# Lab Report 4

1. *[8pts]* Provide a class diagram containing all the classes in your submission using either UML or the informal UML-like notation used in class (ensuring that interfaces are distinguishable from concrete classes). Additionally, provide a visual indicator showing which classes belong to the model, view, and controller.

   Include the Flutter `StatelessWidget` class, but exclude all other Flutter classes (.e.g., `ListenableBuilder`). Include all **public** fields and methods except for those of the Flutter `StatelessWidget` class.

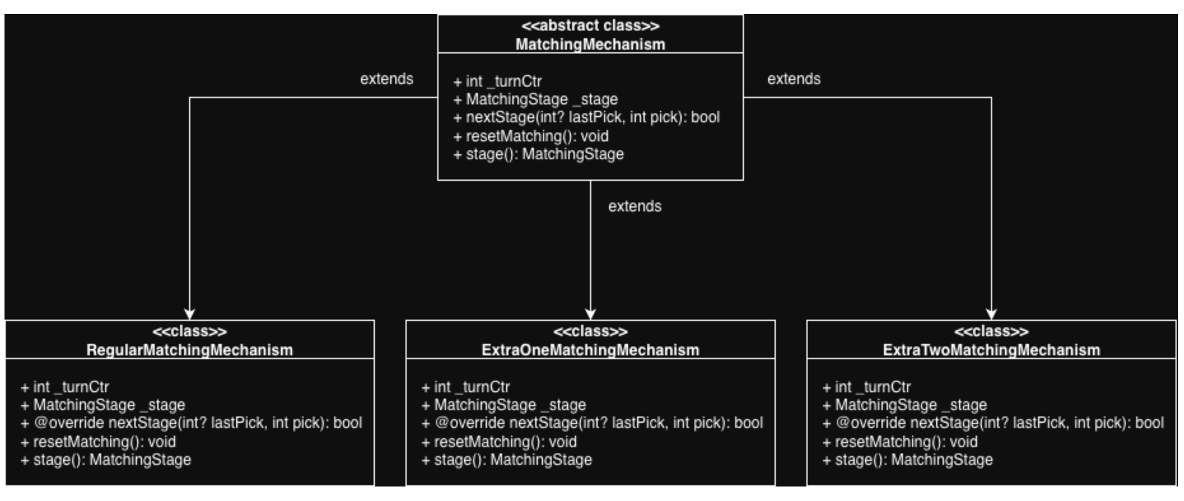
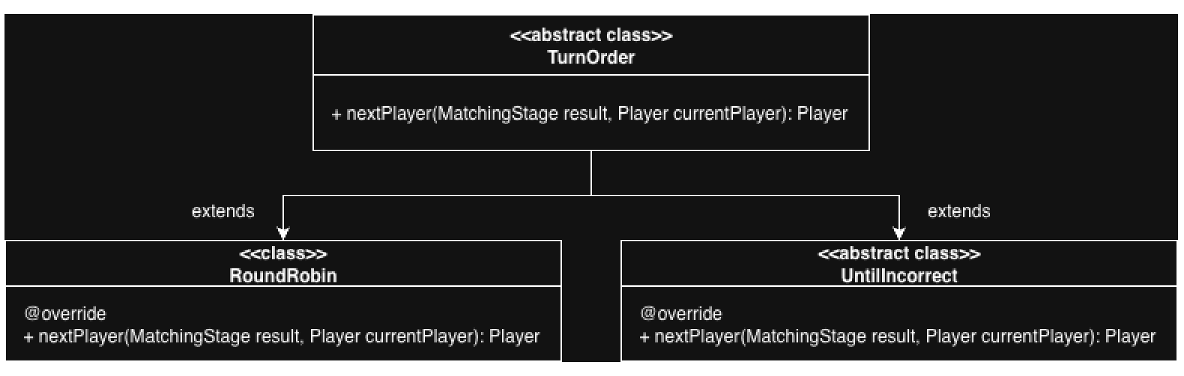


Figure 1: MatchingMechanism classes.



