SWEN30006 Assignment 2 - Report

Workshop 09, Team 02
Lucas Fern & Cameron Maddern

May 28, 2021

This project required changes to be made to an existing Cribbage card game trainer system to add scoring and logging functionality. These additions were made with the use of a variety of design patterns in order to have minimum impact on the existing code, and therefore reduce coupling between existing classes and classes added for the new functionality.

This report will discuss the changes which have been made, provide justification for the design patterns used, and argue in favour of these design patterns over the use of others where appropriate. The additional classes will be covered first, then how these additional classes were integrated into the existing system.

A class diagram of the updated design is included at the end of the report and will be relevant to visualise all changes to the system.

1 Cribbage Observers

In an effort to minimise coupling between added classes and maximise cohesion within them, a `CribbageObserver` interface was implemented. This allowed classes to subscribe to various events of the Cribbage game, and respond with unique actions. This required the addition of a list of subscribers in the main `Cribbage` class, as well as a method to register new subscribers.

1.1 Cribbage Events

Now that a publish-subscribe pattern is implemented it must be decided when events are broadcast to the subscribers. There are a variety of events in the game of Cribbage, and it was decided that all events which relate to the logging and scoring functionality will be broadcast to the subscribers. This allows - for example - the logging functionality for the `deal` to be implemented by an observer responding to a `Deal` event. The various subscribers to Cribbage Events are discussed further in Section 1.2.

The complete list of `CribbageEvent`s implemented appears on the right side of the class diagram, and their functionality is briefly summarised below:

- `SetSeed`

  Subscribers are notified of this event when the game's seed is set from the `cribbage.properties` file. The event contains the random seed which was chosen.

- `InitPlayer`

  This event is broadcast when the players of the game are initialised and assigned a number. This provides subscribers with information about the player type which has been initialised (*eg.* `cribbage.RandomPlayer`) and their number.

- `Discard`

  This event is created once for each player each game, when the player has selected which cards to discard. It provides observers with the players number, and a `jcardgame.Hand` object containing the cards they selected for discarding.

- `PlayStarter`

  The `PlayStarter` event is published once per game when the starter card is selected, and provides the starter card as a `jcardgame.Card`.

- `Play`

  This event is raised on every turn of the game, when a player selects the card they wish to play. It contains the player's number, the card they played, and the total face value of the current board after adding this card.

- `Show`

  The `Show` event occurs at the end of the game when play has ceased and players are showing card combinations from their hands to be scored. This event contains information about the player's number, the starter card, and the cards that they are showing to be scored.

- `Score`

  The `Score` event is a special type of event raised by the Cribbage Scorer in response to other events, it occurs whenever a player's score is incremented, and contains data on the amount of points scored, their total score, and the type of score achieved. This will be discussed further in 1.2.1.

Each of these events also overrides the default `toString()` method. Their implementation of the method returns a `String`, formatted as required for the logging functionality. This will be discussed further in Section 1.2.2

1.2 Cribbage Subscribers

Subscribers are classes which implement the `CribbageObserver` interface and register with the `Cribbage` class to be notified of game events. Their creation and registration is done inside the `Cribbage` class since currently the only subscribers are the Logger and Scorer, which was not complex enough to justify their creation in an external factory.

1.2.1 Cribbage Scorer

The `CribbageScorer` is a singleton class which is cohesively responsible for handling all scoring events in the Cribbage game. `CribbageScorer` subscribes to the `Cribbage` class and its `update()` method is called whenever an event (defined in Section 1.1) occurs.

When an event is received, the `CribbageScorer` updates its records of the moves in the game and delegates responsibility for the scoring of events out to individual score handling classes, which each implement the logic for a single method of earning points. Since there are many scoring methods, these are instantiated by a `ScorerFactory` when the `CribbageScorer` is created. The score handling classes are discussed below.

Score Handlers implement the `ScoringEvent` interface, which requires that they define the following methods:

```
int scoreForPlay(Hand cardSet, int playerScore, int playerNum);
int scoreForShow(Hand cardSet, int playerScore, int playerNum);
```

The distinction between the two methods `scoreForPlay()` and `scoreForShow()` allows for variations in scoring to be easily implemented, since there certain behavioural differences between scoring events during and after the Play stage of the game. For example, flushes are only relevant in the show stage, so do not calculate a score in their `scoreForPlay()` method.

