**Software architecture overview**

Design Patterns - MT803

MTU Kerry

Table of Contents

[Background Information 2](#_Toc119325205)

[Initial state flow 2](#_Toc119325206)

[Initial structure 2](#_Toc119325207)

[Requirements 2](#_Toc119325208)

[Functional 2](#_Toc119325209)

[Non functional 2](#_Toc119325210)

[Use Cases 3](#_Toc119325211)

[Change graphics options 3](#_Toc119325212)

[Load board from file 3](#_Toc119325213)

[Plays a valid move 4](#_Toc119325214)

[Board is saved 4](#_Toc119325215)

[Components 5](#_Toc119325216)

[Board saving 5](#_Toc119325217)

[Presentation, input, and logic 5](#_Toc119325218)

[Players 6](#_Toc119325219)

[Position 6](#_Toc119325220)

[UML 6](#_Toc119325221)

[Design rational 6](#_Toc119325222)

[Microservice architecture 6](#_Toc119325223)

[Adaptor 7](#_Toc119325224)

[Bridge 7](#_Toc119325225)

[State 7](#_Toc119325226)

[Simple factory 7](#_Toc119325227)

[Dynamic factory 8](#_Toc119325228)

[Choice for system 8](#_Toc119325229)

# 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

Player 2’s turn

Player 1’s turn

Game end

Settings

Gameplay stage

Settings chosen

Player moved

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 5 – UML class diagram of chess application

## Components

### 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 3 classes: Player, Board and KeyShortCuts. However, there are not clear divides between the responsibilities of each class. 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.

# 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, 2022d). 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, 2022c)

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.

## 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.

# Redesign and refactor

## New structure

Diagram

Description automatically generated

Figure – Structure of components in microservices network. Each edge is a message that can be sent between components.

|  |  |
| --- | --- |
| Component | Port num. |
| Piece | 65535 |
| GUI | 65534 |
| BoardManager | 65533 |
| Logic | 65532 |
| Player | 65531 |