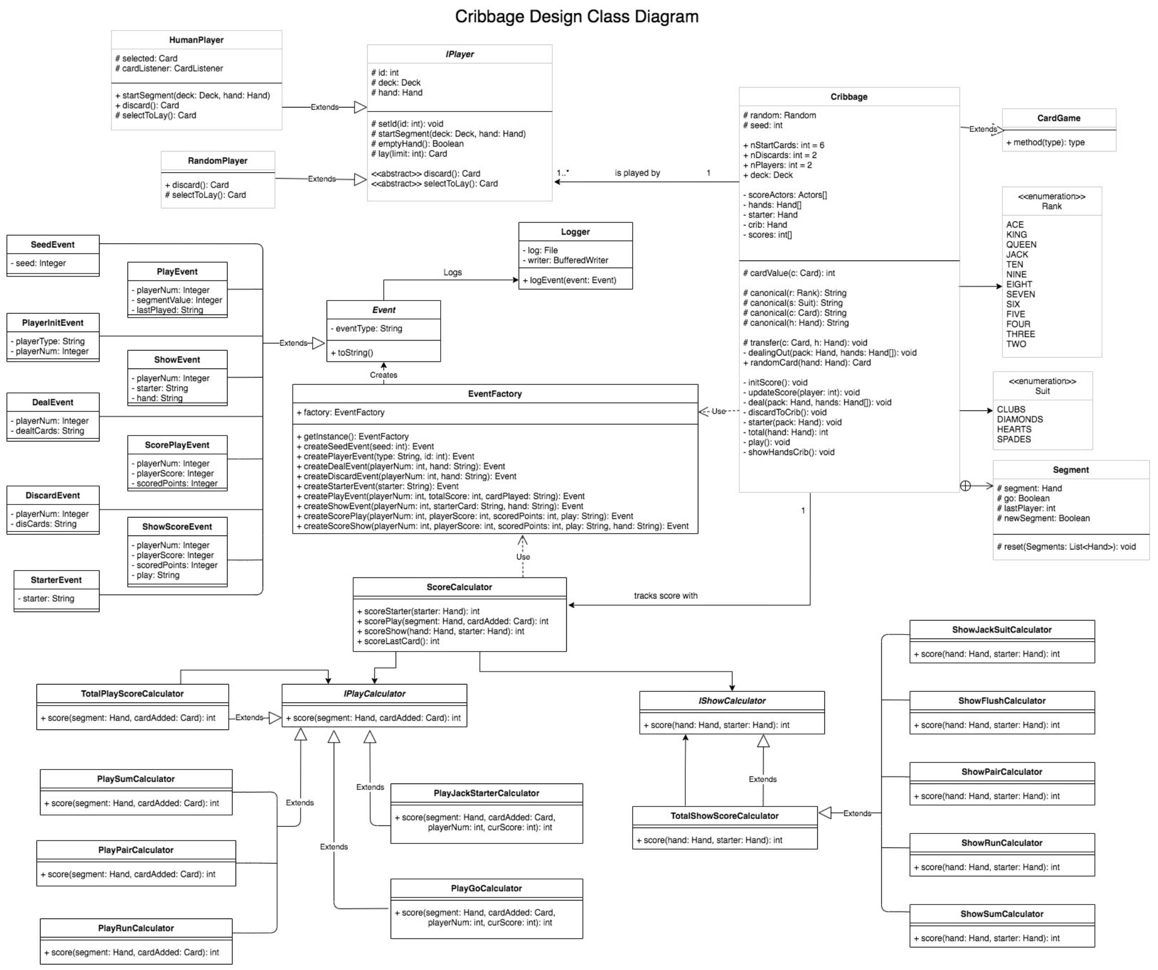
# Cribbage Trainer Design Changes
# SWEN30006 Project 2
**W13 Group 2 - Isobel Byars, Allison Cheng, Eleen Liu**

