FIT3077: Software Engineering: Architecture & Design

Sprint Two Deliverables

Christine Chiong, 31985270
Jensen Kau, 33053332
Wong Yi Zhen Nicholas, 32577869

Table of Contents

Architecture and Design Rationales

Class Diagram

The UML Class Diagram has been pushed to GitLab. Since our class diagram is quite huge, attaching it to this document would make it look blurry. Hence, for the best quality, kindly download the UML Class Diagram PDF from our team's repository. Alternatively, attached is the Google Drive and LucidChart links to the same class diagram.

Google Drive link to UML Class Diagram:
https://drive.google.com/file/d/1DHHfvLwr8OXUPGnJS4af5hb42AM1wWoK/view?usp=sharing

LucidChart:
https://lucid.app/lucidchart/b7faec51-67c3-45a9-adb6-150357f89f11/edit?viewport_loc=-3315%2C-1614%2C6587%2C3170%2C0_0&invitationId=inv_e05d84d6-9ff3-4246-b543-107974dbb609

Key Classes we debated as a Team

Issue: A computer shares a lot of functionality with the normal player and can be implemented as both a class or a method. Which is the better implementation for the game?

Alternative	**A:** Implement the computer as a concrete class, ComputerAI, and therefore also include an abstract Player class for both the HumanPlayer and ComputerAI to inherit from	**B:** Implement the computer as a method which can be invoked in the main game loop's class, MorrisGame, during the computer's turn
Advantages	● Promotes code reusability. ● Promotes code extensibility should there be more player types in the future and is in line with the Open-Closed Principle. ● Promotes separation of concerns which is in line with the Single Responsibility Principle. ● High level class (MorrisGame) does not depend on low level class (concrete child Player classes) but instead depends on an abstraction (Player abstract class) which is in line with the Dependency Inversion Principle.	● Simpler design as there are less classes. ● Method can be invoked within the same class as game loop does not have to go through a chain of functions to get the method to be executed.
Disadvantages	● More complex design with more classes. ● Adds a layer of abstraction to the design.	● Duplicated code as the computer shares a lot of methods with the normal HumanPlayer class. ● Code is not extensible if developers wish to add new player types as we would then have to modify MorrisGame going against the Open-Closed Principle. ● MorrisGame would now know how to execute a computer's turn when by ideal standards, it should only focus on executing the game loop itself which goes against the Single Responsibility Principle.

Decision: Based on the decision table above, it is clear that alternative A is the better choice because the advantages of alternative A significantly outweigh its disadvantages and alternative B has more disadvantages than advantages.

Issue: Removing a token can be implemented as a method within each of the moves once they are executed or extracted into its own class. Which is the better implementation for removing tokens in the game?

Alternative	**A:** Implement removing the token as a concrete class, RemoveToken and have it extend from the Move abstract class	**B:** Implement removing the token as a method in the Move abstract class which can be invoked by all concrete child Move classes
Advantages	● Semantically more accurate implementation. ● Promotes code reusability. ● Prevents modification of the parent abstract class code should there be a need to modify the method for removing a token in the future which is in line with the Open-Closed Principle. ● Promotes separation of concerns which is in line with the Single Responsibility Principle. ● Code in specific classes is simpler due to polymorphism.	● Simpler design as there are fewer classes. ● Lesser dependencies.
Disadvantages	● More complex design with more classes. ● More dependencies between the concrete child Move classes to the RemoveToken class.	● Move abstract class would now know about how to execute a remove token move when by ideal standards, it should only focus on defining methods common to all Move classes which goes against the Single Responsibility Principle. ● Having the remove token method inside the Move abstract class would mean that if there were any changes, it would have to be made there, making it open to modification and violating the Open-Closed Principle. ● Code in Move abstract class is more complex without polymorphism.

Decision: Based on the decision table above, it is clear that alternative A is the better choice because the advantages of alternative A outweigh its disadvantages and alternative B has more disadvantages than advantages. Even though there is the fact that alternative B does have lesser dependencies than alternative A, we are willing to trade that off for code that abides by separation of concerns for ease of refactoring in the future.

