Applying and Evaluating Pattern-Oriented Designs In Improving Code Quality In Complex Software

F66 MSc in Computing (Software Engineering)

Stuart Mathews

Module Code: T847
Assignment Number: EMA

Word count: 9388
(8774 main text + 614 words within illustrations)

Previous Modules:

Software Engineering (M814)
Software Development (M813)

Date: 01/04/2022

School of Science, Technology, Engineering and Mathematics (STEM)

Contents

List of Figures

List of Tables

Executive Summary

It is suggested that Pattern-Oriented Designs (PODs) improve the quality of software by embodying design qualities that not only enhance the quality of technical implementations, but also incorporate non-functional, contextual awareness for varying social concerns such as coping with the effects of change, improving maintainability and reducing cognitive complexity. We explore this by refactoring an existing computer game by introducing three PODs, namely the Mediator, Blackboard and Bridge POD and explore the effect this has on code quality, focusing primarily on Class Coupling, Maintainability Index and Cyclomatic Complexity.

The initial game code is split into two codebases, one representing the original game code and the other on the modified codebase incorporating candidate PODs. A static code analysis is run on both codebases to assess the code quality metrics, which are then comparatively assessed to determine the impact that the PODs have had on code quality.

The results show that the design qualities that are inherent in the candidate PODs translate into improvements in code quality. A reduction is seen in the game's overall aggregate Class Coupling and Cyclomatic Complexity with a slight improviement in its Maintainability Index.

The PODs are also shown to posess a high-level of code quality compared to existing code, and have a correspondingly high influence the overall code quality of the game.

Chapter 1

Introduction

An established view-point in Software Engineering is the importance of software quality. This is arguably because it brings together and reasons about aspects of software that reach far beyond the immediate utility of any particular technical solution.

For example, quality considers what code cannot, such as social variability in terms of its contextual use in society and how its effectiveness is calibrated and perceived in varying situational contexts. These reach out from a reasonably stable and safe technical centre, to more indeterministic and chaotic surrounding contexts, governed by people, reactions and perceptions that are based on complex social and political constraints, most of which are not specifically part of technology blueprints.

In this way, quality is arguably one of the most important aspects of software design, as it fully represents one part of the duality of software, i.e. the techno-social divide, and as such is an important application of social context to software, irrespective of how technically suitable it may be.

Societal constraints, for example, can dictate that the merits of an adequate technical solution falls short in terms of its wider integration into the problem space. The ability to improve, innovate, adapt and be robust in the face of varying social complexities is arguably a feature of design choices, and these are interwoven into choosing quality-focused choices that reach forward and carry a technological solution into the social reality that is industry.

Pattern-Oriented Designs (PODs) are part of an a world-view, that argues that they improve quality and reduce complexity in software, specifically by providing technological blueprints that incorporate many of the social implications, i.e. qualities required, such as maintainability, flexibility and robustness etc., into templated designs/patterns that help bridge the techno-social divide in software solutions. These include both Design patterns and other Pattern-Oriented software architectures, where the patterns form an embedded embodiment of quality concerns.

The purpose of this research is to apply PODs to an existing computer game, and assess the impact that they have on the game's code quality, focusing primarily on aspects of maintainability, coupling and complexity.

It is thought that applying PODs may help to reduce overall complexity by providing simplifying abstractions, like reducing class hierarchies and reducing the coupling effect that dependencies have on propagating change.

This research provides a practical means to learn, understand and apply PODs to an existing game, and to evaluate and assess the implications of their use on software quality.

This research also provides an opportunity to show-case the benefits, discuss the disadvantages and evaluate the limitations of PODs and how they affect a practical software application.

By evaluating existing designs within the game, and then integrating PODs into those designs, provides an opportunity to learn, practise, and more closely understand the nature of underlying design principles and their effects on quality.

By empirically measuring the effect of incorporating PODs using a comparative before/after quantitative analysis, it is possible to quantify the claims made by design pattern theory to provide a firmer foundation on which to discuss and evaluate the quality characteristics of incorporating them to improve code quality.

This research serves as practical example of applying and using PODs that can be used as a learning tool to inform real world scenarios where insights gained from observation and practical application will afford greater insights into understanding how they may help to improve software design and is of particular use for game developers seeking to

rationalize these effects for their own code, as well as non-game developers, and/or managers looking to evaluate the effects on code quality generally for software development teams.

Furthermore, research has shown that, "..the flexibility provided by design patterns becomes clear when the software is extended" (Wedyan and Abufakher, 2020), and by refactoring existing software to maintain its current behaviour, this research helps to simulate a practical and realistic maintenance task that a developer in a team might perform to reduce technical debt to improve the quality of existing code, and the knowledge gained in the process can be used to inform and improve code design. This is useful during code review, both to discuss the quality implications of using PODs, but also to help in applying and communicating their use in future design work.

The objectives of the research are:

Objective 1: Perform primary research by evaluating an existing game's design, applying appropriate PODs into the game's design, and then produce as an artifact, a working copy of game including its modified source code (incorporating the design changes) such that it can be compared with the control's code, i.e. the unmodified source code.

Objective 2: Perform a static code analysis on the source code before and after to gather the changes in the observed quality metrics

Objective 3: Perform a quantitative and comparative analysis on the effect that incorporating PODs have had on the game's code quality, focusing specifically on its maintainability, class coupling, and code complexity.

Chapter 2

Background

Software design is a, "...cognitively challenging endeavour", where good design choices need to be made which incorporate not only multiple competing design constraints, but also deal with software complexity and design ambiguity, all of which ultimately influence the quality of the solution. (Texas at Arlington and Mangalaraj, 2014)

2.1 Pattern-Oriented Designs (PODs)

PODs are, "...external cognitive resources", which serve as abstract design templates and/or solutions to specific design problems, that have been informed by previous design experience, knowledge and experimentation, to not only include good design-choices (which elegantly incorporate many competing design constraints), but also incorporate techniques that improve the quality of the implementation itself. (Texas at Arlington and Mangalaraj, 2014)

The integration of PODs, along with their associated quality attributes, are likely to simplify the context of particular design problems by making these design-choices explicit structural components that are embedded in the design itself, and research suggests that they are "...expected to enhance software quality". (Ampatzoglou, Frantzeskou and Stamelos, 2012).

However, this research also suggests that while they, "...typically improve certain aspects of software quality", they may weaken others. This is further corroborated by research which shows that, "...contrary to common lore, design patterns do not always impact quality attributes positively" but that that they are useful in solving design problems. (Khomh and Gueheneuc, 2008)

This suggests that their application is a trade-off that requires careful planning and evaluation of quality concerns. It is these quality trade-offs that this research seeks to assess.

2.2 Coupling and change propagation potential

While there appears to be utility in incorporating prior design knowledge and the use of abstractions to simplify designs, studies have shown that the size and the "...scattering degree of patterns have a clear impact on quality" (Wedyan and Abufakher, 2020), which suggests that evaluating the degree of coupling between components may prove to be a useful metric in evaluating the distribution and propagation-potential for code change when using PODs.

In this way, an important aspect of PODs, is the careful manipulation, co-ordination, interactions, and exposure of components and their dependencies.

2.3 Maintainability

Equally important is the influence PODs have on maintainability, a sub-component of quality, as the inevitable evolution of software means that its maintenance is important, and research suggests that the "...maintainability of software strongly depends on the quality of its source code", and that well-structured and easily understandable code are important components of maintainability, and therefore software quality. (Yi Guo *et al.*, 2011). PODs in this

respect, are well positioned to provide this structural basis to improve the "... reuse, flexibility and maintainability" of software designs. (Jaafar *et al.*, 2016)

2.4 Complexity and Abstractions

PODs are typically minimalistic design configurations that define only the essential design elements necessary to define and characterise the utility of their design. Consequently, this enables PODs to be used in more scenarios, as they have less dependencies on situation-dependant detail, and can be considered as templated design abstractions.

Design abstractions by definition, reduce complexity by reducing the detail required to represent them, and thus can be seen as higher-level "... abstract cognitive representations as opposed to lower level implementation code". (Texas at Arlington and Mangalaraj, 2014).

A reduction in design and code complexity is likely to afford a greater ability of software developers to efficiently interpret and understand the nature of the software systems that they need to change.

2.4.1 The Implications of Abstraction

Reducing complexity through abstractions does present challenges however, as details that may inform a better understanding of what they represent or how they should be used, are usually absent in the abstraction itself, as it is not essential to its main function. An inevitable trade-off thus exists between representing simplicity through abstraction to reduce complexity, and conveying utility and understanding, and any difficulty encountered in using PODs may stem from the "... difficulty of using abstract patterns" (Texas at Arlington and Mangalaraj, 2014).

In this way, the metaphors and documentation used to describe design abstractions such as PODs, and an understanding of the underlying reasons for using them are important in helping to provide detail about the usage and context of the design abstraction to mitigate this limitation inherent in all abstractions. Without these, research has shown that the effectiveness of the PODs on source code comprehension and maintainability, and therefore software quality is reduced. (Scanniello *et al.*, 2015)

2.5 Flexibility

Like maintainability, an important software quality is the ability to easily adapt software to changing requirements in order to keep up with changing business needs. This flexibility and adaption potential is arguably improved by designs that are simplified, well-structured and implement good design choices that make designs easy to understand and to change.

PODs are thought to improve flexibility by decoupling collaborating parts of a design so that they operate more independently and thus more flexibly, and by separating roles and responsibilities make designs more adaptable to change such that components in the design can vary more independently. This is likely to reduce the effect of propagating side-effects to co-participating components. This otherwise would restrict the flexibility for example, of introducing new changes.

2.6 Candidate PODs

This research looks to implement the Mediator, Blackboard and Bridge PODs.

Table 2.1: Candidate PODs

Pattern-Oriented design (POD)	Description
Mediator Pattern	The Mediator pattern incorporates a strategy of reducing the number of components explicitly referred to in a design.
Blackboard Pattern	The Blackboard pattern is to help solve non-deterministic problems that do not have a pre-determined sequential solution. This is achieved by co-ordinating the intermediate results of several knowledge sources that contribute specific knowledge towards achieving a solution.
Bridge Pattern	The Bridge pattern decouples an abstraction from its implementation so that they can vary independantly. This allows for multiple implementations to be uniformly

Pattern-Oriented design (POD)	Description
	referred to in the same way, allowing for swapping/changing of implementations.

Chapter 3

Project Evaluation and Specification

3.1 Personal and academic suitability

3.1.1 Personal suitability

As a software developer, PODs are often used to communicate information about designs during code review. Studying the application and implication of PODs is therefore of immediate use to me. While I have used patterns in the past such as the Factory, Builder, Observer and Iteration patterns, I've had little exposure to implementing the Mediator, Blackboard and Bridge patterns. The application of the Blackboard pattern has specific implications for potentially improving the game AI. In this way, implementing these and understanding their utility is important to my own personal development.

