**Software architecture overview**

Design Patterns - MT803

MTU Kerry

Table of Contents

[Background Information 3](#_Toc119920863)

[Initial state flow 3](#_Toc119920864)

[Initial structure 3](#_Toc119920865)

[Requirements 3](#_Toc119920866)

[Functional 3](#_Toc119920867)

[Non functional 3](#_Toc119920868)

[Use Cases 4](#_Toc119920869)

[Change graphics options 4](#_Toc119920870)

[Load board from file 4](#_Toc119920871)

[Plays a valid move 5](#_Toc119920872)

[Board is saved 5](#_Toc119920873)

[UML 6](#_Toc119920874)

[Components 6](#_Toc119920875)

[GUI factory 6](#_Toc119920876)

[Board saving 6](#_Toc119920877)

[Presentation, input, and logic 7](#_Toc119920878)

[Players 7](#_Toc119920879)

[Position 7](#_Toc119920880)

[GUI 7](#_Toc119920881)

[Design rational 7](#_Toc119920882)

[Microservice architecture 7](#_Toc119920883)

[Adaptor 8](#_Toc119920884)

[Bridge 8](#_Toc119920885)

[State 8](#_Toc119920886)

[Simple factory 9](#_Toc119920887)

[Dynamic factory 9](#_Toc119920888)

[Command 9](#_Toc119920889)

[Choice for system 9](#_Toc119920890)

[Redesign 10](#_Toc119920891)

[New structure 10](#_Toc119920892)

[Component 11](#_Toc119920893)

[Port num. 11](#_Toc119920894)

[Adaptor 12](#_Toc119920895)

[State 13](#_Toc119920896)

[Simple factory 14](#_Toc119920897)

[Command 14](#_Toc119920898)

[Refactor 15](#_Toc119920899)

[Magic numbers and Magic text 15](#_Toc119920900)

[Moved methods 15](#_Toc119920901)

[Extracted methods 15](#_Toc119920902)

[Refactored Class Diagram 15](#_Toc119920903)

[Conclusion 15](#_Toc119920904)

[Reflection 16](#_Toc119920905)

# Background Information

Chess is an old game enjoyed by many. The initial implementation is one executable, that requires all players to be on one computer screen. I created this game for a short Object-Oriented Design module.

# Initial state flow

Gameplay stage

Game end

Settings

Settings chosen

King struck

Game reset

# Initial structure

## Requirements

### Functional

* Players must be able to set the size of their board.
* Players must be able to set the colour of both teams.
* Players must be able to load a board state from a file before the game.
* Players must be able to save the current board state to a file during the game.
* Players must be able to move their pieces in valid moves on their turn during the game and have their turn end when a valid move is made.

### Non functional

* Players valid moves must be made visible in real time (1/3 seconds or less).
* Players must not be able to cheat by taking invalid chess moves.
* The product should be able to be maintained to fix any defect found going forward

## Use Cases

### Change graphics options

User



Figure – Use case diagram for how the user changes the graphical options

Primary actor: User

Brief: The user changes the size of the board and/or the colour of any team.

Postconditions: The size of the board will be the user’s choice when the game is started. The colour of both teams will be what the user chooses when the game starts. The colour previews of the team are what the user choose.

Preconditions: application was started but no game was yet started.

Triggers: The user clicked the board size drop down or teams’ colour.

Basic flow:

Change screen size

1. The system displays the options of board size: small, medium, or large.
2. The user chooses one of the options.
3. The system stores the selected board size.
4. The system hides the board size options.

Change team colour

1. The user presses a team colour.
2. The system displays a colour selector window.
3. The user selects a colour and confirms.
4. The system stores the selected colour.
5. The system hides the colour selector window.

### Load board from file

User



Figure – Use case diagram for how the user loads a board from a file

Primary actor: User

Brief: The user loads a board state from a .brd file.

Postconditions: The state of the board will be as describe by the file when the game is started.

Preconditions: application was started but no game was yet started.

Triggers: The user click the load board button.

Basic flow:

1. The system displays a file select window.
2. The user selects a file.
3. The system hides the file select window.
4. The file is stored to the board state.

### Plays a valid move

User



Figure – Use case diagram for how the user moves a piece

Primary actor: User

Brief: The user moves a piece on their turn.

Postconditions: The user’s piece will have been moved and it will be the other user’s turn.

Preconditions: The game must have started, and it must be the user’s turn.

