used in and then tests were written to cover only these static types and leave the remaining impure functions as uncovered.

As a result over 50% of the game was revelaved to be covered by tests, which indicates that pure functions can represent the majority of the games logic in Mazer. This is shown below highlighting that the cummulative code coverage of only pure functions lead to most of the game being immediately testable and ultimately more correct than before.

Name	Covered	Uncovered	Coverable	Total	Line coverage	Covered	Total	Branch coverage
MazerPlatformer	**1261**	**1406**	**2667**	**5229**	**47.2%**	**404**	**702**	**57.5%**
MazerPlatformer.ShortCircuitFailure	6	0	6	13	100%	0	0	
MazerPlatformer.NotTypeExceptionFailure	6	0	6	16	100%	0	0	
MazerPlatformer.NotFound	6	0	6	13	100%	0	0	
MazerPlatformer.InvalidDataFailure	6	0	6	13	100%	0	0	
MazerPlatformer.ExternalLibraryFailure	13	0	13	24	100%	0	0	
MazerPlatformer.ExceptionFailure	2	0	2	20	100%	0	0	
MazerPlatformer.Diagnostics	11	0	11	21	100%	0	0	
MazerPlatformer.ConditionNotSatisfiedFailure	6	0	6	12	100%	0	0	
MazerPlatformer.AssetLoadFailure	6	0	6	13	100%	0	0	
MazerPlatformer.MovingStateStatics	21	0	21	33	100%	16	20	80%
MazerPlatformer.AggregatePipelineFailure	17	0	17	29	100%	4	4	100%
MazerPlatformer.LevelStatics	110	1	111	163	99%	44	62	70.9%
MazerPlatformer.CharacterBuilder	68	2	70	109	97.1%	32	32	100%
MazerPlatformer.MazerStatics	103	4	107	179	96.2%	17	18	94.4%
MazerPlatformer.RoomStatics	111	8	119	213	93.2%	52	58	89.6%
MazerPlatformer.Component	11	1	12	42	91.6%	0	0	
MazerPlatformer.Statics	217	20	237	544	91.5%	122	134	91%
MazerPlatformer.TransformExceptionFailure	6	1	7	20	85.7%	0	0	
MazerPlatformer.InvalidCastFailure	6	1	7	14	85.7%	0	0	
MazerPlatformer.RectDetails	27	5	32	55	84.3%	7	12	58.3%
MazerPlatformer.UnexpectedFailureException	5	1	6	15	83.3%	0	0	
MazerPlatformer.Level	209	56	265	415	78.8%	38	62	61.2%

Figure 5.1: Game test coverage of static functions only yields more than half of game

This is important because while it may appear easy to visually detect problems in a game (as very small things become visually obvious), subtle issues that cannot be detected by gameplay testing or integration testing can be easily missed, which otherwise can now be easily identified with unit tests (and proved to be incorrect) and this can lead to game code being more correct, as more tests are put in place to enforce this.

This is significant as it means isolatable changes in the game's pure functions do not require the extensive wait time to time to build the entire game before commencing testing, such as through play-testing cycle.

This in itself could considerably improve the issue of productivity in game development as highlighted by Sweeney (Sweeney, 2006).

5.1.0.12 Immutability

An approach to immutability within an imperative language is copying object data by way of serialization and deserialization to effect a "deep" copy. This was the approach taken in MonoMazer to implement immutability due to its simplicity.

Other approaches could have been used, such as using immutable data types, such as those provided by Microsoft Immutable collections for example.

The newer version of C# will support record types which are inherently immutable and as such would potentially benefit from being returned from functions and passed as parameters. Undertaking this however would have required too much a change in design and effort as the planned refactoring approach was already well underway.

Also, it is possible to re-design code for example to create and return types that only expose public getter methods and thus ensure the instance cannot be changed subsequently. In MonoMazer however this was not implemented since so many datatypes were already being used in MonoMazer that were mutable and the effort to convert them all to suitable immutable types was time restrictive.

Using serialization does add the onus on the developer however to ensure objects are susceptible to serialization and this can be a problem. For example, using JSON serialization can require that a constructor signature exists that can initialize the class using the member of the class as arguments. These types of changes to existing classes can be easily made, however.

This problem can be lessened by using properties in classes, which does not necessitate including them as arguments in a constructor, which can be rather large, as the serializer will use those to set members accordingly during deserialization for example. As this is often done, and easily to do, this does not present much of an imposition, however in 3rd party code this is not always possible.

It is recognised that performance in copying objects is a concern and whatever approach to immutability should be evaluated according to its use in the codebase.

One however cannot ruthlessly copy every object using serialization to enforce immutability. You only have control over the objects that you can create a copy constructor for, this means some objects that are part of 3rd party libraries could be not serialized, however this was not the case in MonoMazer, as all 3rd party code was considered to be unsafe, and relegated to the fringes as per the Hub-Spoke distributed model presented earlier.

It became clear that perhaps one could cache copy operations if they have been copied before (effectively simulating memorization) saving CPU. This would probably increase memory usage if a suitable caching strategy was not designed wisely and could be a burden on the developer. This would also require that all objects may need to be uniquely identifiable, and comparable such that object copies already created would not be re-created.

This was not pursued in the implementation of *MonoMazer*, because it would require implementing *IComparable<>* for every object to implement this caching strategy.

The overall pattern used in Mazer, is to pass the object that you will be using and potentially modifying as an input argument, copy it via serialization (and thus remove dependency on it by external sources) and then return the copy after modifying it.

Another strategy is to return values types such as structs. Primitive types are also immutable by default such as int, string etc.

When a function can be set as static, it indicates that no shared state can be accessed (except for static data, however this should not be done for the sake of jeopardising purity), and as described above, as the function is designed to copy parameters and thus remove all links to the outside dependencies that those parameters might have, this effectively makes the function pure.

```
[PureFunction]
public static Either<IFailure, Room> AddWallCharacteristic(
  Room, Room.Side side, SideCharacteristic characteristic)
{
    // copy characteristic, change it, and then return the copy
    return
        from sideCharacteristic in characteristic.Copy()
        from roomCopy in room.Copy()
        // We are now operating on copies and thus are pure
        from result in AddSideCharacteristic(roomCopy, side, sideCharacteristic)
        select result;
    ....
}
```

The above code for AddWallCharacteristic() uses serialization copying to enforce immutability.

It was found that copying an entire *GameWorld* object for example, is not feasibly possible due to the magnitude of the aggregate components that are not under your control. Focus needs to be instead be on methods that use objects that you have control over and can copy. This leads to requiring a more strategic, hybrid approach to isolating code that could be made immutable and would thus benefit from it such as smaller individual composite functions to make up larger logic circuitry.

In retrospect, using immutable types such as those identified earlier might have been easier and more performant, and this form of copying objects (serialization) is likely to be slow and unacceptable for games in the real world. This however can be improved by using other techniques as suggested above.

It is also not always possible to create pure functions especially in MonoGame where framework callbacks that are provided by the game engine framework, don't have arguments you can use to shuffle data in/out of those functions. This is true of top level functions provided by MonoGame such as *Update(), Draw()* etc. however the implementation

of these can be considered at the very edge of the hub-spoke model and they can be converted to pipelines which call into the core:

```
// File: Mazer.cs
// The expressions call into core functions from the fringes
protected override void LoadContent()
    => GameContentManager
        .Bind(contentManager => contentManager
                                .TryLoad<SpriteFont>("Sprites/gameFont"))
        .Bind(font => SetGameFont(font))
        .Bind(contentManager => GameContentManager)
        .Bind(contentManager => contentManager
                                .TryLoad<Song>("Music/bgm_menu"))
        .Bind(music => SetMenuMusic(music))
        .Bind(unit => LoadGameWorldContent(_gameWorld, _currentLevel,
            _playerHealth, _playerPoints))
        .ThrowIfFailed();
```

In normal applications, as on the fringes of the hub-spoke distribution model, issues here are evident of catastrophic errors should be explicitly catered for, being intrinsically unsafe. Here, at a terminal function on the fringe/spoke, the result of the pipeline is interpreted for failure and exception is thrown if one is present.

5.2 Objective 2: Understanding Language-Ext

Understanding Language-Ext can be challenging not least because it requires an understanding of functional programming but because its syntax is not immediately intuitive and as such, to break-through the theoretical wall of understanding, requires a fairly steep learning curve. Furthermore, its not immediately obvious where and when its language constructs should be used and how they are used.

To remedy this and to aid the primary research objective which is to expose and evaluate the utility of incorporating functional programming and its associated techniques into a game, a step-by-step narrative tutorial is presented through a series of runnable code projects that build up on the basic Language-Ext constructs and explain why and how they are used.

The first concept to understand is what Monads are and how they are used, particularly within practical use-cases. To help achieve this, the general idea of Monads are introduced first by introducing a generalization of a Monad as a box or a container. This analogy is helpful when considering the specific Monads used in Language-Ext and that have been used during the refactoring effort.

Defining the specific Monads types that are used in MonoMazer follow, namely the Either<L,R> and Option types which are then followed by the operations that can be used on these types, which are each demonstrated separately. A key aspect of the tutorials is the progressive continuation of previous concepts and how narration throughout the code samples in the form of comments serve to put into context the operations and constructs at hand.

A unique aspect of Language-Ext, is its integration and use of specific C# language features such as its extensive use of extension methods and novel integration with LINQ. These are explained and help to describe how it is used to drive declarative representation of data transformations using Monads.

As data transformations using Monads are important, primarily in representing logic as expressions instead of statements, this technique is used throughout both the tutorial and MonoMazer. Such transformations are shown to occur within “transformation functions” that act on the data contained with the monad.

There are nuances in performing transformations through the monadic functions, Map() and Bind() and these are illustrated through iterative examples, which ultimately culminate in demonstrating the important concept of pipelining.

Declarative style is exemplified through decomposing monolithic logic into a series of discrete transformations, each describing its own transformation using informative descriptive names. The useful characteristic of pipelining is the ability to incorporate automatic validation and short-circuiting which is shown can be used to cancel remaining transformations, depending on the content of monads.