<<abstract class>>
TurnOrder
+ nextPlayer(MatchingStage result, Player currentPlayer): Player
extends
extends
<<class>>
RoundRobin
@override
+ nextPlayer(MatchingStage result, Player currentPlayer): Player
<<abstract class>>
UntilIncorrect
@override
+ nextPlayer(MatchingStage result, Player currentPlayer): Player

Figure 2: TurnOrder classes.

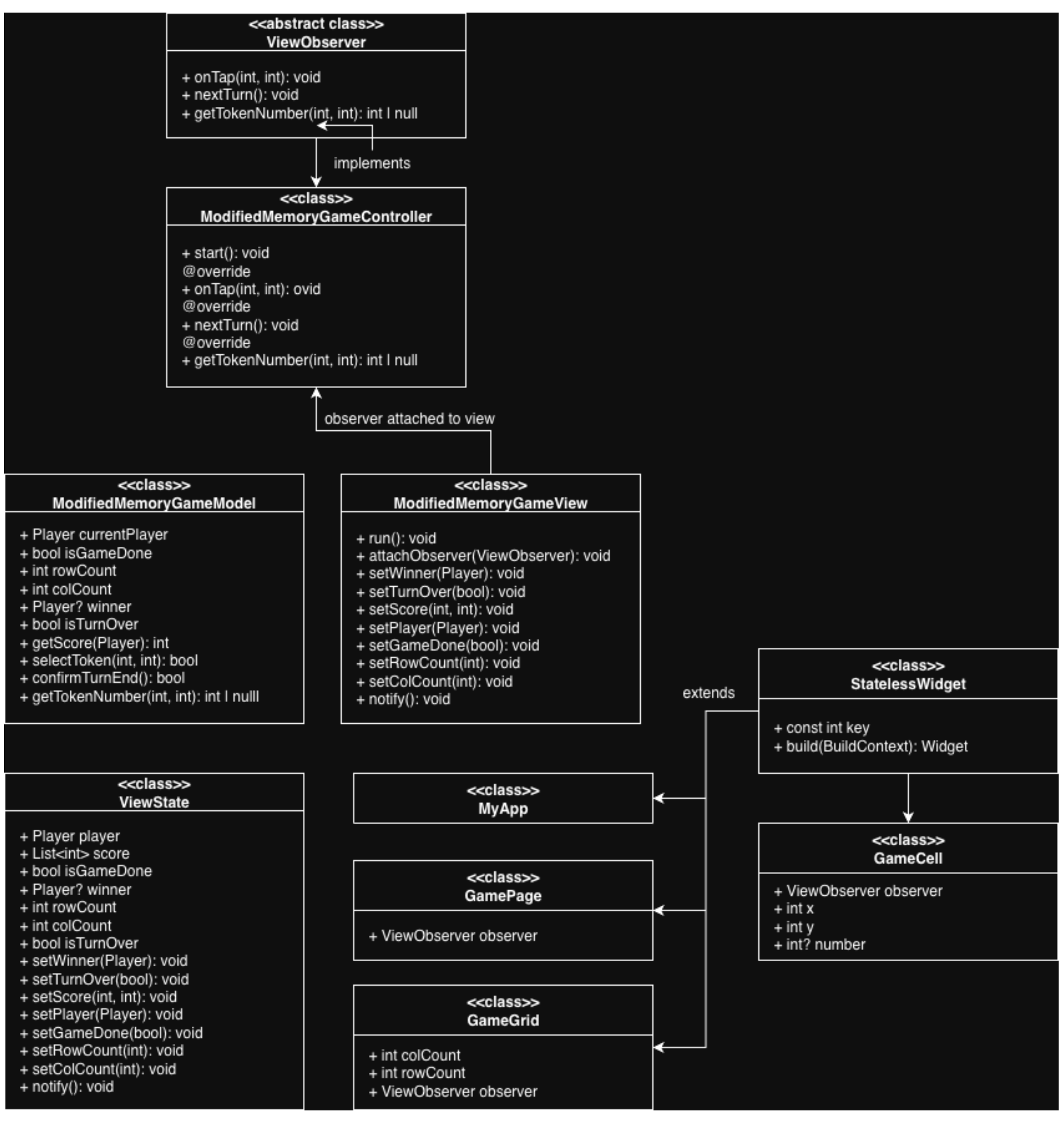


Figure 3: MVC classes.