Triggers: The user clicks down on their coloured piece during their turn.

Basic flow:

1. The user drags their piece
2. The system detects the user’s cursor is over a different grid tile from where it started
3. The system validates the move.
4. If the more is valid, the system changes the position of the visual of the selected piece.
5. When the user lets go and the piece is in a valid move, the piece is moved.
   1. If the piece is in a invalid position or if the piece was not moved, the selected piece is not moved and the graphic is reset.

### Board is saved

User



Figure – User case diagram for how the used saves the board

Primary actor: User

Brief: The user saves the board state to a file

Postconditions: The user will have a file on their computer containing the board state.

Preconditions: The game must have started.

Triggers: The user pressed ‘s’ on their keyboard.

Basic flow:

1. The system displays a menu.
2. They user enters a name for the board state and confirms.
3. The board state is saved under a file called the user’s inputted name followed by ‘.brd’.

## UML

A screenshot of a computer

Description automatically generated with medium confidence

Figure – UML class diagram of chess application

## Components

### GUI factory

GameControl acts as a simple factory which starts the entire program. It contains the logic to guarantee the state is correct when the game is began and restarted.

### Board saving

Saving the board is hardcoded to save to file in the KeyShortCuts class.

private void saveBoard(){  
 String output = JOptionPane.*showInputDialog*(null,"Name board state","Save Board",JOptionPane.*QUESTION\_MESSAGE*);  
 try{  
 FileOutputStream outputStream = new FileOutputStream(output + ".brd");  
 ObjectOutputStream out = new ObjectOutputStream(outputStream);  
 out.writeObject(Board.*grid*);  
 out.close();  
 }  
 catch (IOException e){  
 e.printStackTrace();  
 System.*err*.println(e.getMessage() + " " + e.getCause());  
 }  
}

Figure – SaveBoard method in the KeyShortCuts class.

This limits the game to only file saves only. KeyShortCuts is not a class which would be expected to contain such functionality, meaning maintain the code is more difficult.

### Presentation, input, and logic

The stage of the game where the players see the board and can move their piece, the gameplay stage, has its functionality distributed over 4 classes: Player, Board, KeyShortCuts and Piece. However, there are not clear divides between the responsibilities of each class. The state of the board is access in all these classes as it is public and static. Both Player and KeyShortCuts handle input. Game rules are split between player and board. Presentation is purely performed by player. This distribution has no clear pattern and so would be more difficult to maintain.

#### Chess pieces

Currently, the chess pieces are abstracted via an abstract class in terms of structure. However, they are all instanced within the board class, removing the loose coupling allowed by the abstract class. The board prevents the pieces from being open to extension as any classes which inherit from them would have to be altered in the board to be used. Therefore, changes would most easily be done with modification, breaking the Open/Closed principal.

### Players

Player class stores some data and implements both input and game logic. It is a concrete class which assumes only 2 players and a basic chess rule set for these players. This limits the extendibility of the system. The rules of the game are so hardcoded into the class that it has a reference to the Board class.

### Position

The position class is a singleton. It needs to be a singleton as it is a helper class for handling game logic. This responsibility is divided between multiple classes so preventing clean inheritance. If the division of responsibilities was more defined, this anti pattern would likely no longer be required.

## GUI

The graphical interface is currently managed by classes, such as StartMenu and Board, which get created but never store. It is difficult to track when these classes would be garbage collected. Each of these are created by their predecessor, potentially leaving their predecessors in memory after they are necessary. This is difficult to follow and so harms maintainability

# Design rational

## Microservice architecture

The microservice arcature is when each responsibility is separated into its own separate program. Pacheco emphasises there should clear boundaries between each program with each one being able to be separately deployed. (Pacheco, 2018) No program would require code from another to compile or run, including libraries.

This results in the ability to have the system be composed of loosely coupled parts which can be worked on independently. So, the code base of each component can be loaded and compile independently, speeding up iteration time, and smaller teams can work on each component separately, so there is less communication overhead. Each feature has one responsibility, meaning less complexity. This increases the component’s testability and maintainability. When a component becomes obsolete, weighted down by technical debt, or just outdated, they can be replaced without interrupting the larger system as much as in a monolith.