Examples of pure functions are illustrated, which when composed into pipelines as transformations, enable side-effects free transformations which help make reasoning about the effect of transformations helpfully deterministic.

The following are the topics within the tutorial, with reference to each runnable project written in brackets, where each is numbered within the Visual Studio solution to allow for topic correlation.

The relevance to the refactoring effort undergone in this project is provided in the 2nd column.

Table 5.1: Topics documented in "Understanding language-Ext" and of relevance to MonoMazer refactoring process

Topic	**Relevance**
Monad Basics	Describes a Monad in terms of Language-Ext implementation
Either<Left,Right> Basics	Fundamentals of how Eithers work
Operations on Lists of Either<left, Right>	Examples of how to use Either<L, R> in various scenarios
Option Basics	Example so how to use in various Option<T> scenarios
Introduction to the Box type (a Monad)	Prototypical representation of a Monad
Transforming the contents of a Monad.	Operating on a monad to transform data within it
Using the Select function on a Monad	Understanding the Select() function
Understanding Map and Bind	Theoretical underpinnings behind monadic functions
Pipelining transformation workloads (3)	Shows how Bind is used to create a pipeline of function calls
Monadic functions: Complete example (4)	Practical example using Box Monad
Using LINQ Fluent and Expression syntax (6)	LINQ Fluent and Expression syntax
When to use *Map()* or *Bind()* (7)	Shows you when to use Map() and Bind()
Imperative to Declarative style (8)	Imperative style coding to Declarative style coding
Monad validation (9)	Shows you that pipelines include automatic validation
Returning Monads (10)	Transformation function always return a Monad
Built-in validation and short-circuiting (11)	Monad built in validation and short-circuiting
composition of functions (12)	Composition of functions
Pure Functions (13)	No side effects i.e. mathematically correct
Introducing the Either monad (14)	Shows the basics of Either<L, R> using Bind()
Operations on Either<L, R>	Shows use of map and Bind on Either types
Operating on Lists of Either<left, Right>	Further operations that can be used
Introduction to Option<T> (28)	Introduction to Option
Basic use-case of Option<T> (29)	Basic use-case of Option
Using Option<T> in functions	Implications for returning Optional types
using IfSome() and IfNone()	Transition constructs from imperative to functional programming
Pipelining with Options<T> (32)	Pipelining functions
Using ToEither<>	Converting data to a Monad
Using BiMap() (34)	Using BiMap() - see tutorial 19
The Try<T> Monad - Supressing Exceptions	This is used to ensure that functions return reliably
ThrowIfFailed()	Introducing ThrowIfFailed and Either<IAmFailure, Option<T>>
FailureToNone()	Transition constructs from IP To FP
Memoization	A form of caching
Apply events over time to change an object	Applying change in state over time in a functional manner
Smart constructors and Immutable datatypes	Constructing objects safely

This tutorial then serves as an important reference point into the operation and application of the specific functional programming paradigm and techniques that are incorporated into MonoMazer, but also to accessibly understand what Language-Ext is fundamentally designed to achieve, and how.

5.3 Objective 3: Code Complexity

5.3.1 Overall VSCA cumulative results

The results, particularly LOC and ELOC indicate that there has been a dramatic increase in the amount of new code introduced in the refactored version to maintain like-for-like behavioural parity in the game across the two

codebases.

Table 5.2: Percentage difference in final overall cumulative metrics as reported by VSCA Before vs 'After'

Overall cumulative metrics	Before	After	Difference	Change in %
Maintainability Index	91	91	0	0 %
Cyclomatic Complexity	578	877	+ 299	+ 51.73 %
Class Coupling	133	218	+ 85	+ 63.91 %
Lines Of Source Code	2762	4345	+ 1583	+ 57.31 %
Lines of Executable Source Code	911	1919	+ 1008	+ 110.65 %
Functions	312	1124	+ 812	+ 260 %
Classes	45	88	+ 43	+ 96 %

This appears to indicate that FP has not achieved an overall improvement towards a level of code conciseness that might have been expected overall due to, for example the use of inline compositional function expressions in place of imperative line statements, furthermore that the techniques and paradigms have a larger footprint than merely replacing like-for-like imperative statements for example.

This is evident also in the increase in the number of functions and count of new types.

The number of functions has increased significantly, by well over triple as many as in original codebase.

An increase in functions is reasonably expected in the functional codebase considering the nature of FP, ie. more functions are created to represent and describe more aspects of functionality within functions. However the extent of the increase is surprising.

This however does pose an implication for evaluating Cyclomatic Complexity in FP codebases as FP is pre-disposed to having more functions and therefore contributing to a higher overall cumulative Cyclomatic complexity than imperative codebases which have a less functions would.

Generally the Maintainability Index metric shows that it remained relatively unchanged across the codebases, however class coupling has increased by over half. This suggests that incorporating the functional programming approach has contributed to classes to become more dependant on each other, which increases the propagation risk of errors moving throughout the codebase. This is therefore, along with the increase amount of code is a quality concern.

The overall cumulative cyclomatic complexity has also increased by over half. This suggests that the functional codebase has not made the code less complex, in terms of its impact on reducing the number of linear pathways through the code.

This is somewhat surprising considering most refactored functions appear to have more linear i.e. have less imperative branching statements for example.

A possible reason for this is likely to be that the increase in 'hidden' executable code has contributed to higher levels of conditional statements not in the main function body but in the composed function bodies that make it up.

In this way, functional programming does not appear to have improved the code complexity as in this respect.

To investigate the impact that the increase in functions has contributed to these cumulative results and to explore the idea this might impose an unfair or disproportionate bias, particularly as far a Cyclomatic complexity is concerned, we also analysed the effect on the 'average' function.

We also consider the 'average' type in an exploratory way. Types are compositions of multiple functions that make up the type itself. Type is used interchangeably with 'class' in OOP.

Below the average existing function, new function and overall function (considering both existing and new functions) are compared.

Table 5.3: Percentage difference in the 'average' function

Average function metric	After vs Before (Existing)	New vs Before	Now vs Before (overall)
Maintainability Index	+ 0.12 %	- 1.77 %	- 1.57 %
Cyclomatic Complexity	- 21.33 %	+ 28.11 %	+ 24.95 %
Class Coupling	+ 24.44 %	+ 101.47 %	+ 90.04 %
Lines of Code	- 9.76 %	+ 0.89 %	+ 0.79 %
Lines of Executable Code	- 14.73 %	- 0.16 %	- 0.14 %

The FP refactoring is not considered to have materially influenced the Maintainability Index significantly, with all changes below 2%.
These also corroborate the overall cumulative results reported by VSCA.

The Cyclomatic complexity however has shown a significant improvement in existing functions, by over 20% which suggests that improvement is not totally missing as the cumulative results suggest. They also prove that much of the increase overall is from the new functions.

While still an increase in cyclomatic complexity overall, it might be arguably more representative to discount the cumulative increase of 51.73% in favour of this overall average function increase of 24.95% given the observation made regarding the potential disparity or bias in cumulative complexity to the functional codebase that has more functions.

The Class Coupling across all averages appears to agree with the cumulative result - that the FP code is significantly more coupled, and as such risks potential errors propagating to adjacent code more easily. A likely reason for this is the reliance that almost all changes have on the central static types that house the pure functions. It might be more useful to keep static functions within the enclosing type instead of move them to a new type. The primary reason why this was done was to assess the potential impact that test coverage of only these static types influence the coverage of the entire game, which is explored later.

The conciseness of the functions on average only increased by less than one percent. While this is small, it does however show that the cumulative results, i.e. more than 50% and 110% aren't truly as representative of the average function and appear to exaggerate the increase and appears influenced heavily by the large increase in functions overall. With this concession however, it still does not produce an improvement in conciseness in the functional codebase.

The overall cumulative results as reported by VSCA is also reproduced for comparative purposes.

Table 5.4: Percentage difference in final overall cumulative metrics as reported by VSCA compared to change in average function.

Metric	Cumulative Change in % A	verage change in %
Maintainability Index	0 %	- 1.57 %
Cyclomatic Complexity	+ 51.73 %	+ 24.95%
Class Coupling	+ 63.91 %	+ 90.04 %
Lines Of Source Code	+ 57.31 %	+ 0.79 %
Lines of Executable Source Code	+ 110.65 %	- 0.14 %

This suggests that the complexity of the average function has increased by about a quarter (25%) and it is over 90% more coupled than before, but its conciseness in general has not changed significantly (difference of less than 1%).

In assessing the average type metrics, new types (which are more prevalent) appear to invert the effect that existing types see.

Apart from Maintainability Index, which sees a approximately 3% increase, the remaining average type metrics show a significant reduction overall (Now vs Before), showing that the average type is less complex, less coupled and contains less lines of code.

This is likely due to the increase in functions in each type which increases the divisors and thus reduces the effects more readily as opposed to evaluating individual functions. In this way we consider the average type to be less

indicative of the metrics compared to the average function, however they are still insightful.

Table 5.5: Percentage difference in the average type

Average type metric	After vs Before (Existing)	New vs Before	Now vs Before (overall)
Maintainability Index	- 0.11 %	+ 4.94 %	+ 2.91 %
Cyclomatic Complexity	- 6.43 %	- 57.04 %	- 33.6 %
Class Coupling	+ 58.82 %	- 40.98 %	- 24.14 %
Lines of Code	+ 7.91 %	- 67.62 %	- 39.83 %
Lines of Executable Code	+ 54.8 %	- 60.39 %	- 35.57 %

This appears to suggest that the average type generally has reduced in complexity, and that unlike the cumulative results reported by VSCA, may provide a better alternative cumulative view of types within codebases which have more functions, particularly in the case of Cyclomatic Complexity.

When considering all types in constructing the average type as a general representation of the metrics, it also suggests that the functional types generally has improved the overall situation, in a way the VSCA cumulative results are not able to capture:

The overall results as reported by VSCA is also reproduced for comparative purposes.

Table 5.6: Percentage difference in final overall cumulative metrics as reported by VSCA compared to change in average type.

