MONASH University

# FIT3077
# SPRINT 2: INITIAL DEVELOPMENT

Done By: The Dev Dynasty

*Shoumil Guha (32700660)*

*Rachit Bhatia (32638396)*

*Tan Jun Yu (32025130)*

## UML CLASS DIAGRAM

# SPRINT 2

1. Move, Millchecker

2. Implementation of NeighbourFinder interface

3. Bidirectional relationship between Token and IntersectionPoint classes

4. Player, Move

5. Enumeration, Player

6. Millchecker, Token

7. Usage of singleton design

- **Explain two key classes that you have debated as a team, e.g. why was it decided to be made into a class? Why was it not appropriate to be a method?**

- **Explain two key relationships in your class diagram, e.g. why is something an aggregation not a composition?**

# DESIGN CHOICES RATIONALE

## 1. KEY CLASSES

### a) Move Abstract Class

The Move class is one of the most important classes in the whole system as it is responsible for every prominent action made in the game.

### b) MillChecker Concrete Class

Bidirectional relationship between Token and IntersectionPoint classes

A bidirectional association relationship is used between the Token and IntersectionPoint class since the Token has an attribute of IntersectionPoint while the IntersectionPoint also has an attribute of Token. The main reason for using a bidirectional relationship is to be able to track or reference back to the previous IntersectionPoint that a particular Token was in so that it can be removed from the old IntersectionPoint whenever this Token is moved to a new IntersectionPoint. In other words, if only a single direction association was used from the IntersectionPoint to Token, it is feasible to add this Token to the new IntersectionPoint but it can be difficult to track back the IntersectionPoint that the Token was previously in to remove this Token instance from it. Hence, whenever a Token is added to an IntersectionPoint, the Token needs to be added to the attribute of IntersectionPoint where it belongs to and the corresponding IntersectionPoint also needs to be added to the attribute of Token.

- **Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?**

  Inheritance between Player with HumanPlayer and ComputerPlayer

  Player class is inherited by both HumanPlayer and ComputerPlayer because it is known that these two child classes have common attributes and methods with one another besides having some methods of their own. For instance, both the child classes will have an ArrayList of Tokens and CurrentStateOfMove as attributes alongside the methods of getters and setters. This allows the same code to be reused without having to violate the Don't Repeat Yourself(DRY) principle. Besides that, Player class is declared as an abstract class since it will not be instantiated and most importantly the child classes which are HumanPlayer and ComputerPlayer have a common method that is the setPlayerTurn() but with different operations or implementations and therefore requires the method to be abstract. The setPlayerTurn() of the HumanPlayer is allowing the tokens to be moved by dragging them while the setPlayerTurn() of ComputerPlayer will be used to call the generate random move methods in its own class.

  Inheritance between Move with FlyingMove and SlidingMove

Move class is inherited by both FlyingMove and SlidingMove because it is known that these two child classes have common attributes and methods with one another. For instance, both of the child classes will have Token, xCoordinate,yCoordinate,offSetX and offSetY as attributes alongside the mouseEvents methods. Both the mouseDragged and mouseReleased will have the exact same implementation between FlyingMove and SlidingMove and therefore they are placed in the parent class Move itself. This allows the same code to be reused without having to violate the Don't Repeat Yourself(DRY) principle. Only mousePressed has different implementations and thus it is placed in the child classes themselves.

An alternative to using an inheritance would be having an if-else statement differentiating the implementations of the mousePressed method depending on Flying or Sliding without using an inheritance since mousePressed is the only method with different implementations.In that case, FlyingMove and SlidingMove will not be classes on their own because these two child classes do not have any other attributes or methods of their own.However, considering the extensibility of the code, it is decided that FlyingMove and SlidingMove should be created as separate classes inheriting the Move abstract class  so that it allows new features or implementations to be added with ease in the future.

- **Explain how you arrived at two sets of cardinalities, e.g. why 0..1 and why not 1…2?**
- **Explain why you have applied a particular design pattern for your software architecture? Explain why you ruled out two other feasible alternatives?**