In a microservice system, issues more commonly appear in communication between the components. (Thönes, 2015) Specialised tooling is required to have stack traces between components and components are not limited in who they can communicate. This can make system level defects more difficult to analysis. Complexity also arised from the network connecting the components. (Richardson, 2022) Each component also will be more complex than an equivalent class in a monolith system as they each need all the functionality to communicate with one another in a loosely coupled system.

## Adaptor

The adaptor pattern is when an interface is provided to a class so that different adaptors can be placed which translate the class to the interface into other formats. This pattern can allow for the class’s calls to be extended without modification allowing open/close principle. Separating different interfaces to different concrete classes reduced the complexity of each implementation. However, this does increase the complexity and size of the system as a simple call to a class now requires an interface and a class per context i.e. a class per storage system: oracle database, file, NoSQL database.(Refactoring Guru, 2022a)

## Bridge

The bridge pattern consists of an abstract class with aggregates an interface. These two classes aim to provide 2 dimensions of expansion. (Refactoring Guru, 2022b) Both the abstract and interface can be changed to extend the functionality without modifying the calling class. Responsibility for functionality can also be divided between the 2 classes, allowing for any combination of functionality while still only producing one class per function. This aims to prevent over reliance on inheritance for composite functionality. (Soshin, 2018) The functionality can also be modified at runtime by changing the underlying classes. However, an abstract class and interface are required for this pattern and a concrete class per functionality, increasing the complexity or the system.

## State

A state pattern is when a class which contains both a reference to a state, either interface or abstract class, and an aggregate of states. The aim of this system is to simplify large chains of conditionals which typically occur in a pattern and change said pattern at set points. These patterns are removed into their own state implementation, simplifying their conditional trees. This pattern does resemble the strategy pattern, however unlike that pattern, the states are typically hidden from any class beyond the one which contains them and are not controlled by any other class. (Soshin, 2018; Freeman and Robson, 2020) The logic can then be easily extended by adding new states and only modifying the few states which need to be able to transition to them. The transitions can be decided by the class containing the states or the states themselves. Whether a state is an interface, or an abstract class depends on how much overlapping logic they contain.

States can have significant boiler plate code in each one as they may need to be able to transition to other states. (Refactoring Guru, 2022e). This also increases the complexity of the system as each state requires its own concrete class and a state interface/abstract class is required.

## Simple factory

The simple factory pattern just moves the instantiation process to a separate class. The class is only meant to have the responsibility for creating another class, removing this responsibility from the initial class. Instanced classes typically require additional logic or may be able to be optimised by additional logic. The simple factory can store this logic (Soshin, 2018) without code duplication. Any future modifications can be done in the factory alone. (Freeman and Robson, 2020). The factory also decouples the object produced from the from the classes they are used in. This allows for extensions as the factory can later return super classes of the objects, expanding functionality without modifying the calling class. (Refactoring Guru, 2022d)

The classes are however coupled to the factory. Expanding the number of objects, the factory creates will increase its complexity. For large, multi-dimensional extensions, other patterns, such as abstract factories or builders, would produce less complexity and code to the system.

## Dynamic factory

Dynamic factories were proposed by León Welicki, Joseph W. Yoder, and Rebecca Wirfs-Brock. (Welicki, Yoder and Wirfs-Brock, 2008) This pattern is like the simple factory, but it relies on external meta data to choose what object is returned from it. Through this reliance on meta data, the objects it returns can be varied without alteration to the code. Depending on the implementation, this may include not requiring a restart. This can speed up iteration time and ease of making changes. These benefits come without modification to the client from a simple factory pattern.

The reliance on metadata does create more complexity and makes debugging more difficult as this metadata many are not recorded in traces. This metadata needs to maintain outside the system and is a new vector for defect.

## Command

The command pattern takes the communication between components and decouples them, coupling to it instead. The caller just constructs the command. Then the command will be sent, routed and received before being handled by another component. (Refactoring Guru, 2022c) This pattern requires more classes as each part, command, sender, and receiver, must be created and maintained. In exchange, the communication between components can be easily altered by either component or the 2 components cannot tightly couple as they never gain direct references to each other.

## Choice for system

The microsystems architecture would enable fixing the poor division of responsibilities between classes as the responsibilities will need to be clearly defined between the components. This would improve the maintainability as each function will be able to be modified without effecting other components to the same extent as they are. Telemetry systems, such as OpenTelemetry, can be implemented later to minimise the debugging issues.