Metric	Cumulative Change in %	Average change in %
Maintainability Index	0 %	+ 2.91 %
Cyclomatic Complexity	+ 51.73 %	- 33.6 %
Class Coupling	+ 63.91 %	- 24.14 %
Lines Of Source Code	+ 57.31 %	- 39.83 %
Lines of Executable Source Code	+ 110.65 %	- 35.57 %

In summary, this suggests that assessing the average impact across functions and types should be considered along with the cumulative results and factored into assessing the meaning of the results, particularly when evaluating a functional codebase.

It also suggests that the cumulative results are not as fully representative of code complexity in FP codebases as would be the case in imperative codebases.

Significant for this project however, is that the Cyclomatic complexity, as a result of incorporating the functional programming paradigms and techniques, has increased by at least 24% and more if you consider the cumulative results.

5.3.2 Outliers

In addition to dampening the cumulative bias mentioned previously, measurements relative to the average function have helped expose nuances that the cumulative results could not expose due to their aggregate nature. These nuances have been shown in comparisons of the average function and type after vs before, new vs before and now vs before. Averages also serve as a way to generalize and determine a representative candidate that best represents the dataset, however it too is not without problems.

By analysing the average function or type measurements, it is clear that outliers exist and that they do influence this generalization of the typical function. This is particularly noticeable in the static types which account for almost all the pure functions in the game.

To mitigate the effect of these outliers, the median function provides a outlier-agnostic perspective.

Table 5.7: Median function differences

Metric	After vs Before (Existing)	New vs Before	Now vs Before (overall)
Maintainability Index	0 %	- 4.3 %	- 3.2 %
Cyclomatic Complexity	0 %	0 %	0 %
Class Coupling	+ 50 %	+ 200 %	+ 300 %
Lines of Code(LOC)	+ 50 %	+ 100 %	+ 100 %
Lines of Executable Code (ELOC)	0 %	0 %	0 %

This shows that because of the refactoring effort the overall median function is shown to be 5 times more coupled (300%), contains half has much more lines of code as it did before, but overall it has not changed its cyclomatic complexity or executable lines of code throughout the refactoring process, as the average (including outliers) metrics suggested.

Table 5.8: Median type differences

Metric	After vs Before (Existing)	New vs Before	Now vs Before (overall)
Maintainability Index	+1.11 %	+ 5.56 %	+ 4.44 %
Cyclomatic Complexity	-10 %	- 20 %	- 20 %
Class Coupling	+ 87.5 %	+ 50 %	+ 25 %
Lines of Code	0 %	- 56.52 %	- 34.78 %
Lines of Executable Code	0 %	- 66.67 %	- 66.67 %

After introducing new functional paradigms and techniques, the median type is 20% less complex, 25% more coupled, has 34% less lines of code and approximately 67% less lines of executable code. This is inline with the effect that more functions would have in reducing the effect of some metrics like MMC.

What these median values suggests (as they do differ compared to averages for MCC for example) is that when considering the metrics by median function and the average function, it might be more prudent to consider the interval between the two as a more representative approximation of generalization than considering using either one in isolation. This can be achieved by considering a hybrid average-median or 'Hybrid Generalized' metric by taking the sum of the average and the median, and dividing by 2, effectively regularizing the differences between the two and considering them both as equally important.

While the results do not differ greatly from the average metrics presented earlier, it does shows a slight (or more generalized) variance which takes into consideration both the biases previously identified, and unifies the median and average metrics previously analysed:

Table 5.9: 'Hybrid Generalized' function differences (Average + Median/2)

Metric	Before	After (Overall)	% change
Maintainability Index	(87.38+98)/2 = 92.69	(86+90)/2 = 88	- 5.05
Cyclomatic Complexity	(1+1)/2 = 1	(1.27+1)/2 = 1.135	+ 13.5
Class Coupling	(2.43+1)/2 = 1.715	(4.62+3)/2 = 3.81	+ 122.16
Lines of Code	(4.99+1)/2 = 2.995	(5.03+2.0)/2 = 3.515	+ 17.36 %
Lines of Executable Code	(3.03+1)/2 = 2.015	(3.02+1)/2 = 2.01	+ 0.25 %

Functions thus are 'generally' 5% less maintainable, 13.5% more complex, 122.16% more coupled, have 17.36% more LOC and 0.25% more ELOC.

To put this into perspective, consider the reduction of Cyclomatic Complexity by 20% previously reported (in existing functions only, i.e. Before vs After), the hybrid generalization of this figure is approximately 10%, ie. 10% more conservative, and that 10% has thus been attributed to an "outlier effect", and so has been relatively discounted.

Table 5.10: 'Hybrid Generalized' type differences (Average + Median/2)

Metric	Before	After (Overall)	% change
Maintainability Index	(87.13+90)/2 = 88.565	(89.67+94)/2 = 91.835	+ 3.69%
Cyclomatic Complexity	(18.13+5)/2 = 11.565	(12.04+4)/2 = 8.02	- 30.65
Class Coupling	(11.9+4)/2 = 7.95	(9+3)/2 = 6	+ 24.53
Lines of Code	(90.57+23)/2 = 56.785	(54.49+15)/2 = 34.745	+ 38.81 %
Lines of Executable Code	(30.23+6)/2 = 18.115	(19.48+2)/2 = 10.74	+ 40.71 %

Consequently, types thus are 'generally' 3.69% more maintainable, 30.65% less complex, 24.53 more coupled and have 38.81% and 40.71% more LOC and ELOC respectively.

This method is not without its problems such as considering the effect of outliers and median values as equally weighted however is does provide an interesting assessment of how the cumulative results reported by VSCA could be interpreted when very high function counts are present in the codebase.

5.3.3 Effectiveness of code complexity metrics

Lastly it should be mentioned that cyclomatic complexity is not without its problems.

While the cyclomatic complexity on average of functions within MonoMazer were on average significantly less than 10 or 15 which characterise complex functions, its worth considering that research has shown that the complexity of independent high ranking Cyclomatic complex functions may not necessarily reflect the perceived complexity of their developers, and that in these instances "... human opinions do not correlate with MCC values of these functions". (Jbara, Matan and Feitelson, 2014)

In these cases, these functions "... are in fact well structured" and suggests that "... a high MCC is not necessarily an impediment to code comprehension, and support the notion that complexity cannot be fully captured using simple syntactic code metrics" (Jbara, Matan and Feitelson, 2014).

That study also reiterates a common objection of MCC, that it is heavily correlated to the number of lines of codes (LOC), and that "... this correlation has been demonstrated many times".

There has been a cumulatively large increase in the number of lines of code produced by refactoring MonoMazer, which would have cumulative increased the complexity as perceived by Cyclomatic complexity and this is represented by the high MCC as reported by VSCA.

Chapter 6

Evaluation, Reflection

A fundamental realization is that FP does not specifically reduce "complexity" by promoting less complex code per se, as measured through complexity metrics such as those evaluated in this research, but instead it can transform uncertainty in code and promote predictability and it is this is what makes FP easier to reason about and therefore less complex.

Its ability to prove that the effects of a function will not impact or change over time can help eliminate possibilities that otherwise would need to be determined (which is burdensome) when reasoning about the effect of a piece of code or trying to establish what could have or should happen when the code is executed in different scenarios. This can be seen, for example through eliminating runtime failures scenarios (exceptions and unexpected NULL) and side effects (immutability).

A large part of the thinking behind approaching FP for this project was pre-defined thinking that it might reduce the "complexity" of the game. However, through this research "complexity" has evolved to represent a more multi-faceted idea which should not alone be measured through code complexity, as other important benefits such as improving other cognitive complexity facets such as improving predictability and robustness qualities may not be influenced by Cyclomatic Complexity for example.

Another facet of "complexity" is the ability that FP has to encourage better description of what a logic does through function composition, which allows for more opportunity to describe logic, for example through its multiple declarative occurrences of function names that compose a larger portion of logic. This too is facilitated by it encouraging the design of expressions to describe what code is to do, instead of how it does it (which should reside behind the function names that express/describe the overall logical idea). While these were not specifically part of the initial research objectives, they have become apparent and have surfaced during the research and appear to be both interesting and relevant to assessing complexity generally.

This higher-level reading of the code, coupled with more descriptions of what the code does, arguably adds an extra dimension to complexity as a "cognitive" facet. These qualities have emerged as relevant and useful however are not measured in this research but which deserves future research.

Personally, the journey of coming to terms with how functional thinking can be used to adapt existing imperative code has been challenging but rewarding. This is specifically true when approaching adapting imperative structures to functional ones.

This has required determining, and then ensuring a like-for-like functional approaches exist to replace the imperative equivalent. In some cases this required constructs to be manufactured from scratch, such as the functional replacements for switch statements, for example.

This itself has been insightful in understanding that many aspects of IP are not actually needed (although we think they are because of our traditional imperative thinking). A few examples include, conditional constructs like switch and if statements, which can be entirely replaced by transforming data through monads such as Either and Option. As to whether this is more readable or aids comprehension is not measured here.

Categorising the functional paradigms and techniques used in refactoring MonoMazer into essential ideas that promote predictability or robustness for example has made it easier to think about functional programming as an

enabler of these core ideas and it is useful when strategizing which aspects of the work the refactoring is contributing towards.

Equally, overcoming the initial resistance to thinking in a functional mindset has, as already recognised in the existing literature, been as challenging as finding the right ways to represent FP constructs in the code.

The refactoring process was by far the most time-consuming and challenging activity, taking most of the allocated research time. It is with this insight that the approach was taken to refactor with an incremental approach which allows for co-habitancy of both existing imperative code structures alongside new functional ones.

Refactoring MonoMazer and embracing a functional programming style has been challenging, not least because it required learning and embracing a new and different mindset to approaching implementing logic, but also because MonoMazer was originally designed from the outset with purely imperative constructs in mind. This however has helped to convince me that moving toward a hybrid model incorporating FP paradigms and techniques is possible, not least of all in developing computer games.

