SYST 17796 DELIVERABLE 2
DESIGN DOCUMENT TEMPLATE

Group 1 – The Bronze Medalists

OVERVIEW

1. Project Background and Description

As discussed in Deliverable 1, the aim of the project is to develop an offline virtual simulation of the card game Coup, by Indie Board Games. As per the rules of the game and the design decisions determined for this project, our application will perform the following. An abbreviated version of this can be found in the use case narrative given in section 4

At initiation, the game will provide a brief overview of the rules, after which it will register a user-specified number of players between 3 and 6. The game will then request a username from each player and register them for the game. Once all players are registered, a round will begin. If no prior rounds have been played, a random player will be selected as the starting player, otherwise the winner of the previous round will start. Turn order will proceed from the starting player in the order that players registered (i.e. the player who registered after the starting player will go next and so on). Each player is given two cards (with their associated effects) in their hand corresponding to two influence points, and two coins.

At the start of each turn, the active player's name is shown in addition to a small guide, each player's public information (coins and influence) and a list of discarded cards. The active player is then prompted to select an action to declare. If the effect is valid and the player has enough coins to perform the action, the action is now declared (otherwise the player is prompted to re-enter input). If an action targets another player, the active player must also enter the name of the player to target. At this stage, if the action is refutable, each player is asked if they would like to challenge the action. If any player challenges the action, the active player reveals if they were bluffing, and if so, is required to lose one influence point and discard a card from their hand. Additionally, the declared effect is cancelled (including any expenditure of coins) and the player's turn immediately ends. If the active player was not bluffing, the challenging player loses one influence point and a card from their hand, and the declared effect continues.

If no player challenges, or a challenge is unsuccessful, an unblockable action is executed and its effect is processed. If the action is blockable, the target player is asked if they would like to attempt a block (in the case where the active player declares Foreign Aid, any player may choose to block). Should they choose to do so – and since, like actions, blocking actions have specific character associations – the active player may issue a challenge which will proceed in a similar fashion to the process described above. If the challenge is not successful, in addition to losing influence, the active player's declared effect is blocked (requiring them to pay any coins needed for the action) and the turn ends immediately.

If the target player does not block, or a block is successfully challenged, the declared action is executed, and its effect is processed. At the conclusion of each turn, any player whose influence fell to 0 (or below) is removed from set of living players and will no longer receive a turn for the remainder of the round. The next living player is then set to the active player and proceeds with their turn.

If at the end of a turn, there is only one player remaining in the set of living players, that player is declared the winner of the round and their score is incremented by one. If the user decides to play another round, all registered players are returned to the living players set and the round initiation process is executed. If the user decides not to continue at the end of a round, a list of the players ranked by their score is shown and the program exits. A full summary of the features can be found in the Use Case diagram shown in Figure 1.

2. Design Considerations

Our class diagram as shown in Figure 2 outlines the overall structure of our project elements. Below, we discuss the design principles considered and implemented in the architecture of our code.

- Encapsulation: The principle of encapsulation is consistently used throughout our program to ensure that the properties of any given entity are not accessed or modified improperly. Two key examples include the Effects class and its children wherein key properties of the effect (such as whether or not in can be challenged or blocked) are encapsulated and only assigned accessor methods since there should be no reason to modify these properties (effects are unchanging). Similarly, the Player class encapsulates all properties of the player and assigns setters and getters (with the appropriate return type matching the property) only where appropriate. Lastly, the Deck class which holds the Card objects in the active deck and discard pile is encapsulated to ensure that addition and removal of cards is done predictably and under meaningful controls.

- Delegation: The program is structured in such a way as to ensure each class is delegated its appropriate tasks, with cohesion (discussed below) in mind to ensure that the scope of the class's tasks are consistent. The App class is responsible for the flow of the game while the Game class is responsible for managing interactions between different elements and actions. Other classes represent a particular element of the game. Finally, the Tools class is responsible for performing common tasks unrelated to elements of the game such as validating user input and showing messages and prompts.

- Cohesion: Our codebase is designed to ensure each class is fully and solely responsible for its respective entity, with the overall goal of creating a set of classes which are mutually exclusive and collectively exhaustive in representing the game and its elements. Our Player class contains properties and methods which exclusively relate to representation and modification of an individual player's state in the game (the player's name, number, coins, hand, influence, status and score), while our Card class is given

- Coupling: We have designed our code in such a manner as to minimize tight coupling as much as possible. Effects which modify player states simply call the relevant Player method with the appropriate parameters in order to execute the effect. The execution of the effect is entirely independent of how the player state modification occurs, allowing for updates to be made to player methods with no requirement to update the execution of the effects themselves. Similarly, our main program simply calls the execute() method from the relevant effect rather than depending on its specific implementation in order to run properly.

- Inheritance: Where different entities share several characteristics, our code utilizes inheritance to reuse code and ensure consistency in representation between multiple related entities. In particular, the seven effects used in the game all extend an abstract Effect class which defines the properties required for an effect, how that effect is to be constructed, and an abstract execute() method which is implemented in each of the Effect class's children to be called when that particular effect should be performed.