The simple factory would help resolve the tight coupling between each piece’s concrete class and the game logic. The dynamic factory benefits of easier modification are nullified as each piece is a class which needs to be compiled. So, the extra complexity of the dynamic factory and the need to maintain metadata are not justified by the benefits.

Bridge or adaptor could be used to enable more looser coupling and extensibility for board extending the functionality in a structured way. Adaptor would be more appropriate as this feature only requires translation form the call to save the board to a way of storing it i.e., database, file, or cloud. The bridge’s additional complexity provides multiple dimensions of expansion which is not required here.

The hard coded player rules could be resolved if players were states. Player rules could be each a different state so new rulesets could be changed. This would enable more players to be added for additional potential game mode, such as 2 players versus 2 players, without having to modify the game logic.

Extending the player logic would be going beyond the requirements of this project and require additional complexity to create and maintain. Multiple players would require to be tested. Additionally, the state pattern is not applicable as the player does not change behaviour, the core complexity that the state pattern aims to control.

However, the GUI could be divided to units with distinct behaviour. Separating each menu out to its own state would allow for the code to be more readable and have clearer what happens to the memory as the states change.

The command pattern is necessary in this design to support the microservice architecture. This pattern enables the components to have clear boundaries, reinforcing the boundaries, increasing maintainability further.

# Redesign

## New structure

Diagram

Description automatically generatedFigure – Messages exchanged between services

A screenshot of a video game

Description automatically generatedFigure – Class diagram of messaging system for the microservices.

|  |  |
| --- | --- |
| Component | Port num. |
| Frontend | 65535 |
| Backend | 65534 |
| Data Access | 65533 |

Table – Port numbers of each component

The functionality has now been divided over 3 components: Data Access, backend, and frontend. They are loosely coupled via the command pattern, only exchanging commands which are handled by services. Each service could be changed with a different service, allowing for easier replacement and maintenance. Each component sends messages out via client objects. These can too be switched out to target a different network, such as the internet to make it a true microservice product.

Going from the tightly coupled app to it required more than just moving logic. The flow of logic had to change as calls in 1 function were separated into different methods. Changing from the heavy use of static properties, leaking classes into each other, meant the logic had to be fundamentally altered to decouple them. Initial attempts at decoupling the classes attempted to use the Java socket, aiming to function like running multiple docker instances on a local machine. This direction was decided to require too much work while offering low level functionality. Using a custom solution also allowed for a bespoke command pattern to be incorporated. The port number of each component is a product of this phase of development.

The piece class had to have its methods altered to take in a reference to the board it used to move. This heavy alteration created many more bugs, which themselves were difficult to debug as they could be a defect in many classes; network, receiver, sender, or even one of the component classes. True microservices were not achieved as the frontend and backend components rely on the piece package.

This has meant that changes to the interface requires more work. The client of the sender and the server of the receiver must be changed independently. Changing one without the other does not result in any error but will fail silently. Future failures on the boundary will be more difficult to identify. Also, some responsibilities are duplicated as the front and back do not share resource. The true board and the visual board must be managed separately and synced. This means logic must repeat between them, leaving more room for similar errors and both must be changed in sync to avoid desync issues.

## Adaptor

Graphical user interface, diagram

Description automatically generated

Figure – Class diagram of Board Manager component.

The board manager microservice now contains any board saver. These each can adapt the save and load calls to other interfaces. Board Saver File changes these calls to calls to save the board as a file under the name specified.

This pattern’s simplicity and reuse of interfaces means it is easier than others to implement while providing its benefits. This was done by extracting the relevant methods to new classes and providing the interface. Any future for other data sources can easily be implemented by creating a new concrete class implementing Data Access and a change to what the Data Access Service implements.

## State

Graphical user interface, application

Description automatically generatedFigure – Partial Class diagram of the frontend service.

The frontend component contains a menu member to hold the state of its GUI. This can change to any of the 3 states: game, start, and win. Changing between states has been made easy as the frontend just needs to call the current menu’s clean method then instantiate the new state.

Dividing each menu into these states made it easier to refactor the software as it provided a structure. The previous system had each menu instantiate the new one and stop. There was little structure to it, such as how cleaning the state could be the responsibility of either the new class or old class. With the menu abstract class, there is a defined responsibility for menus to clean up before the next one is created, and the constructor sets up the state for them. Also, memory is more predictable as each menu is only left for the garbage collector once it had finished, unlike before where each class was not contained in any clear parent as it operated.