Furthermore, during previous study, principles for design have been discussed such as Law of Demeter and principles of encapsulation and loose coupling however these have not been applied to any specific context and this project provides an opportunity to do so.

The software under consideration is a 2D platformer game previously created, and I am thus familiar with it, and an improvement in quality would be of direct benefit me in the context of my own design and developer experience.

In terms of ethical implications for this research, no 3rd party code used in the game will be modified, only that which was developed by myself will be subject to modification, the game itself is not associated with any company and is entirely open source and freely available online at https://github.com/stumathews/MonoMazer.

3.1.2 Academic suitability

Research has shown that best practises in Software Development are mostly influenced by "... anecdotal evidence and unquestioning faith in claims made by leading consultants", and, "... rigorous empirical work on design patterns is limited", which show deficiencies in previous studies that, "... provide no insight into the viability of design patterns in a real design context" (Texas at Arlington and Mangalaraj, 2014).

In this way, this research hopes to provide an empirical evaluation into the effects on quality within a comparatively complex software application.

Prior research has also shown that, "... generally there is no consensus on the effect of design patterns on software quality" (Wedyan and Abufakher, 2020), and a practical and empirical evaluation of the effects of PODs on quality in the context of a working game is hoped to help contribute toward reaching a practical consensus about the utility of PODs on code quality.

3.2 Feasibility

An existing and working game and its source are available and is well understood being created prior to this research. In this way, there is no need to build the software which could add unnecessary time and effort and detract from the primary purpose which is to evaluate the effect of PODs on code quality in existing software.

The process of measuring the quality markers in the code can be assessed using Visual Studio's built-in Code Analysis feature and this will reduce the need to invest in building or setting up 3rd party static code analysis tools like Sonar Cube for example. This makes it more feasible to use within the time constrains of this research.

While a large portion of the research requires making code changes in the context of an existing game, and then following up with a comparative analysis of the effects of the changes, the amount of the changes required to incorporate POD can be reduced by limiting to subset of possible PODs which may be integrated. The aim is to incorporate at 3 PODs into the game before performing the final static code and quantitative analysis.

The scope is to assess the quality of the game after implementing the PODs. This can be evaluated by comparing the effect on the quality observed in types/classes in the game, as these usually corresponds to specific roles in the game. In this way it is not necessary to improve all aspects of the game, as this would be time prohibitive, but to evaluate the effect of PODs on at least one subsystem of the game's operation, such as collision-detection, AI, graphics etc., and this would be based on the appropriateness of the integration of the POD to be used effectively in the candidate subsystem.

3.3 Risk

Table 3.1: Project Risks

Risk	Mitigation	Likelihood (1-5)	Impact (1-5)	Severity (1-12)
Introducing too many PODs (consuming too much time)	Restrict to 3 candidates, reduce to 2-1 if it becomes to difficult/time consuming	3	3	9
Too many evaluation criteria to use in comparative analysis	Restrict evaluation to Maintainability and Class Coupling	3	4	12
Backboard is too complex to implement	Select Factory Pattern or alternatives	1	3	3
Design patterns aren't easily integratable into project	Implement Demeter's Law solely to improve effect of structural coupling.	1	2	2
Some structural code metrics does not produce any appreciable-ble results	Eliminate from analysis and note in results	1	5	5
Game too complex to use as subject for refactoring	Use prototypical fragments of contrived software design examples and refactor.	2	3	6

Note: Severity = Likelihood x Impact

3.4 Project specification

3.4.1 Title

Applying and Evaluating Pattern-Oriented Designs In Improving Code Quality In Complex Software

3.4.2 Schedule

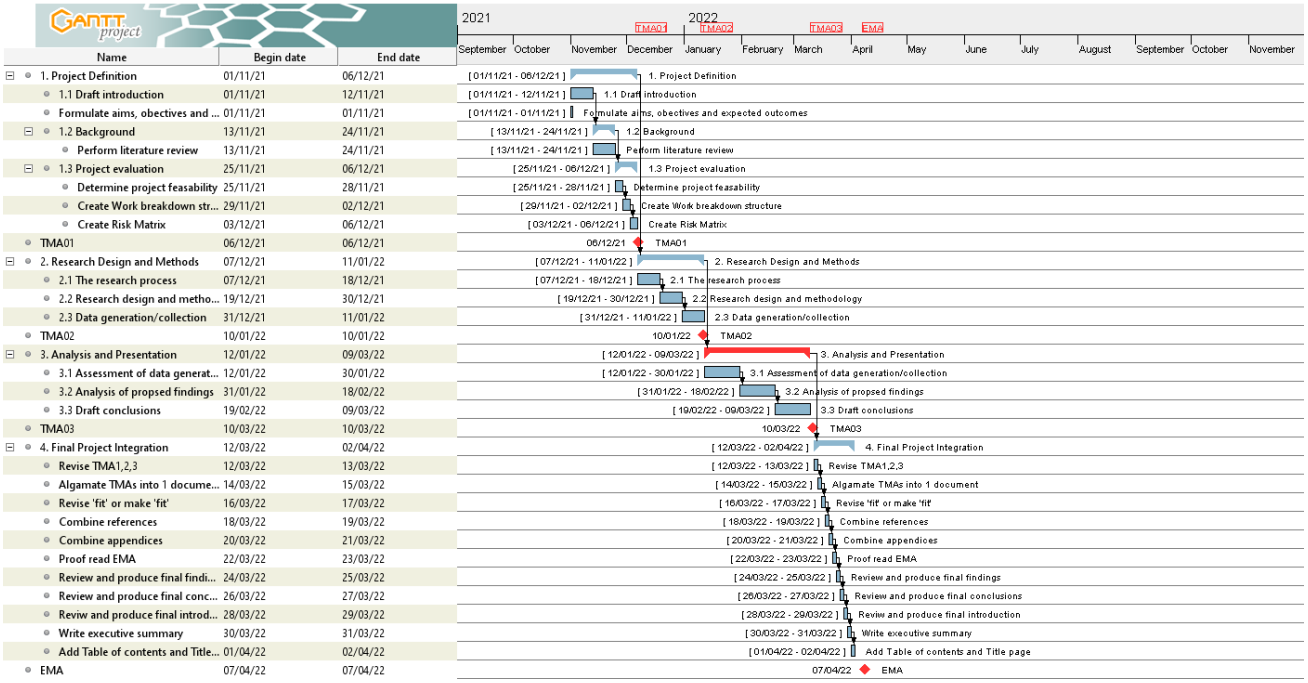

Figure 3.1: Project work break down structure

Chapter 4

Research Process

This research is primarily focused on evaluating the impact that PODs have on code maintainability, class coupling (dependencies) and complexity within the game. These quality measures will form part a comparative analysis of sofware code quality.

By using an approach that requires practical application, i.e, integration, allows for incorporating both dynamic learning and in-situ experience both from a pattern-oriented design educational standpoint and also, from a software development perspective (integration).

Using a comparative before and after approach, where changes to the codebase are applied to an initial state, and where performing an associated static code analysis to capture the quality markers before and after the PODs are introduced, provides a robust means to distinguishes between no-effect i.e., the control state (before), and that which results from performing the POD integration work (after). This also provides a theoretical basis on which to attribute the cause and nature of quality changes that occur as a direct result of the POD integration process.

This approach however does have its limitations. A risk inherent during POD integration, is introducing factors that are unrelated to the POD design, but which facilitate its implementation, such as refactoring or re-structuring adjacent supporting code such that it influences the quality beyond that which is attributed to the PODs themselves. While a degree of error in this regard is possible, perhaps even inevitable, care is taken to limit performing extraneous re-engineering.

Additionally, to conduct the research within its available timeline, only a limited portion of the game will be modified by introducing PODs, which means that the effect on the overall quality in the game will be partial. That said, any change, however small, is considered useful, not only because it incrementally contributes to the overall view of the game's quality, but crucially because it provides a useful indication of a future trajectory, if a wider adoption of PODs was used moving forward.

Furthermore, the quality metrics used as a result of the static code analysis, are only as useful as their ability to capture, measure and accurately represent maintainability, complexity and class coupling from static code. They are less able to capture other social factors such as a programmer's perception of maintainability for example, however as PODs are introduced at a code level, structural code changes that are known to affect maintainability, complexity and class coupling are considered relevant.

Chapter 5

Research design and methodology

The research paradigm used incorporates a positivist viewpoint (Burton-Jones and Lee, 2017), i.e, qualitative, where cause-and-effect are modelled by before-and-after states and comparative analysis. In addition, this is driven by primary research through experimental manipulation of the codebase to incorporate new PODs, and assessing the result and impact of those changes.

By extracting quality metrics from the codebase through a latent process, i.e static code analysis (Louridas, 2006), both before and after the PODs are applied, means that these metrics, relevant to maintainability, complexity and class coupling, can be easily compared. The differences and rationale for their change can then be analysed and discussed.

5.1 Objective 1: Re-engineering MonoMazer using Pattern-Oriented Designs

The overall approach is to firstly identify the suitability of incorporating candidate PODs into the game's existing architecture, then apply these PODs within the game's design, performing a comparative analysis of the impact of these changes, and lastly, evaluate the implications thereof on the quality of the game.

The overall approach is illustrated below:

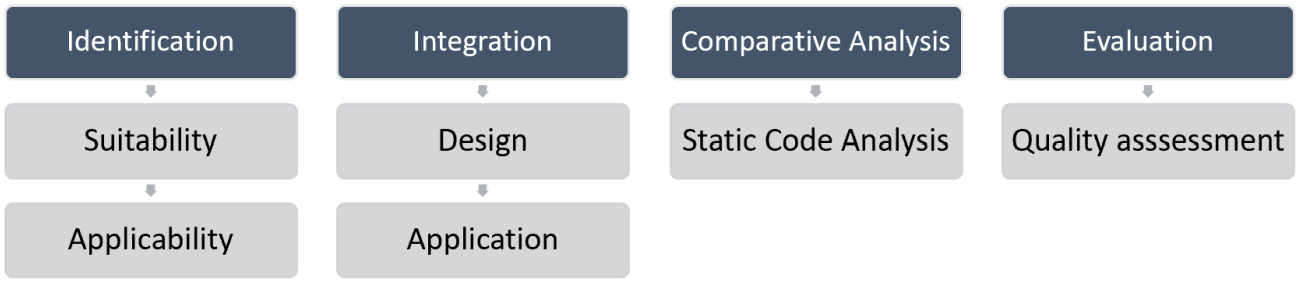

Figure 5.1: Overall approach (left-to-right)

An alternative approach might have been to aggregate a social viewpoint on whether the source code, with the PODs included, is perceived as more or less maintainable after the refactoring has taken place. This could be achieved by designing and collecting responses from software engineers who have assessed the changes. While this was considered, it was felt that re-engineering the game code, designing a suitable format to collect responses, and finding a representative sample of software engineers, ideally with an understanding of game design, would be time prohibitive.