2. *[12pts]* Show how your code follows the Open-Closed Principle using composition for the following scenarios by providing working code snippets that implement them and showing how `ModifiedMemoryGameModel` should be instantiated to accommodate them (do not add them your .dart files; no need to maximize code reuse; no need to change `MatchingModeTag`,`TurnOrderTag`, and `make`).

   (a) *[6pts]* A new **Pick 4** matching mode variant is introduce in which tokens are matched in groups of **four** where each token in the grid has a number shared by exactly three other tokens; players select **four** distinct flipped-down tokens to flip up per turn.

   Also explain how the `ModifiedMemoryGameModel` is able to properly throw an `"Invalid grid"` string for an invalid grid given the Pick 4 matching mode without any modifications to existing code.

   **—ANSWER—**

   Because the `model` is constructed purely by passing in any class implementing the `MatchingMechanism`, we can easily extend the functionality as seen in the `Pick4` class below. It satisfies all the conditions of the abstract class, and hence can be used just as we would for all the other matching mechanisms. Additionally, invalid grid still works because the grouping is self-contained, so whatever the user decides to put into the grouping will be used to verify whether or not the grid is valid.

```
abstract class MatchingMechanism {
  int _turnCtr = 0;
  int _grouping = 1;
  MatchingStage _stage = MatchingStage.matching;

  void resetMatching() {
    _turnCtr = 0;
    _stage = MatchingStage.matching;
  }

  bool nextStage(int? lastPick, int pick);

  MatchingStage get stage => _stage;
  int get grouping => _grouping;
}
```

Figure 4: `abstract class` for matching.

```
class Pick4 extends MatchingMechanism {
  Pick4() {
    _grouping = 4;
  }

  @override
  bool nextStage(int? lastPick, int pick) {
    _turnCtr += 1;
    if (_turnCtr < 4) {
      if (_turnCtr == 1 || lastPick == pick) {
        _stage = MatchingStage.matching;
      } else if (lastPick != pick) {
        _stage = MatchingStage.failed;
      }
      return false;
    } else {
      if (_stage == MatchingStage.matching) {
        if (lastPick == pick) {
          _stage = MatchingStage.success;
        } else {
          _stage = MatchingStage.failed;
        }
      }
      return true;
    }
  }
}
```

Figure 5: implementation of `pick4` matching mechanism

   (b) *[6pts]* A new **Until Correct** turn order variant is introduced in which a player who finds does **not** find a match will be granted another turn; turns granted by this rule would also be eligible for granting yet another turn if a match is still not found (i.e., a player would repeatedly be granted turns until an correct guess is made).

   **—ANSWER—**

   Similar sentiment with (a), because we are only passing any class that implements the `TurnOrder` abstract class, any class which satisfies its conditions will integrate into the model. Hence, no extra code need be added.

```
abstract class TurnOrder {
  Player nextPlayer(MatchingStage result, Player currentPlayer);

  Player changePlayer(Player player) {
    if (player == Player.p1) {
      return Player.p2;
    } else {
      return Player.p1;
    }
  }
}
```

Figure 6: `abstract class` for turn ordering.

```
class UntilCorrect extends TurnOrder {
  @override
  Player nextPlayer(MatchingStage result, Player currentPlayer) {
    if (result == MatchingStage.failed || result == MatchingStage.matching)
{
      return currentPlayer;
    } else {
      return changePlayer(currentPlayer);
    }
  }
}
```

Figure 7: implementation of `untilCorrect` turn-ordering.