This design also allows for more complex variations, for example, if an alternative ruleset did not include flushes then this would be implemented by not subscribing the `HandleFlushes` class to the `CribbageScorer`, or if a variation were to not include runs in the Show phase of the game, then `scoreForShow()` in the `Handleruns` class could simply return the `playerScore` that was passed in.

The complete list of `ScoringEvent`s implemented appears on the bottom left side of the class diagram, and their functionality is briefly summarised below:

- `HandleTotals`

  This `ScoringEvent` determines points allocated for any combinations of 2 cards who's scores sum to 15 as well as for when total face values reach 15 or 31. The amount of points allocated for this is customisable with the `POINTS_FOR_TOTALS` constant.

- `HandleFlushes`

  This `ScoringEvent` determines points allocated for any combination of 4 or 5 cards of the same suit shown in the Show phase. Its `scoreForPlay()` method is unimplemented.

- `HandlePairs`

  This `ScoringEvent` determines points allocated for any pair, triple, or 4-of-a-kind. Points allocated for pairs of varied sizes can be adjusted with the `PAIR_SCORE`, `TRIP_SCORE` and `QUAD_SCORE` constants.

- `HandleRuns`

  This `ScoringEvent` determines points allocated for runs of cards, accounting for run sizes from `SHORTEST_RUN` to `min(LONGEST_RUN, handCards.size())` where `handCards` is the card set from which runs are being searched for. This ensures the program does not search for a run greater than the amount of cards available in the given hand.

- `HandleStarters`

  This `ScoringEvent` determines points allocated for the `dealer` if the first card is a Jack as well as finding any scores from Jacks during the show phase.

This design allows for simple extensibility as any new method of scoring can simply be defined as a new class inheriting the `ScoringEvent` interface, and once it is added to the `ScorerFactory` it will be interrogated for score updates after each Play and Show event.

Finally, the logging functionality for scores is enabled by each of the `ScoringEvents`. Once the amount of points scored by an event is determined, the score handler is able to access the singleton `CribbageLogger` instance and provide it with a `new Score` event. By creating a score event with all the required logging information the remaining logging behaviour is all handled inside the `CribbageLogger` class and thus cohesion within the `ScoringEvents` is maximised. The internal behaviour of the `CribbageLogger` will be discussed in detail below.

1.2.2 Cribbage Logger

The `CribbageLogger` is a singleton class, which appears in the bottom left of the class diagram and is cohesively responsible for all of the logging functionality of the game. As a subscriber to the `Cribbage` class, the logger's `update()` method is called whenever an event (defined in Section 1.1) occurs. The event is passed through to the class in this method call.

On notification of any event, the cribbage logger calls the event's `toString()` method, yielding the information required for logging, and uses a `BufferedWriter` to write this to `cribbage.log`. Taking a `Play` event for example, the contents and respective string representation might be:

```
examplePlay = {                              |
    eventId: "play",  [String]               |
    playerId: "P0",   [String]               |      String representation:
    totalPoints: 23,  [int]                  |      "play,P0,23,KH"
    card: KH          [jcardgame.Card]       |
}                                            |
```

This highly cohesive design, enabled by the observer pattern, is all that is required for the logging functionality.

2 Changes to Existing Classes

Changing existing classes was avoided as much as possible when adding the new functionality to reduce the possibility of bloated classes, and reduce coupling with new classes. This section will provide justification for the few changes which were made to the existing classes despite these considerations.

2.1 Cribbage

The `Cribbage` class is where most of the changes to existing classes were made. The changes made here are:

- **Adding the Subscriber Functionality**

  Since the majority of additions to the system were made by implementing the observer pattern, an attribute had to be added to the `Cribbage` class to store the list of classes which subscribe to the events. A method to register subscribers was also added so that the `CribbageScorer` and `CribbageLogger` could register themselves to be notified.

- **The Registration of Subscribers**

  Since they are separate classes, the `CribbageScorer` and `CribbageLogger` must be instantiated at the beginning of the game, and registered as subscribers to the `Cribbage` class. It was decided that this was an appropriate amount of code (2 lines) to add to the cribbage class, as opposed to creating a factory to instantiate these classes (which would itself need to be initialised on start-up.)

- **Privacy / Visibility Changes**

  To support the functionality of the new system, it was sensible to change the privacy of certain methods. The most significant change was to each of the 4 `canonical()` methods, which were `private` instance methods in the original design, and have been changed to `public static` methods. They were able to be converted to `static` since the conversion of `jcardgame` objects to `String`s did not strictly require any attributes of the `Cribbage` instance.