In contrast, it was felt that using static code analysis (SCA) would be more effective, not only as it would likely achieve greater consistency in terms of how it determined quality, but also would likely be more accurate, more repeatable, less biased and argubaly less error prone, particularly as, "...code review teams also need practise

(perhaps even training) to do the work well" (Louridas, 2006). Additionally, SCA is more suited to compiling and collecting code-level details than a person might be, for example in attempting to determine the number of dependencies in multiple classes. In this way, SCA is felt to be most suited to the objectives of this research.

5.2 Objective 2: Perform a Static Code Analysis

To evaluate the effect that PODs have had on the quality of the codebase, a static code analysis will be run on the source code both before and after the POD implementation.

The before branch is named the 'alpha' branch, and represents the first and initial state of the codebase, while the subsequent modifications and introduction of PODs are made to the 'beta' branch. This naming convention represents the transition from the initial code state A to B (Alpha to Beta).

Microsoft's Code Analyser (MSCA) is run an both branches of the codebase and the results are collected and compared. This quantitative comparison of the resultant static code analysis will form the primary basis for assessing the effects that the PODs have had on the codebase and is the focus of the main analysis presented later.

Three primary measures are to be used to assess the quality of the codebase, namely Maintainability Index and Class Coupling, and Cyclomatic Complexity.

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." (Mikejo5000, no date)

The following is an 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 metric will be used to assess the effect on maintainability before and after the introduction of PODs.

Class Coupling is an indication of how many classes one class uses, and according to Microsoft, it "...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." (Mikejo5000, no date). This metric will be used to determine the risk that code has in propogating change.

Cyclomatic Complexity is a code quality metric which, ".. assumes that overly complex programs are difficult to maintain and more likely to contain errors", and as such it determines complexity as a function of the number of linear pathways through the code. It suggests that those that have a Cyclomatic Complexity value greater than a certain threshold (typically between 10-15) are complex and should be split into smaller functions, effectively reducing its complexity. Cyclomatic Complexity will be used to evaluate the complexity of the resulting codebase.

5.3 Objective 3: Analysing Code Quality

The primary effects on maintainability, complexity and degree of coupling within the game will be measured and assessed quantitatively, both at a local and global level.

A local-level analysis refers to the influence of quality within types that implement the PODs, i.e have been refactored, while a global, refers to the overall effect that these changes have on the game as a whole and represents an aggregate view of overall quality.

Only the files within the MonoMazerPlatformer namespace are analysed as this represents the main source code that is subject to the changes introduced in this research.

Chapter 6

Data Generation/Collection Methods

By way of re-engineering the existing game codebase, subsequent changes to the codebase, as a result of introducing PODs, will be used to generate the data. SCA (Static Code Analysis) will be used to codify software code quality through collecting Maintainability, Cyclomatic Complexity and Class Coupling metrics generated from the modified source code.

The following section outlines how these changes will be introduced. A later chapter (Analysis) will assess the impact these changes have had on the resulting codebase.

6.1 Incorporating the Mediator Pattern

The Mediator POD is designed to reduce class coupling by restricting direct access to functionality such that it, "... controls how a set of objects interact. Loose coupling between colleague objects is achieved by having college objects communicate with the Mediator, rather than one another" (Vlissides-John *et al.*, 1995).

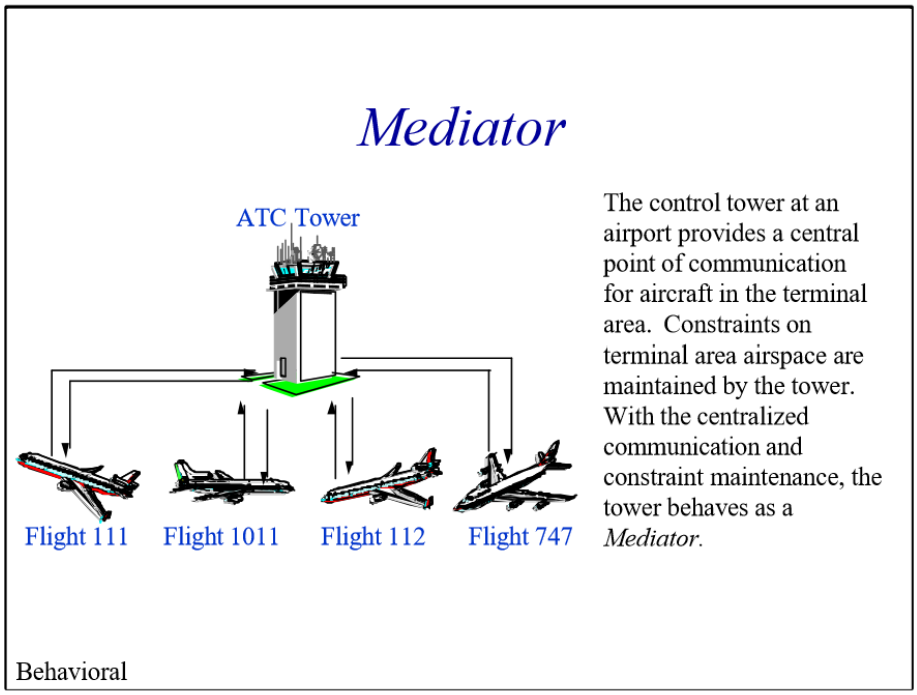

Figure 6.1: Example illustrating the Mediator pattern (Vlissides-John *et al.* (1995))

A systematic implementation approach will be used, as illustrated below.

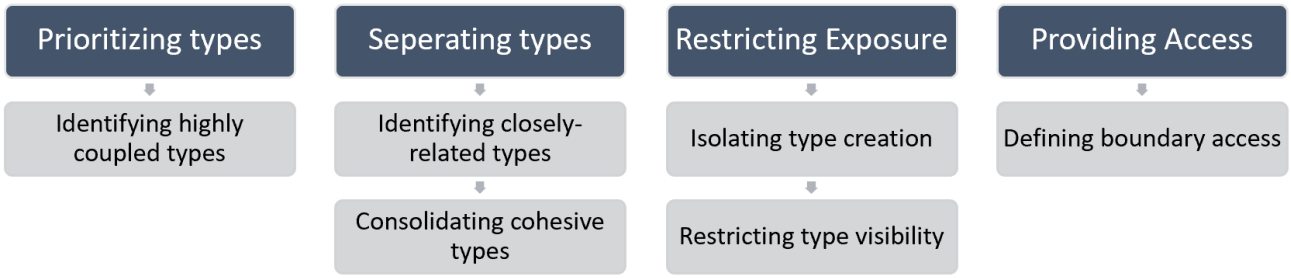

Figure 6.2: Implementing the Mediator pattern (left-to-right)

Prioritizing types, refers to the need to determine which types are most likely to be positively affected by the Mediator POD, i.e., those that are highly coupled. To determine this, a static code analysis is run on the codebase before the Mediator POD is applied. Types with a high Class Coupling are then re-designed to incorporate a Mediator.

Separating types, refers to the need to identify closely related types that are similar in, or are directly supportive of surrounding types. This is done with the view ultimately to achieve cohesive sets of functionalities which are loosely coupled to other less similar functionality.

Restricting exposure, refers to moving the previously identified cohesive types into a Mediator object and hiding or encapsulating access to those enclosed cohesive types so that they can change independently of users of the enclosing Mediator. This allows the cohesive nature of the functionality enclosed in the mediator the remain isolated, and changes within them are restricted to the Mediator and so do not have a large impact on the user's of the Mediator. This is likely to improve maintainability. The identified cohesive types need to be created only within the scope of the Mediator class (*Isolating type creation*), and the Mediator's interface should not directly refer to the internal cohesive types in its external interface definition.

Providing access and Defining boundary access, refers to establishing the collaborative external interface that the Mediator presents, and which indirectly provides access to the underlying functionality provided by the encapsulated types. Special care must be taken not to refer directly to the encapsulated cohesive functionality within its interface.

6.2 Incorporating the Backboard Pattern

The Blackboard POD defines and brings together different fields of information, each which is managed by an expert in that field. A co-ordinated emergence of expert information is then used to uniformly establish if a certain goal has or has not been met, particularly when, "...no deterministic solution strategy is known" (Vlissides-John *et al.*, 1995).

The goals of the system are defined by establishing what combination of information (from the various experts) are required and need to be in place to evaluate dependant goals.

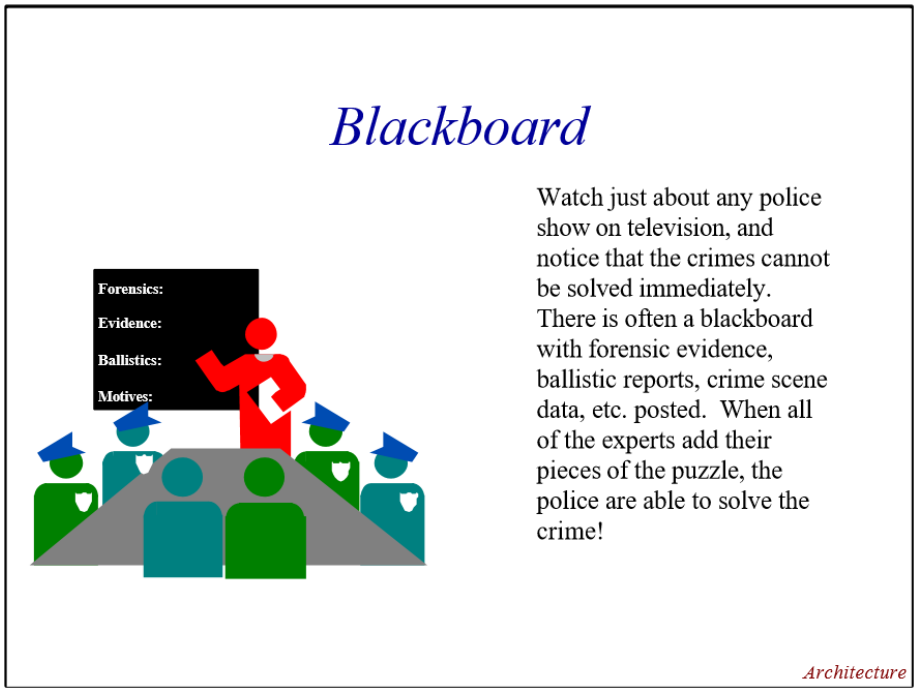

Figure 6.3: Example illustrating the Backboard pattern (Vlissides-John *et al.* (1995))

An approach to implement the Blackboard POD is illustrated below.