Key Relationships in our Class Diagram

Before getting into the explanation, we will clarify what we based the decision of the type of relationship between two classes on. Looking at it from an implementation perspective, a relationship is a:

1. **Dependency** when the class does not have an instance of the other class in the relationship as an attribute but **knows about** this other class in its method parameters, body or return type.
2. **Association or aggregation** when the class **has an** instance of the other class in the relationship as an attribute but does not create this instance in its body. Instead, it has this instance passed in through the constructor. Hence, the instance of the other class can stand independently should the parent class die.
3. **Composition** is when the class **owns an** instance of the other class in the relationship, creates it in its constructor and sets it as an attribute. Hence, the instance of the other class is rendered useless when its parent class dies.

The thing that differentiates between an association and aggregation relationship is more semantics where classes in an aggregation relationship have a stronger relationship and a more obvious “has a” relationship compared to those in an association relationship. Classes in an association relationship will use the other class’ functionality but do not have that tight coupling. With this in mind, there are key relationships that can be discussed in the class diagram.

MorrisGame to MorrisBoard
The MorrisGame has a **composition** relationship to the MorrisBoard. In other words, a MorrisGame **owns a** MorrisBoard. This is because when a MorrisGame is created, an instance of the MorrisBoard is also created together with the MorrisGame in its constructor and this is in line with our understanding of how the composition relationship works in our statement above. Additionally, to make matters more obvious, the MorrisGame is also the exclusive container of MorrisBoard which means that no other class within the diagram has or creates an instance of MorrisBoard (because in that case, the MorrisBoard could stand independently from the MorrisGame which would make it an aggregation instead). This means that should MorrisGame cease to exist, MorrisBoard will also be destroyed along with it and cannot exist meaningfully on its own.

MorrisGame to Player
The MorrisGame has an **aggregation** relationship to the Player. In other words, a MorrisGame **has two** Players. This is because the two Player instances are created outside of MorrisGame and inside of the MorrisGameFactory in its createGame method before being passed into MorrisGame as parameters into its constructor which is in line with our understanding of how the aggregation relationship works in our statement above. Since Player does not depend on the creation of MorrisGame for it to be created and can exist meaningfully on its own when MorrisGame is destroyed, this means that this relationship is clearly not a composition relationship and instead an aggregation.

Decisions around Inheritance

In general, we decided to use inheritance for the purpose of:

1. **Promoting reusability of code,** in line with the Don't Repeat Yourself (DRY) Principle, as concrete classes of an abstract class usually share similar attributes and methods.
2. **Promoting separation of concerns,** in line with the Single Responsibility Principle, to prevent the creation of an all-knowing God class that knows and does more than it should be which would make debugging more difficult.
3. **Promoting extensibility instead of modification,** in line with the Open-Closed Principle, so that new functionality can be added to the game easily without the need to modify already existing code.
4. **Having higher level modules depend on an abstraction instead of lower level modules,** in line with the Dependency Inversion Principle, which decouples the modules making it easier to change without affecting each other.

Inversely, there is no need for abstraction if none of these criteria needs to be adhered to. With this being said, in our class diagram we have applied the use of inheritance in four classes: SceneController, Player, Move, and MorrisGame. The justification for using abstraction in these four classes is as follows:

Class	**Justification**
SceneController	There will be multiple scenes within the game to be switched to. Therefore, there will be duplicated boilerplate code for switching scenes in multiple classes if no layer of abstraction is added. Either that or a possibly better but not ideal solution is having all switch methods in a singleton class which exposes switchScene to all classes. This is not optimal either because classes which are not involved in switching scenes should not know about this method going against Separation of Concerns. As a result, the abstract SceneController class containing a switchScene method which takes in a parameter of scene name is the most ideal solution to prevent these situations from happening. Concrete child classes, TitleScreenController, GameScreenController, InstructionScreenController, and ResultScreenController, which inherit from SceneController can then all invoke this method (as its visibility is protected) while passing in the destination scene name when needing to switch scenes and, at the same time, specify methods which are specific to the classes themselves.
Player	As our chosen advanced requirement was to implement a computer for the player to play against (in single player mode), it would make sense to create a base abstract class, Player, for both the normal HumanPlayer and ComputerAI to inherit from because both player types share a lot of common methods and attributes. Without this layer of abstraction, there will be code duplication which violates the DRY principle and depending on where each player's code ends up, the Single Responsibility Principle may also be violated as explained