## Simple factory

Graphical user interface

Description automatically generated

Figure – Class diagram of pieces package.

The piece factory class performs the instantiation of the classes which inherit from piece on behalf of other classes. With the piece factory, in addition to the existing piece abstract class, other classes no longer must interact with the implementation classes for each piece. This allows for these classes to be extended or replaced while only modifying these 2 classes, allowing for looser coupling, in turn having better maintainability, extendibility and reduced code reuse.

This, like the adaptor, was a simple pattern that did not take long to implement while providing its benefits and only added 1 new class. There was some issue deciding how much functionality the factory should hold. The initial design would have the factory only take in board coordinates and return the appropriate piece. This expanded the scope of the factory, resulting in it not being able to create pieces for other contexts. As such, it was scaled back to be fed a Enum and return the appropriate piece.

## Command

A screenshot of a video game

Description automatically generated

Figure – Class diagram of network package, senders in other packages and receivers in other packages.

Message is the main command class in this implementation of the command pattern. It only contains a source, destination, and payload. Clients and services can send messages but only services can receive them. Implementing this pattern has allowed for the loose coupling required by the microservice architecture used.

A difficulty in implementing this pattern is deciding on the consistency of the command object. For this project, 1 command object with a varied payload was chosen. This allowed for a simple implementation as each receiver would just check for the class of the payload and decide accordingly. However, there are cases where this was insufficient without special data classes, which are code smells. A hacky solution was used for the win message due to lack of time. The winning team were encoded in the source address. The initial design would not have required this hack, as there would be 2 different fields for command and payload. This did not seem necessary originally, so the command field was dropped.

Implementing the receiver and sender was useful as it created clear responsibilities for each component. Only those 2 classes would have to be referenced to figure out how the component would interact with any other component.

# Refactor

## Magic numbers and Magic text

Text was removed from methods into final properties on the class they were contained in. An example of this was the extension of the board state files. Numbers were similarly removed and replaced with descriptively named constants.

## Moved methods

Methods were moved to more appropriate classes. Methods were moved from the god class board and the old player class to the new backend and frontend components.

## Extracted methods

Extracted a few lines of code which have a similar focus into their own separate method. (2) IntelliJ was able to detect when extracted code had duplicates and replace said duplicates with different arguments for the method. This has decreases code duplication. Extraction overall has made the code more readable. Block of over 70 lines which performed many separate functions have been replaced with calls to named methods describing what they do, allowing the block to be more understood.

# Refactored Class Diagram

A screenshot of a computer

Description automatically generated with medium confidence

Figure – Class diagram of entire project post refactoring and re-structuring.

# Conclusion

The class diagrams show the complexity of the system has increased drastically. The class count has doubled from 15 to 30. For this added complexity, the system has become significantly more modular and, hopefully, easier to navigate for maintenance or extensions.

This has come with the issue that state is more difficult to manage between components. Unit testing would be easy on this system as each component could be sent messages and the test would listen for correct responses and verify correct state changes inside the component. Yet issues between components could be in the network, services.

# Reflection

As was correctly pointed out to me, the microservice architecture was an ambitious approach. My initial belief that this would be the more efficient choice with a higher point ceiling may or may not have been correct, but it did result in more difficult refactoring.

Initially, there was an attempt to really embrace the microservice ideology of one component one responsibility resulting in a complex network shown below (Fig. 14). Transforming the existing software into this turned out to be too large of task due to how tightly coupled it was. Even the final product resembles more a merger or layers architecture with client server communication( more accurately peer to peer between the frontend and backend and client server between the backend and data access) than true microservices due to multiple common dependencies.

Diagram

Description automatically generated

Figure – Original component network structure.

These dependencies derive from the fact data exchange happens with java object, meaning both ends require the class. This was chosen purely out of convenience so that a JSON library would have taken time to implement.

Due to the difficulty, I managed to gain valuable experience of common issue which arise from porting a small monolith application to a decoupled, well segmented system. The data access component, which was only a simple input, output system without state was simple enough to port. However, frontend and backend relied on the board state. I chose to give them both a copy which led to new desync issues. The piece class, which followed the OOP principal of tying data to functionality, caused issues as the front end required the piece’s visual and name data while the backend required the methods. Without a way to break a class up, the frontend and backend became coupled without a need to be. If I were to redesign this system further, I would move away from OOP, relying more on functional and data driven principals.