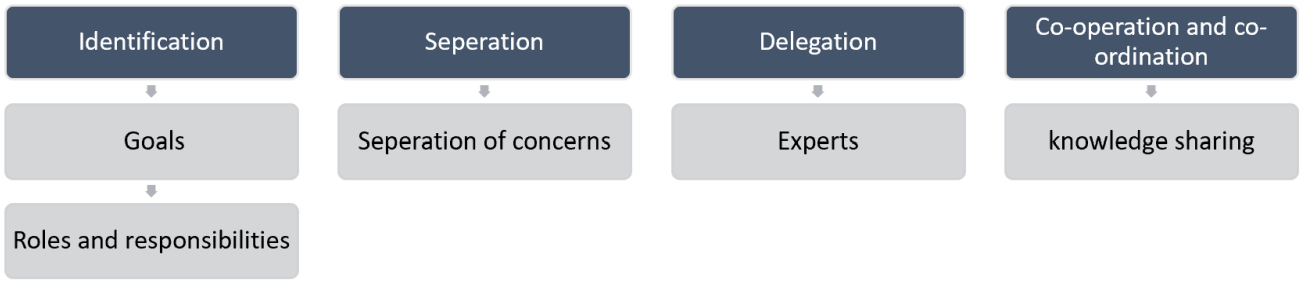

Figure 6.4: Implementing the Blackboard pattern (left-to-right)

Identification, refers to identifying what combination of information is required to determine if a goal is met. Typically, the dependant information is sourced from different subsystems, which when evaluated together help determine if the goal has been met.

A simple analogy is that a goal is like an equation, where the evaluation of the equation yields the goal. The constituents that make up the equation are pieces of information or dependencies sourced from different places, which when combined can be evaluated to determine if the goal is reached.

This step identifies what goals need to be established, and what information is required to define the goals, and which areas of the subsystem are currently responsible for that information.

Separation of concerns (Stutterheim, Achten and Plasmeijer, 2018), refers to separating the functionality that has been previously identified as being responsible for generating the required information, such that this functionality can then be managed separately by a dedicated and centralised 'expert' component.

Experts, refers to the phase which moves the previously identified functionality into 'expert' types, which will use this functionality to generate information required by goals defined on the Blackboard. Each Expert is responsible for its underlying field of functionality.

Co-Operation and co-ordination, refers to bringing together various experts to contribute and share the information

which they are responsible for, and which the goal equations are dependent on, so that the goal can be evaluated as a unification of all these constituents.

This also requires co-ordinating the invocation of each expert, such that each can interrogate its underlying functionality and provide corresponding information back to the Blackboard.

The goal then needs to be evaluated, and any game functionality that depends on the goal being reached (or not) needs to be executed.

6.3 Incorporating the Bridge Pattern

The Bridge POD mandates the division and separation of a type's external interface from its implementation. Specifically, it, "...decouples an abstraction from its implementation, so that the two can vary independently" (Vlissides-John *et al.*, 1995).

Consequently, this means that the implementation of the type can vary independently from the interface that represents it. This promotes loose coupling, by reducing the dependency that existing code has on the implementation, and instead coupling it to the corresponding interface, which is a lesser form of coupling than depending on the actual implementation.

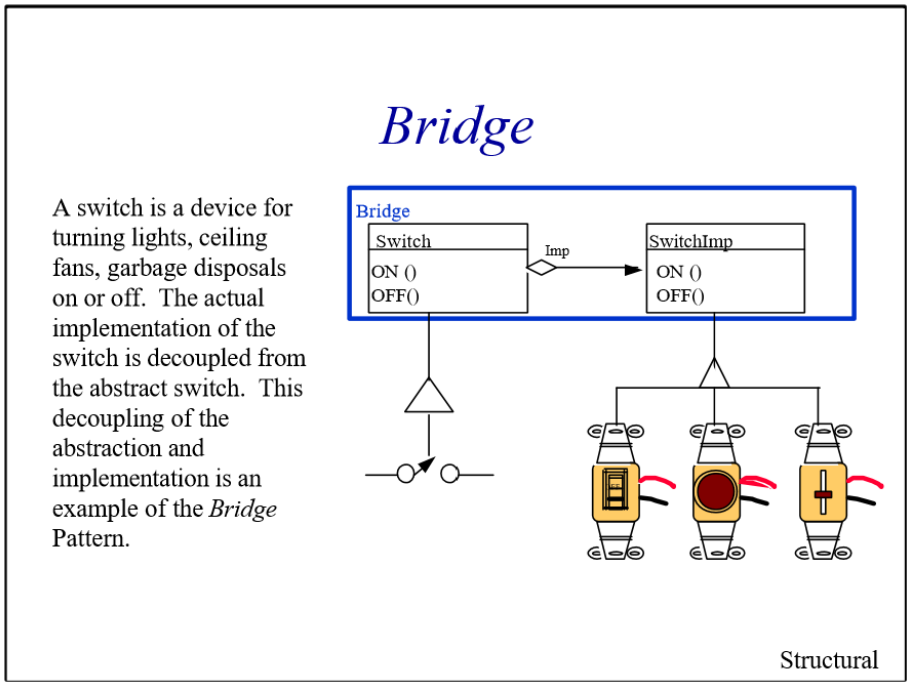

Figure 6.5: Example illustrating the Bridge pattern (Vlissides-John *et al.* (1995))

A four-step approach will be used to implement the Bridge POD, as illustrated below.

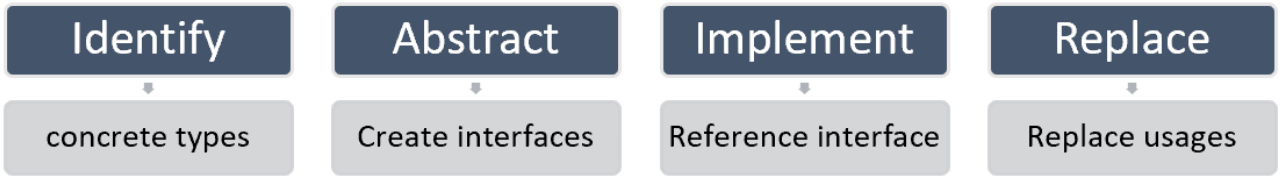

Figure 6.6: Implementing the Bridge pattern (left-to-right)

Identify, refers to identifying the types that should be abstracted from their concrete implementations. Any type is

suitable; however, it makes more sense to start with types that are likely to change often as these will benefit the most from separating their implementations, which can change over time, from their abstractions, i.e interfaces, which ideally should not need to.

Abstract, refers to defining an interface for the currently coupled implementation, and having that implementation present that interface. This interface will then be used instead as a proxy to the underlying implementation moving forward.

To define an interface for the candidate implementation, requires incorporating all the methods that are currently being used to refer to the current implementation. This is so that when replacing references to the actual implementation with the interface type instead, existing code will continue to work as these methods will also be available on the interface.

Implement, refers to refactoring usages that reference properties of the implementation type to instead to reference the associated properties on its interface type. This can be achieved by looking for all references to the implementation type and changing the type to its interface type instead.

The process is continued until there are no more direct references to the implementation type. The only direct references to the implementation should be in the code that associates the interface with the implementation.

The next section briefly assesses the results of having carried out the POD implementation strategies outlined in this section, followed by analysing the effects it has had on the underlying codebase.

Chapter 7

Assessment Data Collection/Generation and any Issues

7.1 Incorporating the Mediator POD

The *InfrastructureMediator* is now responsible for providing an interface that will draw and load assets as well as manage the game's AI (more implementation details can be found in Appendix B). The *UIMediator* has pulled together all the UI functionality under a single cohesive class, and the *GameMediator* encapsulates access to the underlying game class, *Mazer*. The *GameWorld* instead of referring directly to Game event types now goes through the *EventMediator* to access centralized game event types.

By running the SCA on the unmodified code (Alpha branch) and inspecting the outliers, *Prioritizing types by identifying highly coupled types* was successfully achieved, however *separating types by identifying closely related types* was more difficult as the extent of the 'relatedness' needed to be evaluated to determine what would be incorporated into a mediator and what should be left out. This was generally reasoned by deciding how similar or dependent the classes are on each other.

By moving type instantiations into the mediator class, *Restricting type exposure* was successfully achieved, however issues arose as existing code that previously required access directly to those types were now broken, and now required a new means to access that functionality.

Determining how best to provide access to this mediated functionality was not trivial as it required understanding exactly what the existing code did, what it required from the now-inaccessible type, and then determining how to make this accessible through the mediator's external interface, without exposing a direct coupling to the underlying types that provide that functionality. This inevitably means that this task is more suited to an experienced developer.

7.2 Incorporating the Blackboard POD

The *GameWorldBlackboard* now represents a combination of relevant information pertinent to game-wide goals, such as determining if the level is complete, whether the player has been sighted by an enemy or not, or how many pickups are left in the game. It calls upon the expertise of the *LevelExpert* to source and provide this information from the Level subsystem. The coordination of expertise is controlled by the *GameWorldBlackboardController*.

The *CollidingExpert* is now responsible for bringing information about game collisions to the *MovingStateBlackboard* and path finding expertise is now delegated to the the *PathFindingExpert*.

While goals and dependent goal information already existed in the game, it did not exist separately and was often intertwined within various areas of the game code. Through the *separation* phase, disentangling this was required and presented problems when refactored imprecisely, as they game would stop working. This did however make it easy to identify issues.

Identifying goals, moving the goal-evaluation code to the blackboard, and relocating and refactoring the logic of retrieving goal-dependent information into experts was challenging as it required diligent refactoring, and

an understanding of the various subsystems housing the required information. This did however clearly define responsibilities within the code.

A helpful consequence of using the POD within the game, was the ease of which the *co-operation and co-ordination* stage could be integrated into the game-loop which dictated a timely coordinated invocation of the Blackboard's various experts on each frame.

7.3 Incorporating the Bridge POD

The refactoring required to implement this POD was the most time consuming, as the objects identified as part of the *identification* phase, namely Room and GameObject were used throughout the codebase, and thus a large surface area needed to be modified to refer to the associated interface types instead, namely IRoom and IGameObject. This does mean that there is a larger potential for introducing defects throughout this process.

There were instances however where using interfaces was not possible. Specifically, where objects were loaded from files, because deserialization to an interface is not allowed and required a specific implementation type. A solution to this was not found, and instead data for types loaded from files were not 'bridged' in this way.

Chapter 8

Analysis/Findings

8.1 Tools and techniques

The primary tool used for evaluating the results was Jupyter notebook, along with numerical libraries, NumPy and Pandas. Together these were used to interrogate and compare the SCA metric data produced from the codebase both before and after the introduction of the PODs. The data visualization library matplotlib was used to interpret and graphically present the data.

Jupyter was used because it allowed for incremental analysis, as generated results were easily observable and saved in-place. These could be easily re-visited and analysis routines could interrogate the data in a consistent fashion. This was particularly helpful when the analysis routines changed during the project, and allowed for the re-generation of results. Equally useful was that the associated notebook could be stored in source control (GitHub).