	in the first section of this rationale. Additionally, the main game loop class, MorrisGame would also have to know about the inner workings of both specific player types, increasing the coupling between these classes which violates the Dependency Inversion Principle.
Move	Aligned with our chosen design pattern to be implemented which is the Command Design Pattern, having a Move abstract class for specific moves to inherit from is essential to adhere to this pattern. The Command (Move abstract class), which adds a layer of abstraction, hides the underlying game logic from both the Invoker (Player class) and the Receiver (MorrisBoard class) by packaging all of its details into encapsulated objects with a common performMove method which can be invoked by the receiver, adhering to the Dependency Inversion Principle. Additionally, should there be more special moves to be implemented in the future, for example for different Morris board variants, they can simply extend from this parent abstract class which acts as a blueprint, aligning with the Open-Closed Principle.
MorrisGame	MorrisGame is mainly implemented as an abstract class for the purpose of increasing the extensibility of the game should future requirements need us to implement different game modes other than LocalGame which adheres to the Open-Closed Principle. Each child class that inherits from the MorrisGame parent class will also only know how to execute its own variant of the game and not of others, promoting separation of concerns in line with the Single Responsibility Principle.

Explanation on Cardinalities

Relationship	Final Cardinality	Ruled Out Cardinality	Justification
Player to Move	1 to 1	1 to 1..2	Originally, in our domain model, we have stated that one Player entity will create one to two Move entities. However, after thinking of how this will be implemented within the game itself, the Player will now only have, technically, one Move in a turn. What about the situation where the player forms a mill after moving a token and therefore can also make the RemoveToken move? Would that not count as another move? This will be further touched on in the explanation of our use of the State Design Pattern in the next section but as a brief overview, the RemoveToken move is instantiated by any of the other three concrete Move classes when the mill-forming conditions are met and within the RemoveToken class, the initial, previous Move will be returned to the Player. In a nutshell, the Player class will only know about executing one Move without knowing that the RemoveToken move is executing if the mill-forming conditions are met.
MorrisGame to Player	1 to 2	1 to 1..2	While the game can be played in single-player mode or multiplayer mode due to our selected advanced requirement of implementing a computer, one MorrisGame will always have two Player instances instead of having either one or two Player instances. The latter situation (when there are either one or two Player instances) would mean that the computer would not be implemented as its own concrete class but a method invoked from somewhere, most likely in the MorrisGame class. As justified in the first key topic in the first section, this design choice is not ideal and instead implementing an abstract class would prove to be much more beneficial. With the implementation of an abstract Player class and concrete HumanPlayer and ComputerAI child classes to inherit from it, the MorrisGame will always have two instances of Player. In fact, the MorrisGame should not know how to differentiate the execution when there is one or two players in the first place and should ideally focus on just running the game loop until the game ends.

Applied Design Patterns

Design Pattern	Justification
Command	As planned from our Sprint One, we have designed the main game logic revolving around the “Move”s a player can make based on the Command Design Pattern. The Command design pattern is implemented in the Player (Invoker), Move (Command) and MorrisBoard (Receiver). The Move abstract class will have methods that are common to the child classes that can be called to execute the command. The Moves are validated in their own class and then received by the MorrisBoard to be executed. The Command design pattern is extremely useful as when Player invokes a type of Move, each Move class does not need to send their logic to the MorrisBoard directly but instead, the Move abstract class packages for all of the details within these classes so MorrisBoard would not need to know about the inner workings for these Move subclasses. Hence, only the Move abstract class will have a dependency relationship with the MorrisBoard instead of having dependencies to all of these concrete Move child classes. With this Design Pattern, decoupling of classes that invoke and perform actions while ensuring the details of each command executed are hidden can be achieved, achieving the goal of Separation of Concerns which is aligned with the Single Responsibility Principle. Aside from that, we can introduce more Move classes and change existing Move classes without affecting other classes, achieving the Open Closed principle.
State	The State design pattern can be seen in the Player, Move and Move child classes. Instead of having all the moves located in the Player class, each move is extended from the parent class Move which has different validity checks and actions that it will perform. The State design pattern is extremely useful when there are many different kinds of states (moves) and if there are differences in behaviour based on the current state. It also helps prevent the creation of a massive conditional which alters how the Player behaves based on different conditions. These states could all be extended from the base State (Move) abstract class. This abstracts away the details from the Player class as the Player class only needs to know the methods in the Move abstract class. With this design pattern, we can add new Moves or change existing Moves independently without affecting other classes which follow the Open Closed principle. In addition, it also follows the Single Responsibility Principle where each Move class only knows its own validity check and actions it needs to perform and the Player class only knows about the Move abstract class and not the child classes.