The transition and application of FP from these imperative constructs to functional equivalents has been informative, particularly the pursuit of reducing side effects, not only through immutability but more so by eliminating unpredictable code such as those experienced due to runtime errors.

While robustness of the codebase has not been quantified, the ideals and concepts that are designed into FP appear to encourage a more robust approach to dealing with errors which is a useful aspect that merits future research.

When considering robustness specifically, the enforcement by design of making it mandatory to deal with the alternative scenarios, for example when inspecting the value of a optional type such a Either or an Option is imminently useful: A simple imperative 'if' statement alone, without a diligent programmer cannot guarantee that the alterative condition is catered for, and which is likely to be where an unexpected error would occur, as there is no code by default to deal with that risk.

Pipelining as a concept, has made it possible to represent an entire monolithic function or an idea as a series of expressions instead of a series of imperative statements which is a key feature in FP that distinguishes it from IP. In most cases this as eliminated the need to store intermediate values in variables and shows that it is possible.

Applying Language-Ext to a gaming context has revealed that FP, at least in a hybrid capacity can be used to control game loops, updating content, managing level code and almost all other parts of an existing game.

An appreciation for limiting state change, through pure functions or expressing logic a descriptive components has not only made the code more testable in the former case but has made it more descriptive in the latter.

The existing literature research has revealed the many merits of FP and has provided validation of the merits of applying its useful characteristics to game development. The limitations of FP are also equally useful. These are interesting and important considerations when approaching an imperative-to-functional code refactoring effort.

Many norms have been challenged through application of FP style such as the usage of NULL, exceptions and representation of logic as expressions instead of statements and has personally widened the scope of what is possible in designing game code.

Monads have shown that they are useful in encapsulating values and errors so that functions always return faithfully to the caller. Pipelining has shown that entire functions can be reduced to a single line to show how it can be composed as series chained expressions instead of multi-line statements. This might contribute to improving the code conciseness but surprisingly, was not found not to be the case however.

The use of smart constructors has also shown that Optional types can serve to facilitate validation prior to constructing new types, which removes the possibility of exceptions in constructors, another potential source of critical runtime failures.

Finally the use of Monads through Option and Either<L,R> data structures has shown the utility in their inherent ability to cancel execution of remaining code through fully embracing lazy evaluation through their built-in transformation validation behaviour, or short-circuiting.

A strategy to iteratively refactor an existing imperative codebase including utilizing transitionary approaches and techniques has also been described. This allows an existing codebase to continue to function while it is being refactored without propagating a cascade of refactoring effort. This is useful when considering productivity consequences of refactoring a game such that it can be iteratively play-tested while refactoring. This suggests that it would be compatible with incremental development methodologies such as agile.

The source code provides a useful example of how the FP concepts have been implemented and this research's approach, objectives and results are useful to prospective programmers interested in using the benefits of FP within MonoGame, or C# generally. It also provides a practical example of how to use Language-Ext, and alongside the Understanding-Language-Ext tutorial outlined in objective 2, provides a helpful set of tools in pursuing the application of FP in C# using Language-Ext.

An interesting and surprising result is that through implementing pure functions and evaluating the code coverage that those functions provide, results in the game code being immediately more testable.

Game code tends to be highly interdependent and by isolating logic into composable components, such as those afforded by pure functions, has made creating tests that validate their operations possible. This is an important consideration with respects to quality such as maintainability. These function can also be re-used and implemented as libraries in new projects, as they have little to no dependencies. The approach however to put all pure functions into a new static type is likely to have contributed to the increased dependencies in functions, however.

It is a matter of opinion as to whether the resulting code is more readable in light of its declarative and more descriptive transformation. However, while this and code aesthetics was not measured, there is likely to be an improvement due to logic being more thoroughly annotated using functional composition and declarative style and could be a future avenue of research.

That a game continues to function despite a significant underlying change to its code is testament that these ideas and concepts can be implemented, not least within a game but generally in any C# application, however a measured approach should always be taken.

The refactoring process however is limited to a simple approach of substituting existing code for an alternative more functionally orientated approach. The strategy focuses on incorporating almost like-for-like replacement strategy without specifically paying direct attention to how to improve the code it is replacing. The focus was on achieving behavioural parity with the original game, which has been accomplished.

In this way, code that has been refactored could be improved to enhance its underlying functional characteristics, and potentially improve the results.

A limited implementation of enforcing immutability was used while refactoring MonoMazer. It is possible that a more thorough and performant implementation would better showcase its benefits and/or contribute to the complexity analysis of the resulting codebase.

While exception handling is minimized, it does however exist at the the fringes of the pipelines (spokes) to detect catastrophic failures. Due to the nature of MonoGame, it is not possible to change the return type of the *Update()*, *Draw()* and associated functions and as such these functions represent a terminal point at which failures need to be dealt with somehow. This is necessary and an inevitable need to handle failures that have propagated from the core functional components in the game. In MonoMazer an exception is thrown in these terminal functions and this is considered to represent a catastrophic game failure.

A limited implementation of the hub-and-spoke distribution model was used in refactoring MonoMazer, with implementation of pure functions serving as the core hub mechanism and supporting imperative code serving as the spoke analogy, however the code is a representative example.

Not all aspects were fully implemented for example, in some cases while exceptions were mitigated through enclosing most (if not all) function bodies through retuning the Ensure() HOF, the bodies of the functions were not always refactored to implement pipelining for example and still contain some imperative programming.

A difficulty in approaching Language-Ext and indeed functional programming in C# has been understanding how it can be implemented and in what scenarios it is most usefully applied.

The Language-Ext library, without explanation or examples to assist in implementation is difficult to become accustomed to. This was certainly my experience. However by constructing this tutorial in a step-by-step fashion, incorporating many of the elements that exist, particularly around the operations that work on the Monad types Either<L, R> and Option has helped to see them work in the most basic of scenarios such that it is easier to understand and then incorporate them into more advanced scenarios such as implementation within game code.

Also, important is the tutorial's ability to reduce the technicalities of conveying technical syntax or detailing approaches without going into the details which would distract from the true purpose of this research which is not to explain how Language-Ext works but to use it to understand its impact and apply functional programming within a game's codebase.

The refactoring would have been a lot more difficult without a reference point to how the various Language-Ext constructs work practically and should be a useful contribution as part of this project.

Overall, the project has stretched and improved my abilities and has allowed me to overcome many initial deficiencies that only practical application would realistically improve, mostly around functional programming ideas, implementation problems and complexity in general.

6.1 Contributions

In this project FP is compared with IP to highlight its differences, particularly within context of game development, exploring its benefits and limitations and the impact it has had to game development.

Its techniques and paradigms have also been discussed and evaluated in context with the problems which they solve.

Likewise, the rationale and the practical application within game has been demonstrated in the accompanying source code.

The core Language-Ext syntax used in the game code has been explained through a comprehensive tutorial, in a practical fashion that can be used towards future games and/or other programs and the code complexity has been analysed and evaluated.

The resulting source code, fully functional and implementing the FP paradigms and techniques introduced and discussed in this project game illustrates exactly how such an imperative codebase can be refactored in a hybrid imperative-functional way using Language-Ext, C# and MonoGame. The choices that were made to reach a behavioural equivalent have also been discussed in this report.

6.2 Conclusions

The results show that there is a significant increase in the number of functions, types and code in general required to refactor MonoMazer into its functional equivalent. Also, MonoMazer is not particularly complex to begin with in terms of the average cyclomatic complexity, however non-the-less the cyclomatic complexity (a central focus for this project) has increased significantly in the FP codebase, along with the number of functions and types and generally the amount if code introduced.

The overall cumulative Cyclomatic Complexity as reported by VSCA rating appears to be more biased against functional codebases and appears to be more heavily penalised by its inherently disproportionate number of functions compared to imperative codebases. With the introduction of more functions this has resulted in obtaining a higher overall complexity total.

This is seen in the more than 50% increase in total cumulative complexity. However, through further analysis of cyclomatic complexity per function, which attempts to reduce the inherent biases, this sees a more modest increase of only approximately a quarter, ie 24% overall and even less considering the median function at a 13% increase overall.

These metric may improve the evaluation of the effects across functions and types generally, and as such should be considered as perhaps a more functional perspective particularly in function-heavy code along with the arguably more imperatively biased cumulative results. Crucially however and irrespective on the perspective, is that there has not been an improvement in Cyclomatic Complexity.

This project has also revealed that complexity is multi-faceted, and evaluation of code complexity alone is perhaps not enough to assess the true empirical effect of complexity generally and might be a useful topic for future research as its results are likely to be relevant to aspects such as code understandability or comprehension which is likely to provide a richer picture than that which code complexity metrics such as Cyclomatic Complexity can provide.

When evaluating and applying functional programming paradigms and techniques in developing games, particularly in C#, and re-iterating what John Carmack, Tod Sweeney and the existing literature suggests, there really "... is real value in pursuing functional programming..." and incorporating some of the paradigms and techniques presented in this project could realistically improve aspects of future game code - however its unlikely that it will improve its cyclomatic complexity.

Chapter 7

Appendix A: Project Proposal for MSc in Computer Games Technology

7.1 Evaluating and Applying Functional Programming Paradigms in Developing Computer Games

7.1.1 Introduction

The primary purpose of this research is to introduce functional programming paradigms and techniques into an existing game, which otherwise are not exhibited in the original design. Secondly, is to produce a tutorial for LanguageExt (a functional programming library) describing qualitatively these paradigms and techniques, in a generic fashion, irrespective of the domain of its application. Third, is to provide a comparative analysis of the code changes before and after using automated static code quantitative analysis, focusing primarily on code complexity effects such as cyclomatic complexity.

Research questions:

1. How can an existing, imperative game (one that is designed around state changes) be re-implemented to take advantage of using functional programming paradigms using C#, MonoGame and LanguageExt.
2. What are the qualitative benefits of using functional programming in game development and how are they realised in a game (eg concurrency/parallelism, increases reliability, promotes more expressive logic, reduction of repetition/duplication/bugs)
3. How does functional programming affect the game's physical code structures and how does this affect its complexity and associated metrics

To achieve 1, a manual code rewrite of the existing game source code will be produced incorporating into its design functional programming paradigms and techniques using the Language Ext library along with an accompanying tutorial describing them.