8.2 Analysis Methodology

To prepare the data for analysis, the results from the before and after SCA were imported into a Jupyter notebook. The results were filtered to exclude any data which were not subject to analysis such as 3rd party namespaces and unused metrics.

A dataset was assembled that contains a comparison of each metric both before and after refactoring (for each game type/class), and this was used to construct an analysis graph for each metric gathered during the SCA. The new data, i.e., the POD types, which contained only 'after' data, was appended to this dataset.

The analysis graph plots the game's types on the x-axis and the values for its corresponding metric on the y-axis (See 'Global Class Coupling in MonoMazer' figure below)

The difference between the before and after values is illustrated by a colored vertical line between the quality metric-values for each type, where a green segment indicates an increase in the metric, and a red line indicates a decrease.

The analysis graph is separated vertically into two main shaded areas. The purple area represents the initial non-PODs (existing types), i.e. types which existed before adding the candidate PODs, while the light-green area represents the newly added POD types.

Within these regions, regional tendency lines are plotted as horizontal dotted lines and represents only the types that fall within that region. These represent the aggregate effect (mean) for types in the region for the specific for the quality metric in question.

8.2.1 Data Coding

For existing types, there are two *Existing Tendency* lines (before and after) representing the before and after tendencies for existing types. Existing types represent those that will be refactored to accomodate the PODs or will be influenced by them. For new POD types there is a single *Added Tendency* line.

The *Global Tendency* is a solid overall line which extends across both shaded regions and includes all types and represents the net-effect (mean) of introducing the PODs into the game.

The global tendency uses the entire population of available game types and normalizes and evenly distributes the effects throughout the population, which includes all types, changed and unchanged and includes POD types, i.e., everything that might influence the overall quality measure. It is effectively the overall mean of the respective metric per game type, after incorporating the PODs, across the entire game.

A single horizontal grey-filled region represents the region in which data falls within +/-1 standard deviation from the global tendency and represents the expected normal range of that data which also makes it easier to identify unexpected and unusual results.

The values for each tendency line are included in brackets within the legend of the graph.

A supplementary 'change' table for each metric is used to represents only the types that have actually changed, their before vs after values, differential and the overall Change Tendency, which is a mean across only these changed/refactored types and excludes PODs. It is designed to represents the quality tendencies within types that been refactored to utilize the PODs.

Change tendency is restricted to a limited localized subset of the game, i.e., what has been refactored, and is distinct from the global tendency, which is more inclusive and includes the PODs types and types that have not been refactored (unchanged).

8.3 Class Coupling

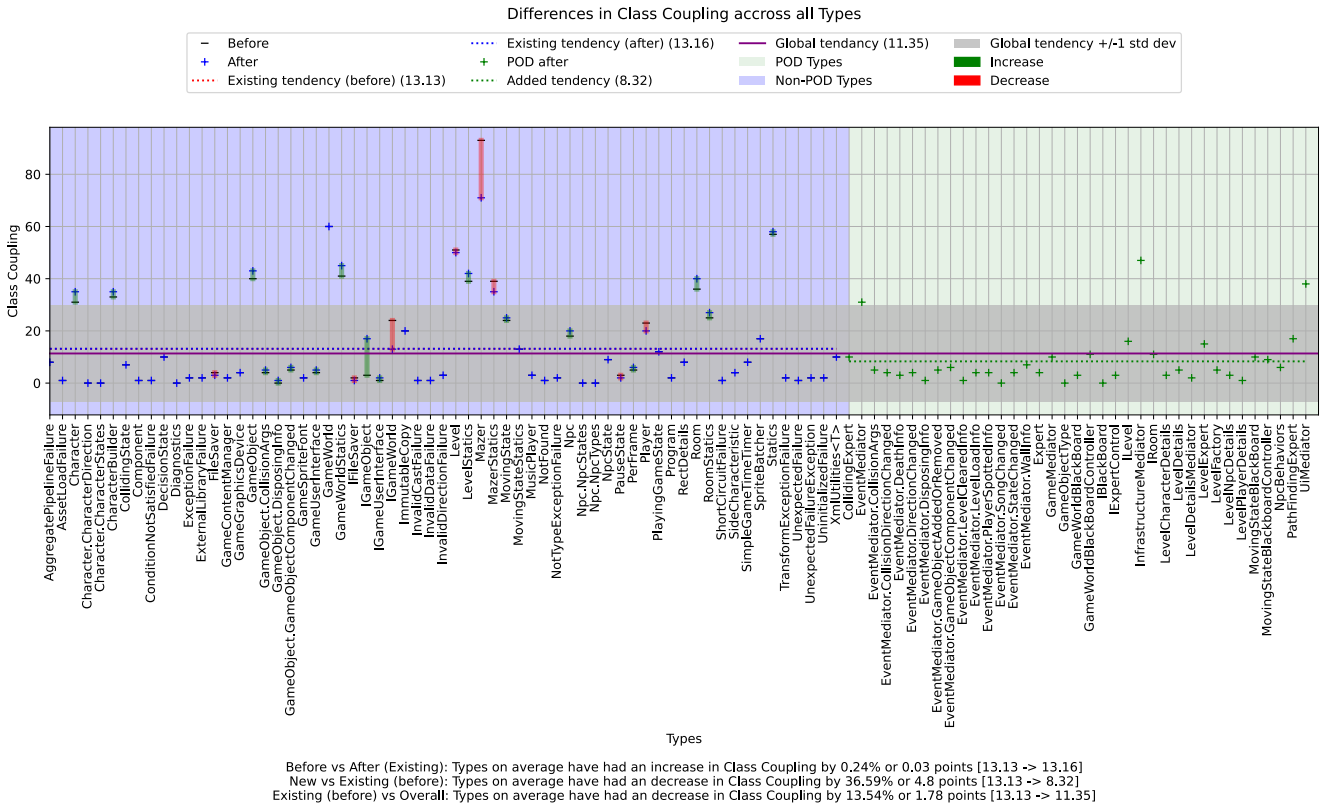

Figure 8.1: Global Class Coupling in MonoMazer

To investigate how the game's overall Class Couping has been affected by the PODs, the global Class Coupilng tendency is analyzed.

The game's global Class Coupling tendency has decreased, when compared to the existing tendency (before), by an average of 13.5% or 1.78 points per type [13.13 -> 11.35].

This result suggests that the game overall has generally benefited from the inclusion of PODs and they have helped reduce the average Class Coupling per type within the game.

To understand how the Class Coupling in existing parts of the game have been affected by the PODs, the existing Class Coupling tendencies (before and after) are analyzed.

The influence that PODs generally have on existing types shows that, on average, they have increased in Class Coupling by 0.24% or 0.03 points [13.13 -> 13.16] per type after refactoring.

This increase, albeit fractional, suggests that to facilitate the introduction of the PODs, existing types generally become slightly more coupled as a result, likely to a result of coupling to the new PODs.

POD types generally are, on average, 36% or 4.8 point less coupled per type [13.13 -> 8.32] than existing types before the refactoring took place, suggesting that PODs are generally of a higher quality than non-PODs.

It is evident that the *InfrastructureMediator*, *EventMediator* and *UIMediator* are unusually highly coupled, falling outside 1 standard deviation point of the norm. This is likely the localized effect of the increased cohesive coupling within the mediators to the mediated types, as opposed to having less cohesion and more distributed coupling throughout the codebase.

To investigate the effect of Class Coupling in types that have been refactored, i.e., excluding new POD types, the change tendency is analyzed next:

	Before	After	Diff	% Diff
Character	31	35	4	12.9
CharacterBuilder	33	35	2	6.1
FileSaver	4	3	-1	-25.0
GameObject	40	43	3	7.5
GameObject.CollisionArgs	4	5	1	25.0
GameObject.DisposingInfo	0	1	1	0
GameObject.GameObjectComponentChanged	5	6	1	20.0
GameUserInterface	4	5	1	25.0
GameWorldStatics	41	45	4	9.8
IFileSaver	2	1	-1	-50.0
IGameObject	3	17	14	466.7
IGameUserInterface	1	2	1	100.0
IGameWorld	24	13	-11	-45.8
Level	51	50	-1	-2.0
LevelStatics	39	42	3	7.7
Mazer	93	71	-22	-23.7
MazerStatics	39	35	-4	-10.3
MovingState	24	25	1	4.2
Npc	18	20	2	11.1
PauseState	3	2	-1	-33.3
PerFrame	5	6	1	20.0
Player	23	20	-3	-13.0
Room	36	40	4	11.1
RoomStatics	25	27	2	8.0
Statics	57	58	1	1.8
Median	24.0	20.0	1.0	7.5
Mean	24.2	24.3	0.1	21.3
Mean Std dev	22.2	19.8	6.1	95.2

Figure 8.2: Class Coupling in refactored types

The change tendency, on average, sees a 21.3% increase in coupling, suggesting that like the increase in Class Coupling in existing tendency (by 0.24%), that among refactored types, they become more coupled.

The amplified increase in refactored types is due to the exclusion of unchanged types and the influence of outliers. To account for the outliers, we discount the mean in favor of the median, which shows that median Class Coupling in refactored types increases by 7.5%.