Design Pattern	Justification
Observer	Originally, we did not plan to include the Observer design pattern as stated in our Sprint One but after thinking it through we decided to include it in our implementation as we believe the combination of these design patterns working together would yield benefits to our application. The Observer design pattern can be seen in the MorrisGame, IMorrisGameSubscriber and GameScreenController classes as it notifies all the GameScreenControllers to update the GameScreen if there is an update in the game. The update in the game may be one of the Moves in the class diagram. The Observer Design Pattern is extremely useful when there are many GameScreens subscribed to the MorrisGame. This can happen when the game is online and there are many subscribers spectating the game. This allows the MorrisGame to notify any moves on the board easily to all the subscribers. It also allows spectators to unsubscribe from a MorrisGame and subscribe to another MorrisGame to spectate the other game instead. With this design pattern, we adhere to the Open Closed principle as we can add more subscribers without changing the MorrisGame code. The subscribers may not be notified in a specific order, but it does not make any difference for our game, so long as everyone is looking at the same game board.

Feasible Design Patterns that were Ruled Out

Design Pattern	Justification
Iterator	A feasible design pattern would be the Iterator design pattern. With this design pattern, in every turn, the game loop would employ the use of an iterator to iterate over an iterable collection, which in our case would be a collection of concrete Move classes, to figure out which Move should be executed during that turn. While, with this design pattern, we have the opportunity to reduce the number of dependencies within our application as the child Move classes will not know about each other as in our current implementation, we believe that applying this design pattern would be a little overkill for the purposes of our application as we only work with a simple collection of Moves. Our current implementation which is based on the State Design Pattern is also much more direct in going through the Move child classes compared to using the Iterator Design Pattern even though there is a tradeoff in terms of the amount of dependencies.
Mediator	Another feasible design pattern would be the Mediator design pattern. With this design pattern, there will be a mediator class which all of the concrete Move classes know and within it, there will be a validity check for different conditions involving these classes to determine which move gets executed within a turn in the game loop. While this implementation is feasible, we collectively highly dislike this type of implementation as the Mediator class acts as a God class as it will know all of the classes it is supposed to mediate, leading to a high number of compositions which is a strong type of association with the highest coupling. There will also be a huge if-else block statement in order to check for the different possibilities of these moves which would make it more difficult for us as developers to refactor in the future. Our current implementation performs the validity checks within the child Move classes which avoids the need of the huge if-else block statement while promoting separation of concerns as none of these classes would evolve into a God class.

References

Nirajrules. (2011, July 17). *Association vs. Dependency vs. Aggregation vs. Composition*. Niraj Bhatt - Architect's Blog. https://nirajrules.wordpress.com/2011/07/15/association-vs-dependency-vs-aggregation-vs-composition/

What is the difference between association, aggregation and composition? (n.d.). Stack Overflow. https://stackoverflow.com/questions/885937/what-is-the-difference-between-association-aggregation-and-composition

Refactoring Guru (2023, January 1). *Command*. https://refactoring.guru/design-patterns/command

Refactoring.Guru. (2023b, January 1). *State*. https://refactoring.guru/design-patterns/state

Refactoring.Guru. (2023b, January 1). *Observer*. https://refactoring.guru/design-patterns/observer

Refactoring Guru (2023, January 1). *Iterator*. https://refactoring.guru/design-patterns/iterator

Refactoring Guru (2023, January 1). *Mediator*. https://refactoring.guru/design-patterns/mediator