To achieve 2, the application of functional programming paradigms into key aspects of the game code will serve as means to qualitatively articulate the specific implementation of the functional design benefits.This will additionally be described in the accompanying report.

To achieve 3, the code will be analysed using industry code analysis tools before and after modifications using a mixed approach however focusing on the quantitative measures of complexity and associative measures (Cyclomatic complexity and # of dependencies etc)

Scope and Objectives

Build 1 objectives:

Produce a running game as an artefact, including the modified source code, which will be the result of the redesign incorporating the functional programming paradigms and techniques into the existing game.

Build 2 objectives:

Produce a tutorial as an artefact, which will illustrate and describe the use of specific functional programming paradigms using the LanguageExt library, consisting of a walk-through narrative and associated C# code samples hosted on Git. (This will later be used to identify and illustrate specific functional paradigms and techniques which will be used in the subsequent redesign of the game's source code to incorporate these functional paradigms.)

Build 3 objectives:

Produce a static code analysis both before and after to glean insights into the impact of the changes brought about by incorporating the functional programming paradigms and techniques, including code complexity, conciseness(LOC), expressiveness etc. - details of which are covered later in this proposal under the *Methods* section in *Planned Analysis* subsection.

Beneficiaries:

1. MonoGame game developers specifically and/or C# game developers generally.
2. Software Developers interested in learning Functional Programming in C# to evaluate its benefits and disadvantages and how its realised in a practical fashion.
3. Unity game developers
4. Lecturers and academics who might wish to gain insight into functional programming paradigms and techniques, irrespective of the implementation language.

7.1.2 Critical Context

Functional programming is a set of programming paradigms, based on mathematical ideas that promote the immutability of program state wherever possible to achieve reliability, robustness, and expressiveness of computation, "...in contrast to an imperative program, which consists of a series of statements that change global state when executed, a function program models computation as the evaluation of expressions" (Butcher, 2014).

Avoiding state change(and its many benefits) is at odds with the traditional view of describing computation through changes in state, however , "..functional concepts are being introduced into classic object-oriented technologies such as Java and .NET. While this trend may not result in a complete paradigm-shift,"paradigm-mix" is already well established" (Turek, et al, 2018).

This research aims to align with this shift, particularly with the advent of functional programming libraries such as LanguageExt which can be used to extend otherwise non-native functional programming languages such as C# to take advantage of some of the functional paradigms mentioned above, and which could be used in gaming context that utilize C# such as Unity and MonoGame.

This includes the use of higher-order functions, *Functors* and *Monads*. (Functors apply a function to a contained/wrapped value, and Monads apply a function to a contained/wrapped value that returns wrapped value. HOFs refer to passing functions as parameters)

Furthermore, research has shown that functional code is "significantly" more concise than procedural code and that "shorter programs will tend to have fewer bugs." (Nanz et al, 2015).

This alone is a compelling reason to use functional programming, not least within games.

Minskey et al highlights that the improvements that functional programming paradigms bring specifically helps to promote a more robust, reliable and correct codebase which is more predictable in real world circumstances, particularly where state mutation is the progenitor of unpredictability.

Problems of reliability and uncertainty embedded in imperative style programming - which otherwise would be curtailed by functional designs - expose particular problems inherent to real-time, concurrent and multitasking execution environments such as games where task concurrency and parallelism are important mechanisms (Murphy et al, 2018) particularly as predictability is essential when reasoning about thread safety (Butcher, 2014).

This research will thus look to identify and implement functional programming paradigms which will effectively mitigate the pitfalls that imperative code brings when it comes to ensuring predictability at runtime within games.

Some representable paradigms in this regard, includes techniques such as the removal of NULL types and their remediation through use of Option<T> monads, along with other techniques that prevent of potential state changes such as removing traditional conditional exception handling constructs through the use of monadic types such as Either<L,R> to express failure and use of 'pure' functions, in general to promote function reliability.

This research will also look to implement structures to mitigate 'impure' functions which otherwise can return unexpected results due to mutability of states in the program, including those that throw exceptions. (Impure functions produce side-effects and do not depend exclusively on their input)

Using Higher-order functions (HOFs), which alongside immutability is a key tenant in functional programming which contributes also in the separation of concerns, deduplication of code and contributes to code concieceness.(Buonanno, 2017). These aspects will be incorporated into the functional design of the existing source code and exposed in subsequent analysis.

As these aspects are of direct importance for real-time programming environments like games, particularly reliability of functions in concurrent domains and simulations (Turek et al, 2018), a practical application of functional programming paradigms in the context of game code, would provide useful insight into how they can be achieved.

Functional programming paradigms have largely only been used and developed using specific functional languages which have by default explicit support for promoting these paradigms. These languages include Haskell, Lisp and Erlang to name a few - programming languages which are not routinely used when creating computer games as opposed to imperative alternatives such as C/C++ including C# as is the case with MonoGame and Unity.

Those specific functional languages include native support for typical functional constructs such as Monads and associated data types such as Functors which make key design aspirations such as immutability and other functional paradigms, part of the language itself.

[LanguageExt] introduces key functional "extensions" to the C# language such as the use of Monadic types such as Either<T> and Option<T> and along with C#'s partial support for immutable language features such as LINQ, Tuple data type etc, can be used to facilitate the more substantial functional paradigms introduced by LanguageExt library.

7.1.3 Approaches: Methods & Tools for Design,

A design and build featuring a complete redesign of an existing, imperative C# game, re-written using newly introduced functional programming paradigms which will additionally be documented in a tutorial and presented in a report and used to expose inherent benefits and application thereof.

The game itself will use MonoGame and LanguageExt library, including the auxiliary creation of a tutorial based on the functional programming paradigms which will be used therein and which will demonstrate practical functional programming paradigms in the context of game development.

In order to understand and expose functional programming paradigms and what impact they have, the following approaches will be used:

- Construction of a step-wise functional programming tutorial for LanguageExt introducing the key concepts including monadic functions, declarative style and immutability to achieve various functional programming traits and which will form the foundation of the design considerations used in the code redesign.
- This tutorial will be referenced in a main report, which will be used to describe how and why certain aspects of functional programming are used during the redesign and to provide alternative examples and perspective. This will serve as the foundational introduction to the functional paradigms and techniques that will be employed.
- The tutorial will look to demonstrate and explain the following functional programming paradigms in LanguageExt, including providing representable code examples provided in a public source code repository: Monads, Match/Map/Bind operations, Pipelining, Declarative Style, Monad construction, Function Composition, Pure Functions, Basics of functional Data Types including Either<L,R> , Option<T> and associated operations

In order to apply these FP techniques to a game context, the ideas and techniques introduced in the tutorial will be applied practically to the existing game code:

The game rewrite process will comprise the following phases/tasks:

1. Converting all functions return types to return Monads of type Either<> or Option<> depending on the suitability - to increase reliability of functions
2. Converting methods from imperative to declarative implementation using the concepts of pipelining, higher-order functions (HOF) and utilizing some functional compatibility C# language features such as Linq and the use of monadic types to enforce short circuiting and validation through declarative compositions of functions - to enhance the expressiveness of computational logic.

3. Designing pure functions to avoid state mutations and side effects where possible including the use of Smart Constructors (to ensure reliable creation of objects) and reducing dependence on I/O as this introduces impure functions that aren't guaranteed to be correct/reliable.
4. Elimination of usage of the NULL construct which can affect reliability of function results.
5. Elimination of conditional exception handlers to reduce and possibility of sudden, alternative control paths affecting correctness of functions.

To provide a qualitative description of the process, lessons learnt, provide reflection and to articulate the redesign choices and implementation, a report which will include:

- The design decisions in the existing codebase, challenges faced, observations as well as the outlining the overall re-design process undertaken as a whole.
- A static code analysis of the source code before and after the redesign will be included, where static code analysis tools will be used to consider the effect the functional code changes have had on the code, particularly in light of the arguments about the conciseness and expressiveness that functional programming affords. This will include an analysis on complexity and readability of the codebase in comparison to the original source code.

7.1.3.1 Planned analysis

1. By way of the redesign, identification of the areas of applicability within existing game code that could be afforded by functional paradigms. Including:
    a. Identifying and then mapping each relevant/appropriate functional programming technique illustrated in Tutorial to the relevant candidate within the existing game code. (Qualitative categorization of techniques used in game)
    b. Highlight the effectiveness and utility afforded by applying the selected technique to an identified area of the existing codebase by discussing its design and the impact of the choice. Referencing the tutorial and source code as a guide. Qualitatively explain why these techniques are used in the game and the design goal is.
    c. Qualitative review and discussion of any unexpected aspects in the resulting code.
2. Use static code analysis using tools such as such as SonarCube, StyleCop eg. on the code to provide insights to code complexity, design, style and quality considerations before/after the redesign. These tools provide metrics such as the code's cyclomatic complexity, indications of potential problems i.e. code smells and will provide valuable insights to contrast code quality issues in the code before/after.

Specifically of interest are:

a. Evaluating the code for changes in code complexity and conciseness due to introduction of functional paradigms as described by (Nanz et al, 2015). Quantitatively measured using percentage change in cyclomatic complexity/ dependencies etc.

b. Code Readability: Qualitative comparison of declaratively styled(concise) implementation vs imperative functional implementation.

c. Qualitative comparative debugging experience when stepping through monadic functions and declaratively expressed logic(new) as opposed to imperative code (original).

d. Qualitatively evaluating the functionality of the game before and after - does it still work and are there any obvious changes as a result of the redesign worthy of note.

7.1.4 Work Plan

Please see full Proposal on Moodle for full work plan.

7.1.5 Risks

Please see full Proposal on Moodle for full risk assessment.

7.1.6 Ethical, Legal and Professional Issues

All contributions made by myself including the LanguageExt tutorial, game source code is based on my own work, which will be freely available to the public. LanguageExt is an open source 3rd party library developed by Paul Louth. The ideas and techniques are not the property of any other 3rd party, many have been successfully used while working in the author's professional capacity as a software developer - they represent freely transferable ideas/concepts.