8.4 Maintainability Index

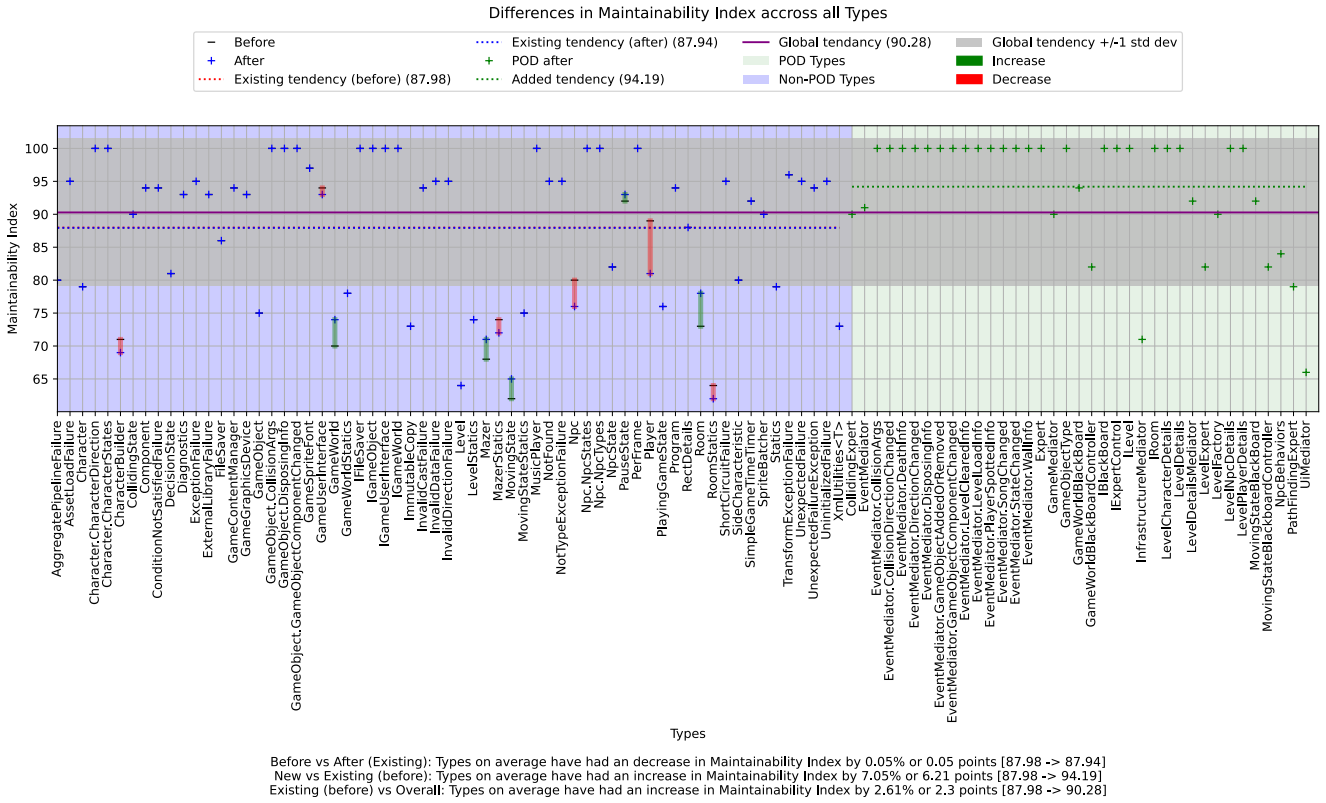

Figure 8.3: Global Maintainability Index

To investigate how the game's overall maintainability has been affected by the PODs, the global Maintainability Index tendency is analyzed.

The game's global Maintainability Index tendency has improved, when compared to the existing tendency before, by an average of 2.61% or 2.3 points [87.98 -> 90.28] per type.

This result suggests that overall, the game has become slightly more maintainable as a result of incorporating the PODs.

To understand how the Maintainability in existing parts of the game have been affected by the PODs, the existing maintainability index tendencies (before and after) are analyzed. This analysis shows that the existing tendency (after), per type, on average has has decreased in maintainability by 0.05% or 0.05 points [7.98 -> 87.94].

This result suggests that generally existing types have become slightly less maintainable, and that it is the presence of the PODs that improves maintainability overall.

In comparison to existing tendency (before), added tendency is, on average, 7.05% or 6.21 points more maintainable, per type, that existing types [87.98 -> 94.19]. This indicates that the new PODs types are generally more maintainable than existing types.

To investigate the effect of maintainability in types that have been refactored, the change tendency is analyzed:

	Before	After	Diff	% Diff
CharacterBuilder	71	69	-2	-2.8
GameUserInterface	94	93	-1	-1.1
GameWorld	70	74	4	5.7
Mazer	68	71	3	4.4
MazerStatics	74	72	-2	-2.7
MovingState	62	65	3	4.8
Npc	80	76	-4	-5.0
PauseState	92	93	1	1.1
Player	89	81	-8	-9.0
Room	73	78	5	6.8
RoomStatics	64	62	-2	-3.1
Median	73.0	74.0	-1.0	-1.1
Mean	76.1	75.8	-0.3	-0.1
Mean Std dev	10.6	9.6	3.7	4.8

Figure 8.4: Maintainability Index in refactored types

The change tendency shows that, on average, refactored types see a fractional median reduction by -1.1%. This indicates that in refactored types become slightly less maintainable.

8.5 Cyclomatic Complexity

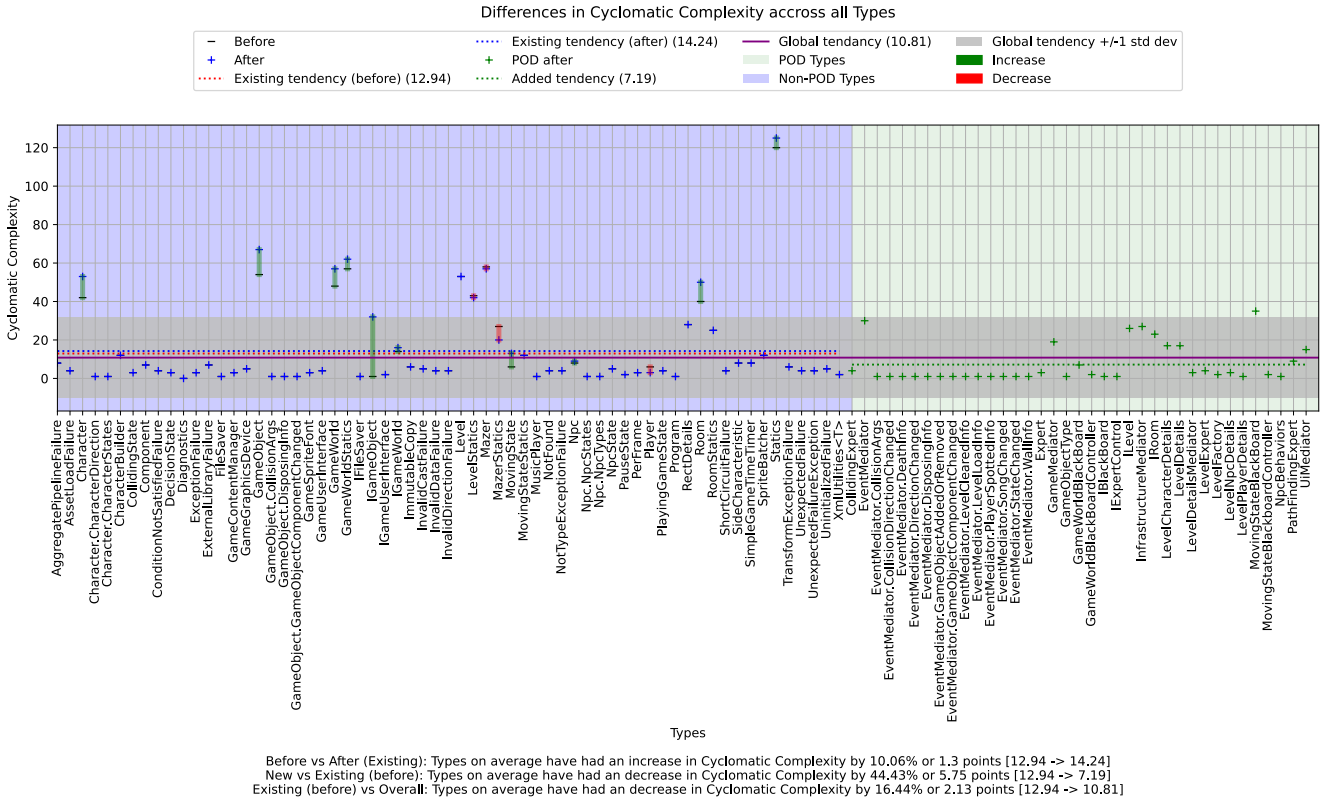

Figure 8.5: Global Cyclomatic Complexity

To investigate how the game's complexity overall has been affected by the PODs, the global complexity tendency is analyzed.

The game's global complexity tendency has decreased, when compared to the existing tendency (before), on average, by 16.44% or 2.13 points (12.94 -> 10.81) per type, suggesting that overall the game has become less complex and has generally benefited from the inclusion of PODs.

To understand how the complexity in existing parts of the game have been affected by the PODs, the existing complexity tendencies (before and after) are analyzed.

The existing tendency (after), on average, has increased by 10.06% or 1.3 points (12.94 -> 14.24) per type, suggesting that existing types tend to become approximately 10% more complex after refactoring.

The added tendency is, on average, 44.43% or 5.75 point less complex per type (12.94 -> 7.19). This suggests that PODs types are generally less complex than the existing types.

To investigate the effect of complexity in types that have been refactored, the change tendency is also analyzed:

	Before	After	Diff	% Diff
Character	42	53	11	26.2
GameObject	54	67	13	24.1
GameWorld	48	57	9	18.8
GameWorldStatics	57	62	5	8.8
IGameObject	1	32	31	3100.0
IGameWorld	14	16	2	14.3
LevelStatics	43	42	-1	-2.3
Mazer	58	57	-1	-1.7
MazerStatics	27	20	-7	-25.9
MovingState	6	13	7	116.7
Npc	8	9	1	12.5
Player	6	3	-3	-50.0
Room	40	50	10	25.0
Statics	120	125	5	4.2
Median	41.0	46.0	5.0	13.4
Mean	37.4	43.3	5.9	233.6
Mean Std dev	30.3	30.7	8.9	795.8

Figure 8.6: Cyclomatic Complexity in refactored types

The change tendency shows an increase in complexity, on average, by 223.6%, suggesting that among refactored types, they generally become more complex because of their changes.

Looking at the adjacent values, this is likely disrupted by the outliers, including IGameObject at 3100%, and as such we discount the mean tendency in favor of the median, which eliminates the outliers. This revised tendency thus suggests, that the among refactored types, the median increases by 13.4%.

Chapter 9

Conclusions

9.1 Coupling and Propagation Potential of PODs

The global tendency for Class Coupling shows that the average game type, as a result of incorporating PODs, has become on average, 13.5% less coupled.

This improvement is significant, because as change occurs in the codebase, as a result of maintenance activity, it now has less potential to disrupt or propagate the effects to other adjacent code.

Furthermore, "... it is generally held that, of two comparable systems, the one with lower coupling (i.e., higher isolation) and higher cohesion, is (or should be) of better quality" (Moser and Misic, 1997)

Also, research has shown that by having a reduced surface area which is defined by, "... minimizing object dependencies", leads to better testability and maintainability. (Curtis, Sappidi and Szynkarski, 2012)

In this way, the PODs have reduced and localized the distribution of effects.

9.2 Complexity

The global tendency for Cyclomatic Complexity shows that the average game type is generally 16.44% less complex.

This reduction in complexity has wide ranging implications for the codebase, as research has shown that, "... complex code is hard to write correctly and hard to maintain, leading to more faults" (Jbara, Matan and Feitelson, 2014).

The significance that complexity plays in defect management is confirmed by (Schroeder, 1999), who shows that, "... complex code contained more than 40 percent of the problems", and that simpler code housed only about 12% of all defects.

In this way, not only is complexity a critical aspect of maintainability, but by reducing complexity, the candidate PODs have contributed to improving aspects of code maintainability, including code comprehension, and reducing the potential for complex code to house defects.

The potential improvements in understandability of code indicates that, "Transferability - the ease with which a new team can understand the application and quickly become productive" (Curtis, Sappidi and Szynkarski, 2012), is likely to also improve. As a result, the game's code should be easier to understand and therefore more likely to be maintained correctly.