- Composition: Our program uses composition to develop more complex entities from simpler parts. In our class diagram, the deck is composed of between 3 (the minimum count of cards that would be in the deck if all players held two cards in their hand in a six-player game) and 15 cards (the total number of cards in the game). Additionally, the Game class, which controls the procedure of the program, is composed of between 3 and six Players, 7 Effects, and 1 Deck.

- Flexibility/Maintainability: By applying all of the principles described above, our code retains a high degree of flexibility and maintainability. Encapsulation ensure that bugs related to improper access of class properties are kept to a minimum, while delegation, cohesion and loose coupling reduce the need to make significant changes to multiple classes when modifying or updating the code. Utilization of inheritance allows for reuse of code and a centralized location to modify when changing common behaviours (the parent class) while

also simplifying more specific modifications by isolating these to a given child class. Lastly, composition allows us to modify or extend our codebase in a part-wise manner using small, focused approaches on particular elements rather than requiring that an entire whole be extensively modified for each incremental update. The flexibility of our code is of course best demonstrated by the numerous updates and modifications we have been able to make throughout the development lifecycle, showing that it does indeed achieve the primary goals of the aforementioned design principles.

3. Figures

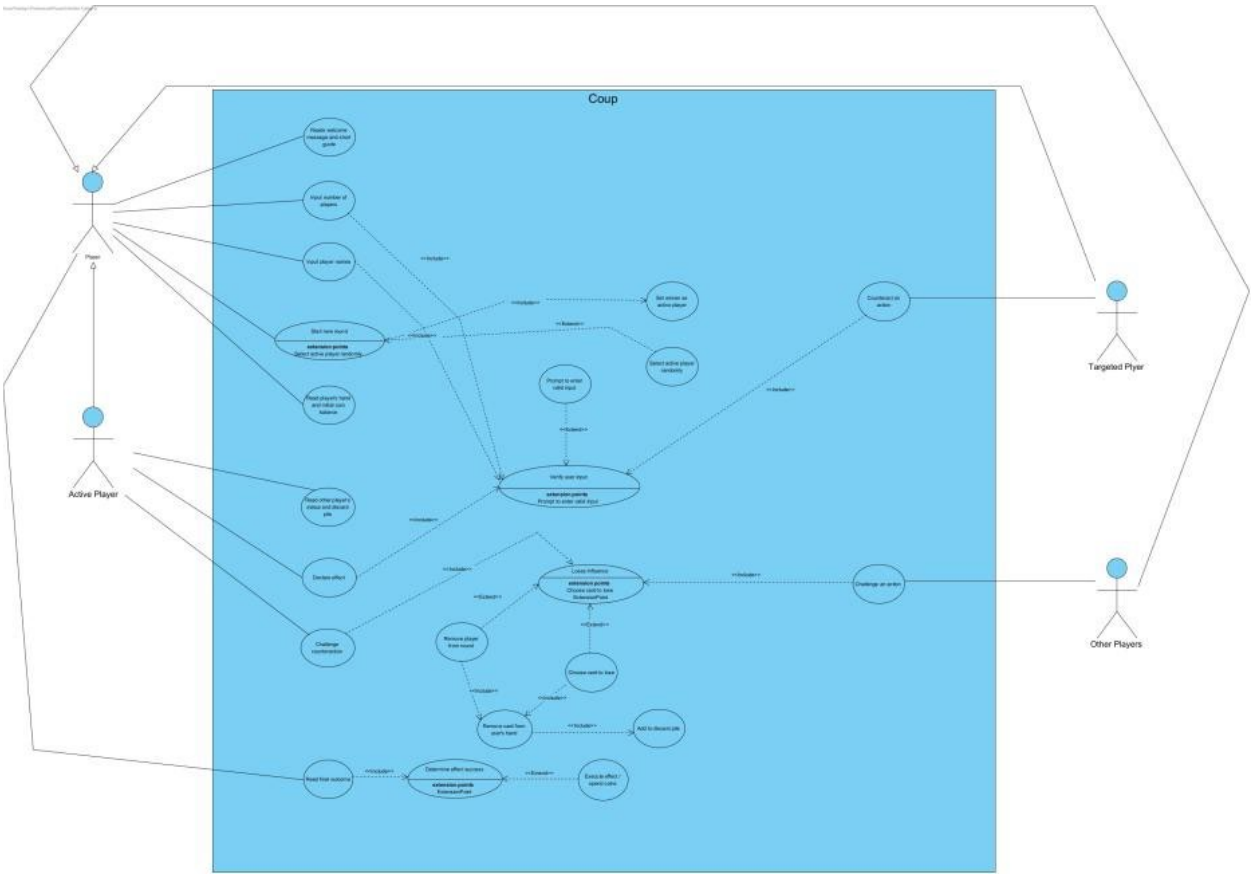

Figure 1: UML Use Case Diagram

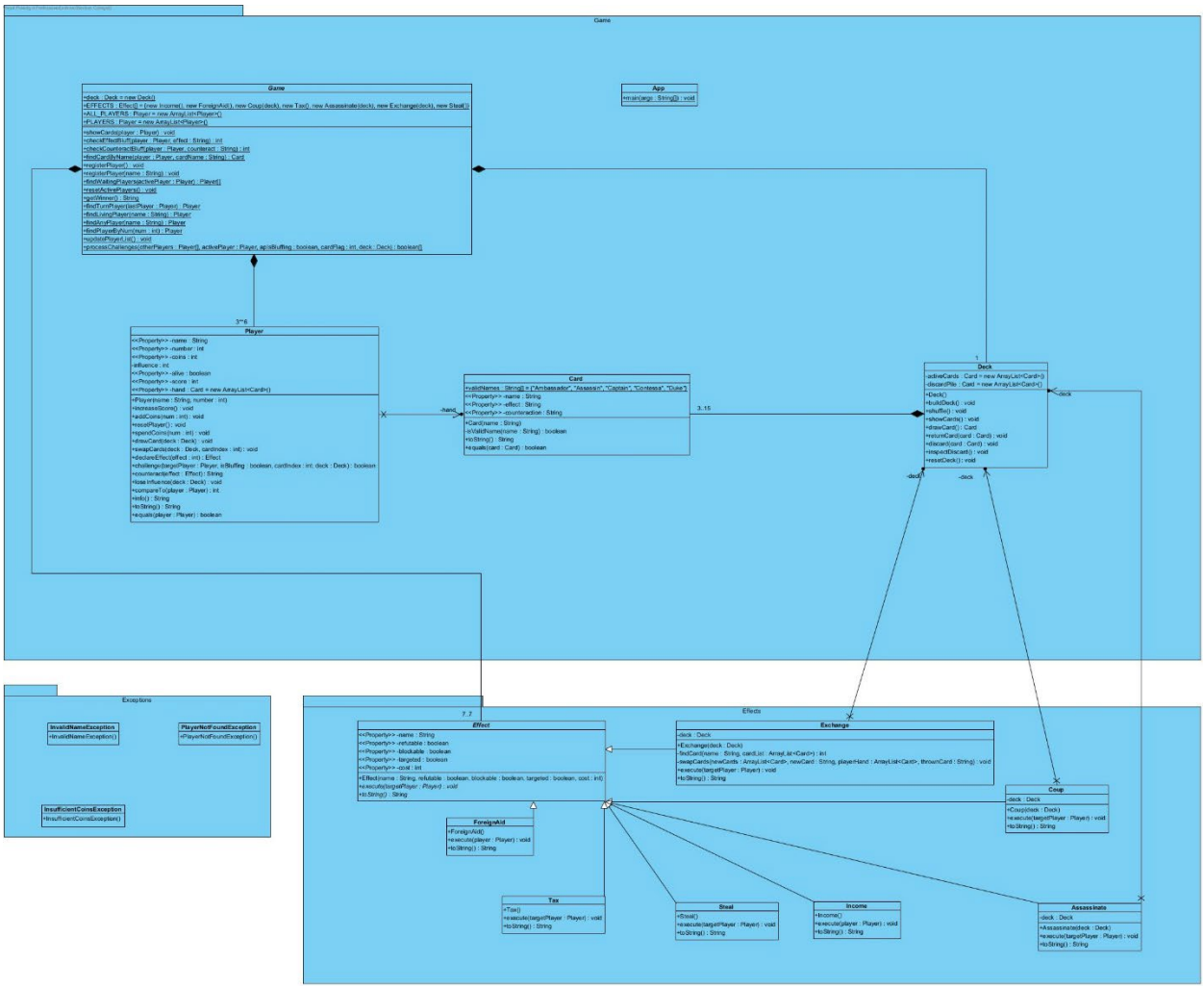

Figure 2: UML Class Diagram

4. Use Case Narrative

Use-case steps:

1. Players read welcome message and short guide.

2. Players input number of participants.

    2.1. Verifies User Input.

        2.1.1. Prompt to enter valid input.

3. Player starts new round.

    3.1. Set winner as the active player.

    3.2. Select active player randomly.

4. Read player's hand and initial coin balance.

5. Active player reads other player's status and discard pile.

6. Active player declares effect.
    6.1. Verify user input.
        6.1.1. Prompt to enter valid input.
    6.2 Targeted player can counteract the action.
        6.2.1 Active player can challenge counteraction.
            6.2.1.1. Reduce influence for loser of challenge.
    6.3. Other players can challenge the action.
        6.3.1. Reduce influence for loser of challenge.
7. Players read the final outcome.
    7.1. Determine effect success.
        7.1.1 Execute effect/spend coins.
8. Turn End
    8.1. Check round end.
    8.2. Remove player with zero influence from game.