Please see full Proposal on Moodle for full declaration of the Research Ethics Review Form.

7.1.7 References:

1. Buonanno, E. (2017) Functional Programming in C# (Meap Edition, Version 10)
2. Butcher, P. (2014) Seven Concurrency Models in Seven Weeks: When Threads Unravel.
3. Murphy, J.C., Shivkumar, B., Pritchard, A., Iraci, G., Kumar, D., Kim, S.H. & Ziarek, L. 2019, "A survey of real-time capabilities in functional languages and compilers", Concurrency and Computation: Practice and Experience, vol. 31, no. 4, pp. e4902-n/a.
4. Minsky, Y. (2008) 'Caml trading', Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, 43(1), pp. 285–285. doi: 10.1145/1328438.1328441.
5. Nanz, S. and Furia, C. A. (2015) 'A Comparative Study of Programming Languages in Rosetta Code', in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, pp. 778–788. doi: 10.1109/ICSE.2015.90.
6. Turek, W, Byrski, A, Hughes, J, Hammond, K, Zaionc, M. Special issue on Parallel and distributed computing based on the functional programming paradigm. *Concurrency Computat Pract Exper.* 2018; 30:e4842. https://doi.org/10.1002/cpe.4842
7. haskell - Why do we need monads? [WWW Document], n.d. . Stack Overflow. URL https://stackoverflow.com/questions/281 do-we-need-monads (accessed 4.2.20)
8. louthy/language-ext: C# functional language extensions - a base class library for functional programming [WWW Document], n.d. URL https://github.com/louthy/language-ext (accessed 3.28.20).

Chapter 8

Appendix B

8.1 Brief introduction to MonoMazer

The goal of the game is to collect all the balloons which are pickups. Once all the pickups are gathered, you progress along with your points and health to the next level where you encounter new enemies and a different level layout.

You move the human character using the arrow keys. You can walk through walls if you need to. Colliding with enemies will reduce your health, while collecting pickups will increase your points.

You can pause the game at any time by pressing escape, to bring up the main menu.

For more details on the other keystrokes available(mainly for debugging and testing) see Mazer.cs

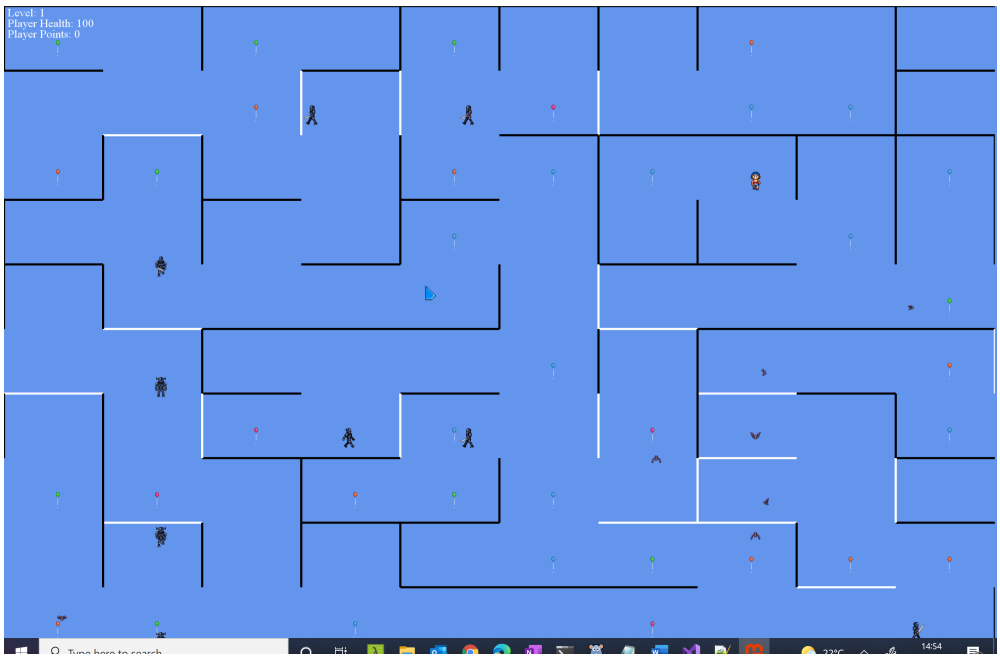

Figure 8.1: MonoMazer in action

A video demonstration along with the game architecture is available https://www.stuartmathews.com/index.php/gaming/1051-monomazer

8.2 Reproduction of Results

MonoMazer uses MonoGame 3.7.1 which can be found at https://community.monogame.net/t/monogame-3-7-1/11173.

LanguageExt, and other dependant libraries are automatically downloaded when the project is restored by NuGet.

The report itself is generated using Pandoc by invoking the make.sh script in the project directory. WSL was used as the executing environment. Both can be found at (https://pandoc.org/) and (https://docs.microsoft.com/en-us/windows/wsl/install) respectively.

The data collection process consists of running Visual Studio Code Analyser on the MonoMazer solution in both the 'before' branch and the 'beyond' branch.

This can be done invoking the analyser from the 'Analyze 'Visual Studio Menu, followed by 'Run Code Analysis' and then 'Run Code Analysis on MazerPlatformer':

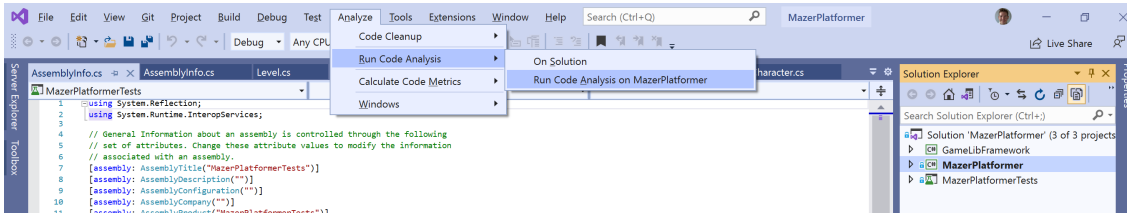

Figure 8.2: Running Visual Studio Code Analyser

The result of the analysis can be exported from the 'Code Metrics Results' window as 'after.xlsx' or 'before.xlsx' depending on which branch on MonoMazer you are on:

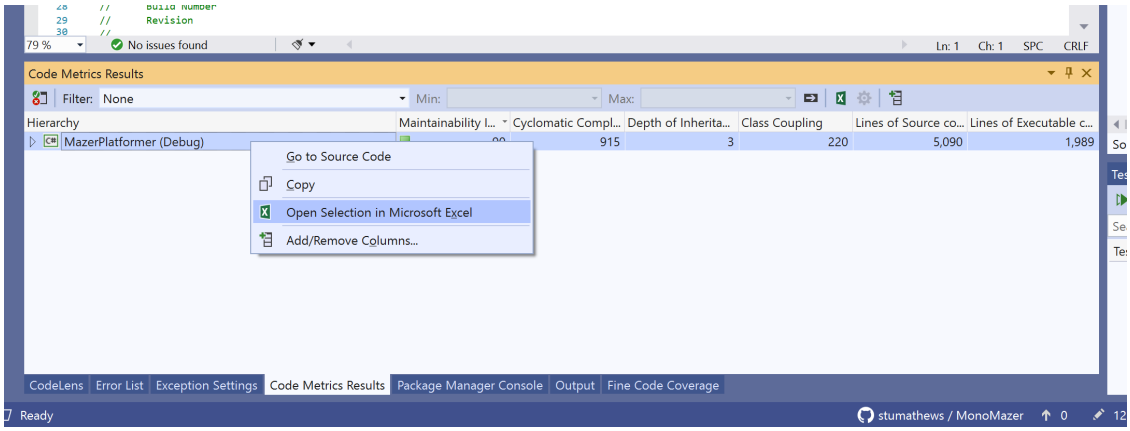

Figure 8.3: Exporting the code analysis results

The graphs that are used in the report are generated using matplotlib and the Jupyter notebook named 'Thesis-Stats.ipynb' which is in the project directory.

Further analysis required the pandas and numpy libraries, which can be found at https://pandas.pydata.org/ and https://numpy.org/ respectively.

The paths to the aforementioned exported code analysis results files can be configured in the notebook and are used as input for the subsequent anslysis:

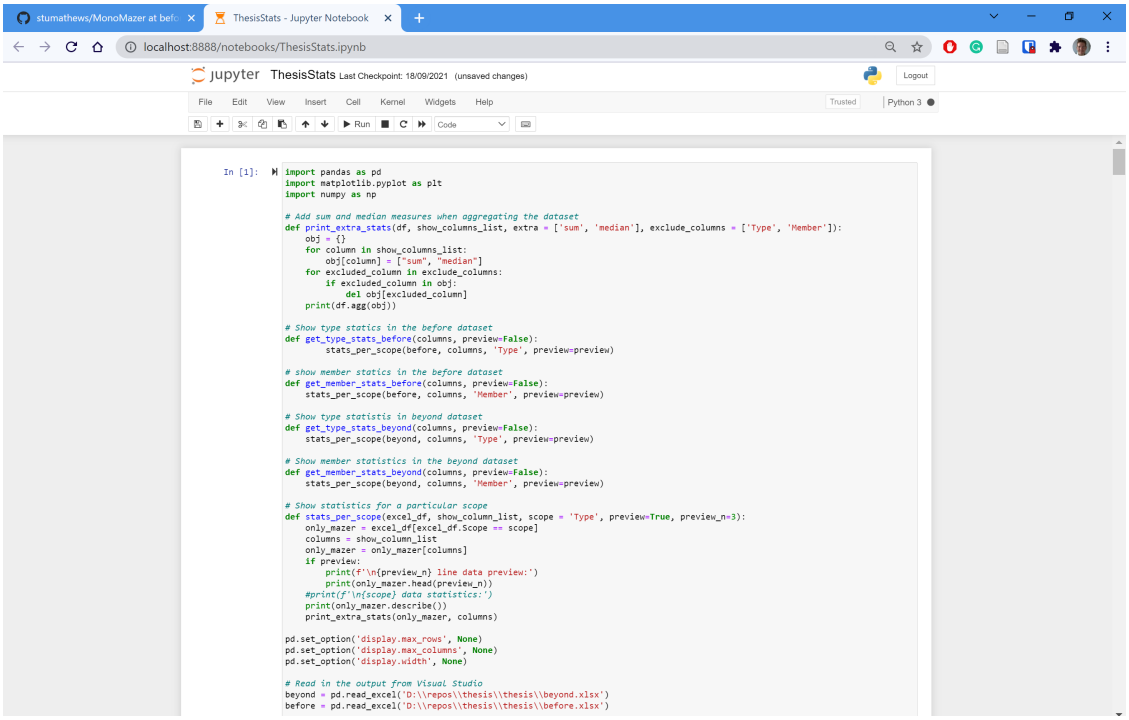

Figure 8.4: Configuring the notebook to use analysis results

Running the notebook will perform the anayslis and generate the graphs:

Figure 8.5: Generating the graphs and performing additional analysis

8.3 Commit History

Commit #	Task #	Task Description	Commit Message
1	FP-9	Return Eithers	First pass at converting return void types to Either
2	FP-9	Return Eithers	Convert return types to IFailures
3	FP-9	Return Eithers	more either return types
4	FP-9	Return Eithers	More return type conversions
5	FP-9	Return Eithers	Return types still
6	FP-9	Return Eithers	More either return types for functions
7	FP-9	Return Eithers	More returning eithers
8	FP-9	Return Eithers	Mazer game class all returns eithers
9	FP-15	Refactoring Misc	Refactoring
10	FP-15	Refactoring Misc	Refactoring
11	FP-16	Create Pure functions	Make GetCenter static and start journet of making it pure
12	FP-16	Create Pure functions	Modify incoming parameter as return as opposed to modify state of underlying class member
13	FP-17	Create Utility functions	Decide to what to do with a failure when you get one - in this case ignore and return as unit - simulates success
14	FP-15	Refactoring Misc	Rename ToFailure as ToEitherFailure
15	FP-17	Create Utility functions	Create an Ensured() version of DoIff with EnsureIff()
16	FP-17	Create Utility functions	Use EnsureIf()
17	FP-15	Refactoring Misc	Use toFailure rather than ToEitherFailure
18	FP-15	Refactoring Misc	Create EnsureIf() that can return type Rename DoIfReturn() => DoIf()
19	FP-11	Avoid Exceptions	Use EnsureIf
20	FP-15	Refactoring Misc	Remove void returning doIf
21	FP-15	Refactoring Misc	Add documentation to EnsureIf-style functions
22	FP-15	Refactoring Misc	rename action to then
23	FP-17	Create Utility functions	Introduce IgnoreFailure overload as a handy way to ignore failures and return Nothing implicitly.
24	FP-17	Create Utility functions	Add EnsuringMap and Bind to automatically ensure binding/transformation functions
25	FP-11	Avoid Exceptions	Ensure block of unsafe code
26	FP-11	Avoid Exceptions	Finding where failures occur is difficult with pipelining. Adding some context to failures...
27	FP-15	Refactoring Misc	Causes entire drawing pipeline to break if the failure is not ignored (which is ok btw)
28	FP-15	Refactoring Misc	refactoring
29	FP-15	Refactoring Misc	commentary
30	FP-17	Create Utility functions	EnsuringBind will act like a normal Bind EnsuringMap will act like a normal Map
31	FP-17	Create Utility functions	Use version of try this that will put exceptions within Failures.
32	FP-17	Create Utility functions	Using ensureing Bind and Map
33	FP-15	Refactoring Misc	Create IFailure from concrete ConditionNotSatisfied class
34	FP-17	Create Utility functions	Allow aggregate faliures to specify the right type when aggregating failures
35	FP-17	Create Utility functions	Use AggregateUnitFailures
36	FP-17	Create Utility functions	use aggregateUnitFailures
37	FP-15	Refactoring Misc	Remove reference to AddToGameObjects2
38	FP-17	Create Utility functions	Add ThrowIfNone to simulate an unrecoverable situation when something expected was not found
39	FP-10	Remove NULLs	Good example of eradicating NULLs with Option
40	FP-15	Refactoring Misc	Refactoring
41	FP-10	Remove NULLs	Another good example of when to use Option
42	FP-10	Remove NULLs	Innovative use of Option on GetMinMaxRange. Also make other functions to return Options
43	FP-15	Refactoring Misc	Add an attribute to easily identify functions that are known to be pure

Commit #	Task #	Task Description	Commit Message
44	FP-11	Avoid Exceptions	Ensure some Remove
45	FP-15	Refactoring Misc	Simple refactoring in GameWorld
46	FP-15	Refactoring Misc	include pure attribute in project
47	FP-17	Create Utility functions	Use ThrowIfNone or ThrowIfFailed as stop gap to having to deal with Monads in tightly imperative code
48	FP-15	Refactoring Misc	Move classes into there own files
49	FP-11	Avoid Exceptions	Add Creation function to failure
50	FP-15	Refactoring Misc	move out classes into own files
51	FP-9	Return Eithers	Got almost all GameObject functions returning Either
52	FP-15	Refactoring Misc	include new files created to house refacting of exsting classes into new files to project
53	FP-9	Return Eithers	Quick win converstions from void to Either
54	FP-18	Fix discoveered bug	Bug! Converting returning bool to Either
55	FP-15	Refactoring Misc	Add a generic failure that you can use to overrride the failure in ThrowIfFailed()
56	FP-17	Create Utility functions	Overload throwifFailed() to override failure with custom user provided one
57	FP-9	Return Eithers	Returning eithers of ifailure
58	FP-15	Refactoring Misc	include class in project (UnexpectedFailure)
59	FP-18	Fix discoveered bug	return eithers - this one introduced a bug as an if(x) is atomatically true when x is Either
60	FP-15	Refactoring Misc	Move classes into their own files
61	FP-23	Use Adapter ThrowIf_x	This is a useful way to continue to use the previous signature but use an adapter that will ThrowIfFailed()
62	FP-23	Use Adapter ThrowIf_x	Good use of using adapter functions to keep initial interface but remove throwIfFailed once the interface has been changed to Either
63	FP-15	Refactoring Misc	commentary
64	FP-11	Avoid Exceptions	Update could fail on SimpleGameTimer
65	FP-15	Refactoring Misc	Add comments and TODOs
66	FP-15	Refactoring Misc	Move failures into a failure folder
67	FP-15	Refactoring Misc	This is not needed.
68	FP-15	Refactoring Misc	Refactoring
69	FP-15	Refactoring Misc	Refactoring
70	FP-17	Create Utility functions	Add FalIfTrue and FailIfFalse and ShortCircuitIfTrue
71	FP-22	Make more readable	Trying to make functional programming look more readable!! (BIAS!!)
72	FP-19	Make code declarative	Impressive simplification using Linq Query syntax when parameters are now enforced to be Monads (Option)
73	FP-18	Fix discoveered bug	Fix a bug that only became noticible through using Ensure() and Either
74	FP-22	Make more readable	Make intent clear, we're returning nothing.
75	FP-19	Make code declarative	I think using linq query syntax is much more redable and avoids seeing bind/map which can confuse people and makes it more difficult to read in my opinion
76	FP-15	Refactoring Misc	Interesting note about Ensuring delegates - ie ensuring they dont return failures from receiver
77	FP-17	Create Utility functions	Some interesting helper methods
78	FP-19	Make code declarative	Convert everything to Linq Query syntax. I like this much better than Bind/Map (perfomance possibly going down??)
79	FP-17	Create Utility functions	Throw if None for an Option
80	FP-15	Refactoring Misc	new InvalidDataFailure Failure type
81	FP-11	Avoid Exceptions	Use a Smart Constructor on GameWorld. Trivial validations not that important right now.
82	FP-15	Refactoring Misc	Refactoring - favour less one line functions in favour of inlining them instead

Commit #	Task #	Task Description	Commit Message
83	FP-17	Create Utility functions	TryLoad
84	FP-15	Refactoring Misc	move to its own file (uninitialized failure)
85	FP-15	Refactoring Misc	refactoring
86	FP-17	Create Utility functions	Finally, a EnsuringBind on an action
87	FP-15	Refactoring Misc	Refactoring - making things a little more readable
88	FP-17	Create Utility functions	Favour Must() instead of Require()
89	FP-19	Make code declarative	I like From statements
90	FP-15	Refactoring Misc	Refactor
91	FP-15	Refactoring Misc	Minor refactoring
92	FP-16	Create Pure functions	Just pass in rndom room directly not entire collection
93	FP-15	Refactoring Misc	Move to failures folder
94	FP-17	Create Utility functions	AggregateFailures returns orignal non-failures if no failures occured
95	FP-21	Pipelining Code	Turn these into more functional representatives
96	FP-15	Refactoring Misc	Refactoring
97	FP-21	Pipelining Code	Make Loading of Level code functional
98	FP-15	Refactoring Misc	rename variable
99	FP-15	Refactoring Misc	Make Level more functional
100	FP-15	Refactoring Misc	Use safer function method
101	FP-9	Return Eithers	Get rid of void returning functions
102	FP-21	Pipelining Code	functionalize!
103	FP-15	Refactoring Misc	Correct spelling of Diagnostics
104	FP-9	Return Eithers	Make sure the collision code is returning an either and that the caller code resolves it or throws
105	FP-15	Refactoring Misc	refactoring
106	FP-9	Return Eithers	eliminate void function
107	FP-9	Return Eithers	Return Either
108	FP-9	Return Eithers	return either
109	FP-15	Refactoring Misc	This is only class that can return non-eithers as it depends on root eithers that throw if failed
110	FP-15	Refactoring Misc	minor refactoring
111	FP-11	Avoid Exceptions	Use smart constructor (without validation!) for Creating a Room
112	FP-11	Avoid Exceptions	Add validation to room smart constructor
113	FP-11	Avoid Exceptions	Pull out potential throwing code out of constructor and into own method called after smart constructor validates creation
114	FP-21	Pipelining Code	Ensure all drawing operations in the pipeline succeed.
115	FP-21	Pipelining Code	Pipeline initializing the character animation
116	FP-15	Refactoring Misc	refactoring
117	FP-12	Implement Immutability	First Attempt at copy on write data structure
118	FP-15	Refactoring Misc	Minor refactoring
119	FP-12	Implement Immutability	Immutable Field Test
120	FP-12	Implement Immutability	Add to unused and untested features (NoForceOverwiteLock & ForceLockOnAccessIfLocked)
121	FP-12	Implement Immutability	Buildable NewObject from Value stacks
122	FP-12	Implement Immutability	Preliminary but incomplete support for Immutable field with NewObject
123	FP-15	Refactoring Misc	Add commentary
124	FP-12	Implement Immutability	Must purge cache if you want to create a 'new' NewObject as we're essentially a Versioned
125	FP-12	Implement Immutability	start to make provision for using
126	FP-18	Fix discoveered bug	Profiling showed that looking up the stack trace was the reason the game slowed down so much
127	FP-12	Implement Immutability	VersionedStack Research Prototype
128	FP-16	Create Pure functions	Make many function in Room static and thus reusable/testable and easily composable