  This change was made to support the logging functionality, since the log file contains canonical representations of the cards and hands, and the `String`ification of the `jcardgame` objects occurs in the `toString()`

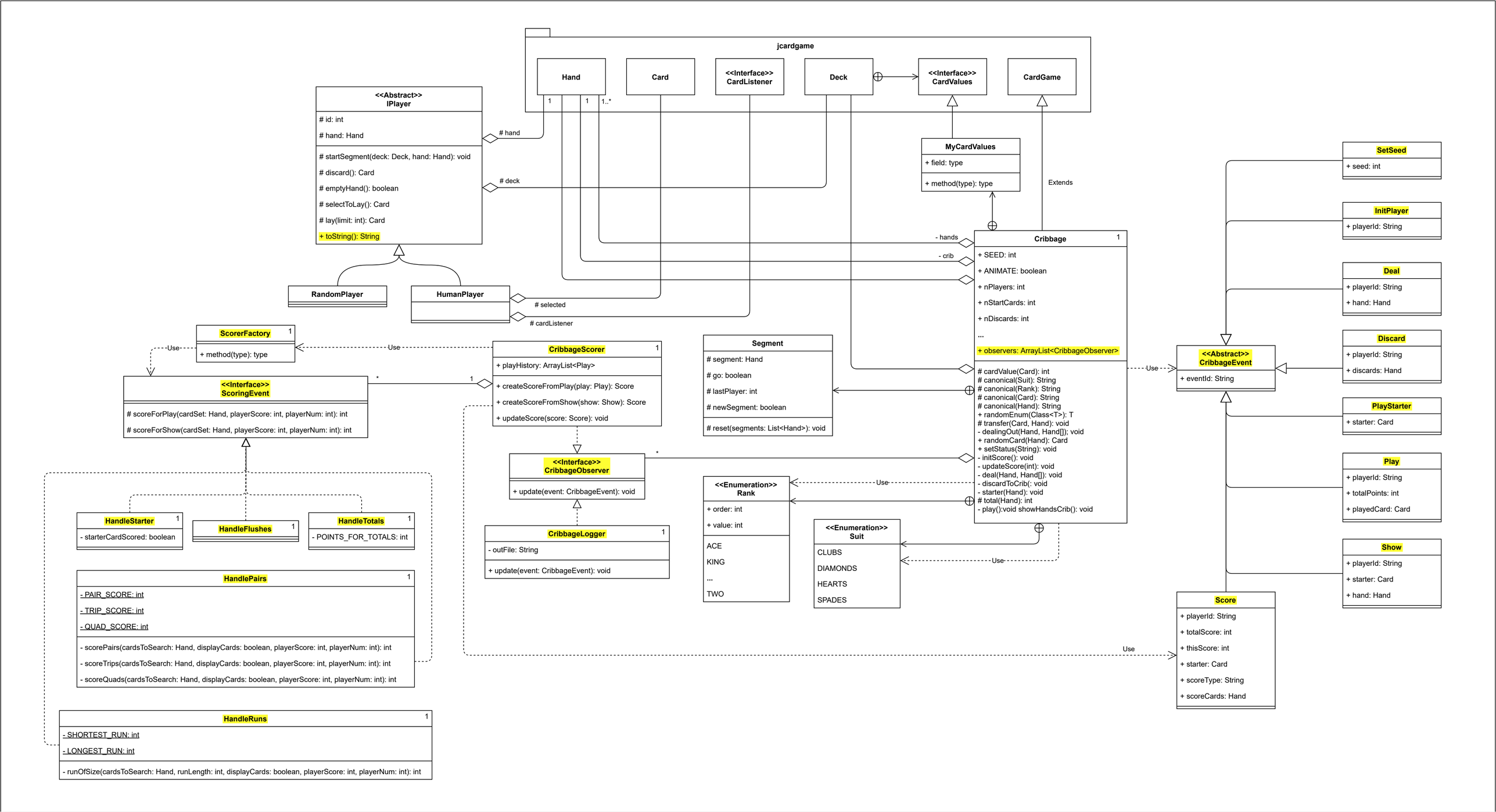
methods of each of the `CribbageEvents`. Considering the alternative would be to pass string representations as well as the original `jcardgame` objects around with each event; and the fact that there is no clear downside to this change, this was an obvious choice.