Furthermore, with the "... increasing size and complexity of today's software systems" (Moser and Misic, 1997), managing complexity is becoming a key quality concern, and using abstractions inherent in PODs to reduce complexity may help lower the quality risks inherent in complex software by simplifying the context of design problems.

9.3 Maintainability

The global Maintainability Index tendency, correspondingly, shows that the game's types have improved by 2.61% per type.

While this shows that using the candidate PODs have contributed positively to improving the maintainability of the codebase, it is less pronounced than, in comparison, to the improvements seen reducing class coupling (13.5%) or code complexity (16.44%).

This indicates that while the PODs have been effective in improving the maintainability Index, they are less efficient in doing so compared to the other metrics.

9.4 Quality of PODs

The results also show that for new types (added tendency), the PODs are well-designed relative to existing types, showing on average, a 36% reduction in class coupling, 7.05% increase in maintainability, a 44.43% reduction in complexity, and that in all measured aspects, the PODs types are generally of a higher quality than the existing types. This suggests that adding PODs to an existing codebase, while potentially improving the quality of the design, can improve the code quality by reducing the coupling and associated propogation risk, simplifying code and benefitting from the improvement seen in maintainability afforded by PODs.

9.5 The Effects of Refactoring

The substantially smaller and less influential change analysis (change tendency), shows that despite the inherent quality in PODs, integrating them into a solution does affect and pose its own quality risks.

These results suggest that possibly, in the future, as the more of the game is refactored, the global tendency might see a reduction as the influence of the change tendency increases, meaning that the game quality could degrade over time, based on the current trajectory.

While this is a speculative prediction and limited to how much of the game code is currently refactored, the current change tendency suggests that the median of refactored types could generally become at least 7.5% more coupled, 13.4% more complex, and 1.1% less maintainable.

Qualifying the effectiveness of this change tendency prediction would be an interesting avenue of future research.

9.6 Design Quality in PODs

An important part of PODs is their inherent design quality, and an aspect of this research has been determining how this influences the overall code quality. To achieve this end, three PODs have been evaluated and applied to the codebase, namely The Mediator, Blackboard and the Bridge PODs.

While design quality is not directly measured, the preceding results seem to indicate that those inherent in the candidate PODs appear to improve code quality.

In implementing the Mediator, a key observation is its explicit requirement to determine of extent of the ‘relatedness’ between types. This is a particularly important because it ensures that *Law of Demeter* is followed (Yi Guo *et al.*, 2011). The law’s principle idea i.e. of, ‘talking only to your friends’, is achieved through organizing ‘friends’ into a cohesive collective managed by the mediator and defining functional boundaries encourages loose coupling.

Additionally, the process of considering who your friends are, or should be, allows the Mediator to evolve as new dependencies and their relatedness is assessed and this encourages a process of continual maintenance improvement.

A key aspect observed in the implementation of the Blackboard POD, it the explicit definition of goals and the formal relationship between subsystems through expert collaboration. This makes the separation of concerns (Stutterheim, Achten and Plasmeijer, 2018) an explicit requirement.

The Bridge POD makes it easier to change or swap-out software components because using a common interface allows for other alternative and varying implementations to be used, which provides greater flexibility in designs.

An observed consequence of depending on interfaces, is that it reduces the need for top-level design change, which has a more wide-ranging impact, as different implementations can be used without changing the overall design of the code. This is because the top-level code that uses the interfaces need not change, but only the underlying implementations do. This not only makes the code more generic, which reduces redundancy in code, increases the flexibility to easily change implementations, but also isolates the scope of change to specific underlying implementation.

In addition, an increasingly important implication for using the Bridge POD, is that it dramatically improves unit testing capability, which is, "... undisputedly the most frequently used technique in defect management" (Younessi, Zeephongsekul and Bodhisuwan, 2002). This is because it enables creating alternative testing implementations which conform to the same production interfaces used in the game and can serve as mock production implementations, "... enabling programmers to write and test their code as if all the necessary dependencies are in place", and is the basis for integrating with mocking frameworks such as Moq, RhinoMocks and Mockito. (Samimi *et al.*, 2013)

The ability to improve the quality of software by asserting its quality through testing is an important criterion for improving software quality, particularly as research shows that testing, "... helps to produce better quality code" (Shihab *et al.*, 2011), particularly when tests can be created to be more representative of production code.

9.7 Overall Game Quality

The global influence that PODs have on game types show that generally, with the inclusion of PODs, the game's code quality has improved overall.

The average game type, as a result of incorporating PODs into the game, is now 13.5% less coupled, 2.61% more maintainable and 16.44% less complex.

Furthermore, the quantitative results show that through a systematic approach to POD implementation and empirical evaluation, the results support the claim made by (Ampatzoglou, Frantzeskou and Stamelos, 2012), that PODs enhance the software quality, and that superior design quality in PODs appears to have been translated into corresponding improvements in code quality.

Chapter 10

References

Ampatzoglou, A., Frantzeskou, G. and Stamelos, I. (2012) 'A methodology to assess the impact of design patterns on software quality', *Information and software technology*, 54(4), pp. 331–346. doi: 10.1016/j.infsof.2011.10.006.

Burton-Jones, A. and Lee, A. S. (2017) 'Thinking About Measures and Measurement in Positivist Research: A Proposal for Refocusing on Fundamentals', *Information systems research*, 28(3), pp. 451–467. doi: 10.1287/isre.2017.0704.

Curtis, B., Sappidi, J. and Szynkarski, A. (2012) 'Estimating the size, cost, and types of Technical Debt', in *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 49–53. doi: 10.1109/MTD.2012.6226000.

Jaafar, F. *et al.* (2016) 'Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults', *Empirical Software Engineering*, 21(3), pp. 896–931. doi: 10.1007/s10664-015-9361-0.

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.

Khomh, F. and Gueheneuc, Y.-G. (2008) 'Do Design Patterns Impact Software Quality Positively?', in *2008 12th European Conference on Software Maintenance and Reengineering*, pp. 274–278. doi: 10.1109/CSMR.2008.4493325.

Louridas, P. (2006) 'Static code analysis', *IEEE Software*, 23(4), pp. 58–61. doi: 10.1109/MS.2006.114.

Mikejo5000 (no date) 'Calculate code metrics - Visual Studio (Windows)'. Available at: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values (Accessed: 23 December 2021).

Moser, S. and Misic, V. B. (1997) 'Measuring class coupling and cohesion: A formal metamodel approach', in *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, pp. 31–40. doi: 10.1109/APSEC.1997.640159.

Samimi, H. *et al.* (2013) 'Declarative mocking', in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery (ISSTA 2013), pp. 246–256. doi: 10.1145/2483760.2483790.

Scanniello, G. *et al.* (2015) 'Documenting Design-Pattern Instances: A Family of Experiments on Source-Code Comprehensibility', *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3), pp. 1–35. doi: 10.1145/2699696.

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

Shihab, E. *et al.* (2011) 'Prioritizing the creation of unit tests in legacy software systems', *Software, practice & experience*, 41(10), pp. 1027–1048. doi: 10.1002/spe.1053.

Stutterheim, J., Achten, P. and Plasmeijer, R. (2018) 'Maintaining Separation of Concerns Through Task Oriented Software Development', in Wang, M. and Owens, S. (eds) *Trends in Functional Programming*. Cham: Springer International Publishing (Lecture Notes in Computer Science), pp. 19–38. doi: 10.1007/978-3-319-89719-6_2.

Texas at Arlington, U. of and Mangalaraj, G. (2014) 'Distributed Cognition in Software Design', *MIS quarterly*, 38(1), pp. 249–274. doi: 10.25300/MISQ/2014/38.1.12.

Vlissides-John *et al.* (1995) *Design patterns: Elements of reusable object-oriented software*. (Addison-Wesley professional computing series).

Wedyan, F. and Abufakher, S. (2020) 'Impact of design patterns on software quality: A systematic literature review', *IET software*, 14(1), pp. 1–17. doi: 10.1049/iet-sen.2018.5446.

Yi Guo *et al.* (2011) 'An Empirical Validation of the Benefits of Adhering to the Law of Demeter', in. IEEE, pp. 239–243. doi: 10.1109/WCRE.2011.36.

Younessi, H., Zeephongsekul, P. and Bodhisuwan, W. (2002) 'A General Model of Unit Testing Efficacy', *Software quality journal*, 10(1), pp. 69–92. doi: 10.1023/A:1015724900702.

Chapter 11

Appendix A: Data Generation Details

11.1 Incorporating the Mediator POD

Four Mediators were introduced into the code's codebase.

Table 11.1: Summary of the Game's Mediator POD Types

Type	Purpose
GameMediator.cs	Mediates access to the Game class, Mazer
InfrastructureMediator.cs	Mediates access to underlying infrastructure, i.e.. Graphic, Audio, Assets
EventMediator.cs	Mediates access to game events
UiMediator.cs	Mediates access to all UI drawing

11.1.1 The Infrastructure Mediator

The *InfrastructureMediator* is now generally responsible for providing an interface that will *DrawCircle(), DrawRectangle(), DrawLine(), Load assets etc.*, and manage the game's state-machine.

Instead of calling the underlying *ISpriteBatcher* object to handle drawing operations, the *Character* classes are now modified to call the *InfrastructureMediator*, which now is responsible for delegating access to the underlying graphics subsystem. This means these classes now do not need to have direct access or dependency the sub-components that make it up.

An important implication of using the *InfrastructureMediator*, is that the main game class, *Mazer*, which controls the main game loop, now refers to and passes the *InfrastructureMediator* around as the primary means to interact with the underling drawing infrastructure. This is also true for other classes that issue drawing commands.

A large portion of the code that existed in the main game class has been moved into the *InfrastructureMediator*, and as such it is no longer responsible or directly coupled to it. This is same for the other objects that call the *InfrastructureMediator*, and this in itself is likely to represents an increased reduction of overall code for users of the Mediator.

11.1.2 The UI Mediator

The *UIMediator* has pulled together all the UI drawing functionality under a single cohesive class, and in doing so, has decoupled the main game class, *Mazer*, from directly initializing UI element types. These UI elements and UI functionalities have now been isolated within the *UIMediator* type. This includes functionality that shows the main menu, game over screen, instructions dialog and the drawing of UI elements such as panels and buttons etc., in addition to handing UI specific events such as on button-pressed.

Incidentally, the *InfrastructureMediator* is also used to draw the UI elements.

The *UIMediator* also incorporates indirect access to the *Mazer* class through referencing the *GameMediator* to start/pause/resume the level when UI events are trigged. A fair amount of reuse appears to be attributable by using

Mediators. While this is not directly measured in this research, this maybe a secondary effect of using Mediators in this way.

11.1.3 The Game Mediator

The *GameMediator* encapsulates access to the underlying game class, *Mazer*.