Commit #	Task #	Task Description	Commit Message
129	FP-12	Implement Immutability	First immutable copy function using XML serialization
130	FP-17	Create Utility functions	First preliminaries of an EnsureWithBindingReturn type function
131	FP-15	Refactoring Misc	identify which one is used for ensuringbind()
132	FP-15	Refactoring Misc	tidy up
133	FP-16	Create Pure functions	Pull in Newtonsoft for serialization
134	FP-16	Create Pure functions	First real pure function using newtonsoft to copy arguments
135	FP-15	Refactoring Misc	Refactoring
136	FP-15	Refactoring Misc	Refactoring
137	FP-15	Refactoring Misc	Refactoring
138	FP-12	Implement Immutability	Remove room references to avoid self-referencing.
139	FP-19	Make code declarative	Make more declarative
140	FP-15	Refactoring Misc	minor refactoring
141	FP-15	Refactoring Misc	dont need list of player components
142	FP-15	Refactoring Misc	refactoring
143	FP-15	Refactoring Misc	use new way of assigning on casting of Either
144	FP-16	Create Pure functions	Add more statics to gameworld statics
145	FP-18	Fix discoveered bug	This was what was causing the slowndown and garbage collection - multiple exceptions of indexoutofbounds!
146	FP-15	Refactoring Misc	Make sure we clear out and dispose as much as we can on Unload()
147	FP-15	Refactoring Misc	refactoring
148	FP-16	Create Pure functions	add more gameworld statics!
149	FP-16	Create Pure functions	Add Tests for static functions
150	FP-20	Removal of conditionals	Example of removal of conditional? via IfSome
151	FP-15	Refactoring Misc	remove unused using blocks thoughout solution
152	FP-16	Create Pure functions	Abstract IO or state change as function paramters!! Also pass read-only access to params instead og modifying local class members!!
153	FP-15	Refactoring Misc	Refactoring
154	FP-15	Refactoring Misc	Refactoring
155	FP-11	Avoid Exceptions	Throw exception is cannot remove game object

References

Backus, J. (1978) 'Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs', *Communications of the ACM*, 21(8), pp. 613–641. doi: 10.1145/359576.359579.

Blow, J. (2004) 'Game Development: Harder Than You Think: Ten or twenty years ago it was all fun and games. Now it's blood, sweat, and code', *ACM queue*, 1(10), pp. 28–37. doi: 10.1145/971564.971590.

Bonelli, N. *et al.* (2014) 'A purely functional approach to packet processing', in *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems - ANCS '14*. Los Angeles, California, USA: ACM Press, pp. 219–230. doi: 10.1145/2658260.2658269.

Buonanno, E. (2017) 'Functional Programming in C#', *Manning Publications*. Available at: https://www.manning.com/books/functional-programming-in-c-sharp (Accessed: 28 August 2021).

Butcher, P. (2014) 'Seven Concurrency Models in Seven Weeks'. Available at: https://pragprog.com/titles/pb7con/seven-concurrency-models-in-seven-weeks (Accessed: 28 August 2021).

Carmack, J. (2012) 'In-depth: Functional programming in C++'. Available at: https://www.gamedeveloper.com/programming/in-depth-functional-programming-in-c- (Accessed: 28 August 2021).

Goldberg, B. (1996) 'Functional programming languages', *ACM computing surveys*, 28(1), pp. 249–251. doi: 10.1145/234313.234414.

Hanus, M. (2007) 'Putting declarative programming into the web: Translating curry to javascript', in *Proceedings of*

the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming. New York, NY, USA: Association for Computing Machinery (PPDP '07), pp. 155–166. doi: 10.1145/1273920.1273942.

Hudak, P. (1989) 'Conception, evolution, and application of functional programming languages', *ACM computing surveys*, 21(3), pp. 359–411. doi: 10.1145/72551.72554.

Jbara, A., Matan, A. and Feitelson, D. G. (2014) 'High-MCC Functions in the Linux Kernel', *Empirical software engineering : an international journal*, 19(5), pp. 1261–1298. doi: 10.1007/s10664-013-9275-7.

Kühne, T. (1994) 'Higher order objects in pure object-oriented languages', *SIGPLAN notices*, 29(7), pp. 15–20. doi: 10.1145/181593.181595.

Lincke, D. and Schupp, S. (2009) 'The function concept in C++: An empirical study', in. ACM (WGP '09), pp. 25–36. doi: 10.1145/1596614.1596619.

Lopez, A. M. (2001) 'Supporting declarative programming through analogy', in *Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*. Evansville, IN, USA: Consortium for Computing Sciences in Colleges (CCSC '01), pp. 53–65.

Louth, P. (2021) 'C# Functional Programming Language Extensions'. Available at: https://github.com/louthy/language-ext (Accessed: 3 September 2021).

Louthy, P. (no date) 'Louthy/language-ext', *GitHub*. Available at: https://github.com/louthy/language-ext (Accessed: 28 March 2020).

Maggiore, G., Bugliesi, M. and Orsini, R. (2011) 'Monadic Scripting in F# for Computer Games', *undefined*. Available at: https://www.dsi.unive.it/~orsini/documenti/MonadicScripting2.pdf (Accessed: 28 August 2021).

Mathews, S. (2021) 'MonoMazer'. Available at: https://github.com/stumathews/MonoMazer (Accessed: 3 September 2021).

McCabe, T. (1976) 'A Complexity Measure', *IEEE Transactions on Software Engineering*, SE-2(4), pp. 308–320. doi: 10.1109/TSE.1976.233837.

Mikejo5000 (no date) 'Code metrics - Class coupling - Visual Studio (Windows)'. Available at: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-class-coupling (Accessed: 29 August 2021).

MINSKY, Y. and WEEKS, S. (2008) 'Caml trading – experiences with functional programming on Wall Street', *Journal of functional programming*, 18(4), pp. 553–564. doi: 10.1017/S095679680800676X.

Murphy, J. C. *et al.* (2019) 'A survey of real-time capabilities in functional languages and compilers', *Concurrency and computation*, 31(4), p. n/a. doi: 10.1002/cpe.4902.

Murphy-Hill, E., Parnin, C. and Black, A. P. (2012) 'How We Refactor, and How We Know It', *IEEE transactions on software engineering*, 38(1), pp. 5–18. doi: 10.1109/TSE.2011.41.

Nanz, S. and Furia, C. (2015) 'A comparative study of programming languages in Rosetta code', in. IEEE Press (ICSE '15), pp. 778–788. doi: 10.1109/ICSE.2015.90.

Nilsson, H. and Perez, I. (2014) 'Declarative Game Programming: Distilled Tutorial', in *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. New York, NY, USA: Association for Computing Machinery (PPDP '14), pp. 159–160. doi: 10.1145/2643135.2643160.

Schroeder, M. (1999) 'A practical guide to object-oriented metrics', *IT professional*, 1(6), pp. 30–36. doi: 10.1109/6294.806902.

Sowell, B. *et al.* (2009) 'From Declarative Languages to Declarative Processing in Computer Games', *arXiv:0909.1770 [cs]*. Available at: http://arxiv.org/abs/0909.1770 (Accessed: 28 August 2021).

Sweeney, T. (2006) 'The next mainstream programming language: A game developer's perspective', in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: Association for Computing Machinery (POPL '06), p. 269. doi: 10.1145/1111037.1111061.

Teatro, T. A., Mikael Eklund, J. and Milman, R. (2018) 'Maybe and Either Monads in Plain C++ 17', in *2018 IEEE Canadian Conference on Electrical Computer Engineering (CCECE)*, pp. 1–4. doi: 10.1109/CCECE.2018.8447584.

'TIOBE INDEX TIOBE - THE SOFTWARE QUALITY COMPANY' (2020). Available at: https://www.tiobe.com/tiobe-index/ (Accessed: 24 October 2020).

Turek, W. *et al.* (2018) 'Special issue on Parallel and distributed computing based on the functional programming paradigm', *Concurrency and computation*, 30(22), pp. e4842–n/a. doi: 10.1002/cpe.4842.

Wise, D. S. (2003) 'Functional programming', in *Encyclopedia of Computer Science*. GBR: John Wiley; Sons Ltd., pp. 736–739.

Wyatt, R. (2003) 'Understanding functional programming', *Journal of Computing Sciences in Colleges*, 18(5), pp. 109–117.

Xu, Y., Jia, X. and Xuan, J. (2019) 'Writing Tests for This Higher-Order Function First: Automatically Identifying Future Callings to Assist Testers', in. ACM (Internetware '19), pp. 1–10. doi: 10.1145/3361242.3361256.

Zeller, C. and Perez, I. (2019) 'Mobile game programming in Haskell', in. ACM (FARM 2019), pp. 37–48. doi: 10.1145/3331543.3342580.