3 Consideration of Alternative Designs

The final design presented in this report was the product of many design iterations. Originally, designs with high coupling to the `Cribbage` class were considered as alternatives to the final design, which is centred around the implementation of the Observer pattern. These alternative implementations would have involved creating Logging and Scoring classes and calling these directly from the `Cribbage` class when relevant events occurred in the game. This clearly causes enormous coupling between the classes since logging and scoring functionality is complex and required at all stages of the game.

The final design is the antithesis of this highly coupled design, and was chosen for its implementation requiring extremely small changes to existing classes - these being the notifications to subscribers of new events. This design ultimately allows for so much extensibility that future editions of the game - such as an update of the GUI - could subscribe to the events raised by the observer pattern to conduct a variety of other operations (again such as updating GUI elements).

Alternatives were also considered to the implementation of the scoring method. Originally the score handling was done in methods of the `ScoreHandler` class, but it was clear that the complexity of scoring rules and potential for extensibility made it a sensible design decision to extract each scoring rule into its own class and instantiate these within a factory.


jcardgame
Hand
Card
<<Interface>>
CardListener
Deck
<<Interface>>
CardValues
CardGame
<<Abstract>>
IPlayer
id: int
hand: Hand
startSegment(deck: Deck, hand: Hand): void
discard(): Card
emptyHand(): boolean
selectToLay(): Card
lay(limit: int): Card
+ toString(): String
# hand
# deck
MyCardValues
+ field: type
+ method(type): type
Extends
RandomPlayer
HumanPlayer
selected
cardListener
- hands
- crib
Cribbage
1
+ SEED: int
+ ANIMATE: boolean
+ nPlayers: int
+ nStartCards: int
+ nDiscards: int
...
+ observers: ArrayList<CribbageObserver>
# cardValue(Card): int
# canonical(Suit): String
# canonical(Rank): String
# canonical(Card): String
# canonical(Hand): String
+ randomEnum(Class<T>): T
# transfer(Card, Hand): void
- dealingOut(Hand, Hand[]); void
+ randomCard(Hand): Card
+ setStatus(String): void
- initScore(): void
- updateScore(int): void
- deal(Hand, Hand[]): void
- discardToCrib(: void
- starter(Hand): void
total(Hand): int
- play():void showHandsCrib(): void
Use
ScorerFactory
1
+ method(type): type
Use
Use
CribbageScorer
1
+ playHistory: ArrayList<Play>
+ createScoreFromPlay(play: Play): Score
+ createScoreFromShow(show: Show): Score
+ updateScore(score: Score): void
Segment
segment: Hand
go: boolean
lastPlayer: int
newSegment: boolean
reset(segments: List<Hand>): void
<<Interface>>
ScoringEvent
scoreForPlay(cardSet: Hand, playerScore: int, playerNum: int): int
scoreForShow(cardSet: Hand, playerScore: int, playerNum: int): int
*
1
<<Interface>>
CribbageObserver
+ update(event: CribbageEvent): void
*
CribbageLogger
1
- outFile: String
+ update(event: CribbageEvent): void
<<Enumeration>>
Rank
+ order: int
+ value: int
ACE
KING
...
TWO
Use
<<Enumeration>>
Suit
CLUBS
DIAMONDS
HEARTS
SPADES
Use
<<Abstract>>
CribbageEvent
+ eventId: String
Use
SetSeed
+ seed: int
InitPlayer
+ playerId: String
Deal
+ playerId: String
+ hand: Hand
Discard
+ playerId: String
+ discards: Hand
PlayStarter
+ starter: Card
Play
+ playerId: String
+ totalPoints: int
+ playedCard: Card
Show
+ playerId: String
+ starter: Card
+ hand: Hand
Score
+ playerId: String
+ totalScore: int
+ thisScore: int
+ starter: Card
+ scoreType: String
+ scoreCards: Hand
Use
HandleStarter
1
- starterCardScored: boolean
HandleFlushes
1
HandleTotals
1
- POINTS_FOR_TOTALS: int
HandlePairs
1
- PAIR_SCORE: int
- TRIP_SCORE: int
- QUAD_SCORE: int
- scorePairs(cardsToSearch: Hand, displayCards: boolean, playerScore: int, playerNum: int): int
- scoreTrips(cardsToSearch: Hand, displayCards: boolean, playerScore: int, playerNum: int): int
- scoreQuads(cardsToSearch: Hand, displayCards: boolean, playerScore: int, playerNum: int): int
HandleRuns
1
- SHORTEST_RUN: int
- LONGEST_RUN: int
- runOfSize(cardsToSearch: Hand, runLength: int, displayCards: boolean, playerScore: int, playerNum: int): int