This means operations such as querying the Game's current level, state, player statistics etc, can be uncoupled from calling code. The *GameObject*, for example, uses the *GameMediator* to access drawing functionality to draw diagnostic information on the screen including the GameObject's perimeter, e.g. bounding boxes (used to debug object collision), and text overlays that indicate the character's AI state and/or position information. Accessing this functionality in this way, is also done in *GameWorld* and *Room* classes to assist in level drawing.

11.1.4 The Event Mediator

The *GameWorld* instead of referring directly to Game event types now goes through the *EventMediator* in order to access game events.

While the nature of the *EventMediator* is that these types are exposed directly (instead of going through another layer that abstracts the event types themselves), this is considered acceptable due to the number of event types that would need to be refactored in that way, within the restricted timeframe. The utility of having the *EventMediator* in this respect, is only to provide central place for unified access to event types instead of defining them as members of existing objects that use them, for example for raising events, which would arguably decrease cohesiveness.

The same *EventMediator* is also used in the main game class, *Mazer*.

In many cases refactoring classes requires referring to the new Mediator classes instead of the underlying types, which the Mediators now manages. This also appears to suggest that it would have an the effect potentially of reducing the number of executable lines of code.

11.2 Incorporating the Blackboard POD

Table 11.2: Summary of the Game's Blackboard POD Types

Type	Purpose
Expert.cs	Common functionality for all Experts
GameWorldBlackBoard.cs	Defines GameWorld goals and current working knowledge repository
GameWorldBlackBoardController.cs	Coordinates and controls querying game world experts
IExpertControl.cs	Common Interface for Exports
LevelExpert.cs	Level knowledge expert, e.g. number of pickups left to collect in the level
CollidingExpert.cs	Collision detection knowledge expert
MovingStateBlackboard.cs	Defines MovingState goals and current working knowledge repository
MovingStateBlackboardController.cs	Coordinates and controls querying world experts e.g. CollidingExpert and PathFindingExpert
PathFindingExpert.cs	Path finding knowledge Expert, e.g. determining if there is line-of-sight between player and enemies

11.2.1 The Game World Blackboard

The *GameWorldBlackboard* represents a combination of relevant information pertinent to game wide goals, such as determining if the level is complete, whether the player has been sighted by an enemy or not, or how many pickups are left in the game.

It calls upon the expertise of the *LevelExpert* to source and provide this information from the Level subsystem. The coordination of expertise is controlled by the aptly named *GameWorldBlackboardController*.

The *LevelExpert* upon being summoned by the controller, is responsible for determining relevant Level information, and making this information available in turn to the *GameWorldBlackboard* where game world goals are evaluated in light of this information.

11.2.2 The Moving State Blackboard

The *CollidingExpert* is now responsible for bringing information about game collisions to the *MovingStateBlackboard*, such that this information can be used later collaboratively to determine character collisions.

The moving state is part of the game's AI system which is represented by a finite state machine and where many decisions need to be made while the character or player is moving. Decisions based on answers to questions/goals such as ,'Is the Player sighted?', or, 'Is there a collision with a wall?', and, 'Is the enemy in the same row or column as the player?', are all facilitated by information provided by both the *CollidingExport* and *PathFindingExpert*, which routinely report their corresponding information to central blackboard for evaluation.

Indeed, delegating path finding expertise to the *PathFindingExpert* has meant that specific and targeted functionality is now managed solely by the expert. It also keeps this functionality separate and cohesive within the expert class, meaning future changes are likely to remain confined within it.

Through the Blackboard, a unified combined assessment of all contributed expert information is achieved via a central accessible repository for assessment of specific questions and goals.

Not withstanding the ease of which disparate systems such as Collision Detection and Path Finding can be brought together in an co-ordinated way to achieve common goals, is the obviousness of their apparent decoupled collaboration. In this way, these specific roles are not only owned by the expert, but it is also clear where maintenance of this responsibility lies. This possibly will contribute to making this code more maintainable.

The expert functionality is therefore now also separate from other less related logic, which in the future might have become more closely coupled to it, had it remained integrated with the specific functionality now managed by the expert.

This separation of logic and responsibility might help to isolate specific roles, declutter existing code, and therefor reduce the amount of tasks and responsibility managed by larger general code, for example, relieving the *Level* class from the responsibility of ascertaining if it is complete, while catering to other responsibilities also such as positioning assets within the level.

The goals that are defined within the Blackboard are now well defined and easily determined, making it easier to interrogate the blackboard, than sourcing the information directly from adjacent subsystems.

11.3 Incorporating the Bridge POD

Table 11.3: Bridge POD Types

Type	POD	Purpose
IGameObject/GameObject	Bridge	Defines interface/implementation for Game objects
IRoom/Room	Bridge	Defines interface/implementation for Room objects
ILevel/Level	Bridge	Defines interface/implementation for Level objects

In introducing the Bridge POD into the Game, required a great deal of refactoring such that existing types that were to be 'bridged' such as GameObject, Room and Level needed to instead be referred to their newly defined interfaces, IGameObject, IRoom and ILevel. This effectively coupled the game code to these interfaces instead of their implementations. This was by far the most time-consuming of activities, affecting almost all the subsystems that referred to the original implementations. This however might also represent potentially the most wide-ranging effect on the code base.

The operation itself was not difficult, despite the extent to which it applied, as the interface types now being referred to, already defined the same methods that were previously being referred to by the original implementations. This did however require effort ensuring that these methods were part of the newly introduced interface types.

This POD makes it easier to change the implementations behind the interfaces without materially modifying code that refers to their corresponding implementations. This is thought to improve its maintainability.

Chapter 12

Appendix B: Code commits

Three PODs have been ingerated into to the codebase's beta branch, namely The Mediator, Blackbord and the Bridge PODs. A SCA is run on both alpha/beta branches.

The game's code is freely available online at https://github.com/stumathews/MonoMazer and relevant source code commits, and implementation details can be found in the appendices, while the source code for the alpha/beta branch is.

Table 12.1: Beta branch POD code commits

#	Hash	Description
1	2cf8caf998009a192be201a5389420fd6295c712	(HEAD -> beta, origin/beta) use bridge pattern for room and gameobjec
2	371150d2922ed87d6b53f896ae514e073a7e847a	use Bridge Pattern on Level class
3	eb73e88389ae6ac7101f0ae8154124ebdaab5156	blackboard
4	584c1da3daeb5a202c702933e82dfa2f45f8407a	pass event mediator from Mazer into gameworld
5	37c3045b426e1234a3e0f7f5f1d17684a9518c4b	pass event mediator around, dont copy() objects
6	dd7a24dd786684b077afaa2b1a4e16c9a32d455b	improve target detection in moving state
7	857c2990198967620dea771c1fcfc2bfe6f2e52a	Add blackboard pattern to Gameworld & NPCMovingState
8	25ce281db9c8b944ecf58ba930b924d2dd805cd4	Add folders
9	1fc897cefc2ec2696ae65bc081110332804e88af	simple refactoring
10	58a5f103f6967661f6c075bbc59909e8d5207353	add event mediator
11	8f0516801bdbaf08ce9693f8df419268e88884bc	implement infrastructure, game and ui mediators

Chapter 13

Appendix C: Top level Game Architecture Objects

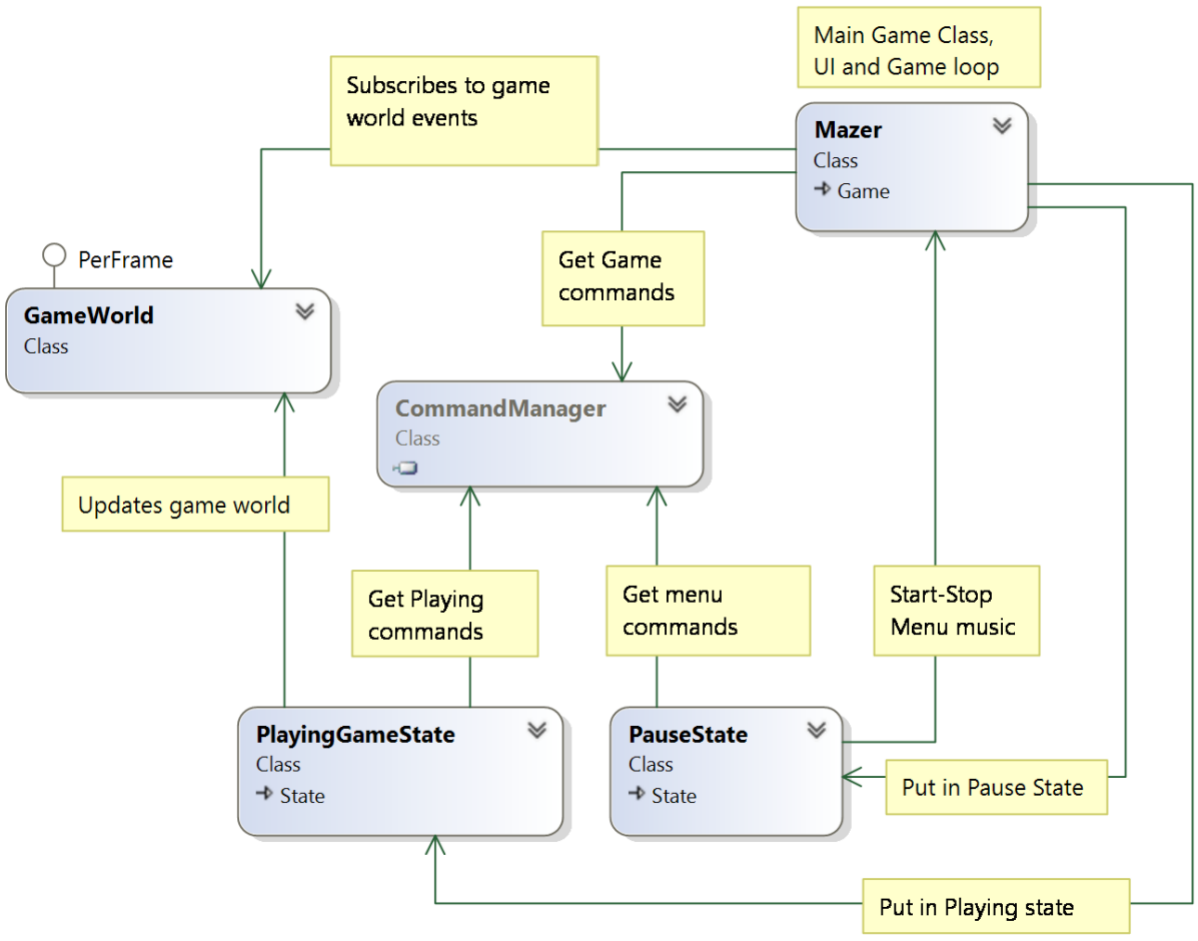

Figure 13.1: Figure: Top level Game Architecture Objects in MonoMazer