**28 May 2021**



Cribbage Design Class Diagram
HumanPlayer
# selected: Card
# cardListener: CardListener
+ startSegment(deck: Deck, hand: Hand)
+ discard(): Card
# selectToLay(): Card
IPlayer
# id: int
# deck: Deck
# hand: Hand
# setId(id: int): void
# startSegment(deck: Deck, hand: Hand)
# emptyHand(): Boolean
# lay(limit: int): Card
<<abstract>> discard(): Card
<<abstract>> selectToLay(): Card
RandomPlayer
+ discard(): Card
# selectToLay(): Card
Extends
1..*
is played by
1
Cribbage
# random: Random
# seed: int
+ nStartCards: int = 6
+ nDiscards: int = 2
+ nPlayers: int = 2
+ deck: Deck
- scoreActors: Actors[]
- hands: Hand[]
- starter: Hand
- crib: Hand
- scores: int[]
# cardValue(c: Card): int
# canonical(r: Rank): String
# canonical(s: Suit): String
# canonical(c: Card): String
# canonical(h: Hand): String
# transfer(c: Card, h: Hand): void
- dealingOut(pack: Hand, hands: Hand[]): void
+ randomCard(hand: Hand): Card
- initScore(): void
- updateScore(player: int): void
- deal(pack: Hand, hands: Hand[]): void
- discardToCrib(): void
- starter(pack: Hand): void
- total(hand: Hand): int
- play(): void
- showHandsCrib(): void
CardGame
+ method(type): type
<<enumeration>> Rank
ACE KING QUEEN JACK TEN NINE EIGHT SEVEN SIX FIVE FOUR THREE TWO
<<enumeration>> Suit
CLUBS DIAMONDS HEARTS SPADES
Segment
# segment: Hand
# go: Boolean
# lastPlayer: int
# newSegment: Boolean
# reset(Segments: List<Hand>): void
Logger
- log: File
- writer: BufferedWriter
+ logEvent(event: Event)
Logs
Event
- eventType: String
+ toString()
Creates
SeedEvent
- seed: Integer
PlayEvent
- playerNum: Integer
- segmentValue: Integer
- lastPlayed: String
PlayerInitEvent
- playerType: String
- playerNum: Integer
ShowEvent
- playerNum: Integer
- starter: String
- hand: String
DealEvent
- playerNum: Integer
- dealtCards: String
ScorePlayEvent
- playerNum: Integer
- playerScore: Integer
- scoredPoints: Integer
DiscardEvent
- playerNum: Integer
- disCards: String
ShowScoreEvent
- playerNum: Integer
- playerScore: Integer
- scoredPoints: Integer
- play: String
StarterEvent
- starter: String
EventFactory
+ factory: EventFactory
+ getInstance(): EventFactory
+ createSeedEvent(seed: int): Event
+ createPlayerEvent(type: String, id: int): Event
+ createDealEvent(playerNum: int, hand: String): Event
+ createDiscardEvent(playerNum: int, hand: String): Event
+ createStarterEvent(starter: String): Event
+ createPlayEvent(playerNum: int, totalScore: int, cardPlayed: String): Event
+ createShowEvent(playerNum: int, starterCard: String, hand: String): Event
+ createScorePlay(playerNum: int, playerScore: int, scoredPoints: int, play: String): Event
+ createScoreShow(playerNum: int, playerScore: int, scoredPoints: int, play: String, hand: String): Event
Use
ScoreCalculator
+ scoreStarter(starter: Hand): int
+ scorePlay(segment: Hand, cardAdded: Card): int
+ scoreShow(hand: Hand, starter: Hand): int
+ scoreLastCard(): int
tracks score with
TotalPlayScoreCalculator
+ score(segment: Hand, cardAdded: Card): int
IPlayCalculator
+ score(segment: Hand, cardAdded: Card): int
PlaySumCalculator
+ score(segment: Hand, cardAdded: Card): int
PlayPairCalculator
+ score(segment: Hand, cardAdded: Card): int
PlayRunCalculator
+ score(segment: Hand, cardAdded: Card): int
PlayJackStarterCalculator
+ score(segment: Hand, cardAdded: Card, playerNum: int, curScore: int): int
PlayGoCalculator
+ score(segment: Hand, cardAdded: Card, playerNum: int, curScore: int): int
IShowCalculator
+ score(hand: Hand, starter: Hand): int
TotalShowScoreCalculator
+ score(hand: Hand, starter: Hand): int
ShowJackSuitCalculator
+ score(hand: Hand, starter: Hand): int
ShowFlushCalculator
+ score(hand: Hand, starter: Hand): int
ShowPairCalculator
+ score(hand: Hand, starter: Hand): int
ShowRunCalculator
+ score(hand: Hand, starter: Hand): int
ShowSumCalculator
+ score(hand: Hand, starter: Hand): int

*Figure 1. Full UML design class diagram of the Cribbage Trainer program. The added classes are shown with black borders and the original classes are shown with grey borders.*

## <u>Score Calculation</u>

### Problem Overview

The logic behind calculating scores for Cribbage depends on many game play factors; however, its input is always of the same type - a Hand object. Thus, the calculator faces the challenge of taking this input and deducting the correct point value it earns, which is subject to a wide array of rules, each earning a particular point value.

**Solution**
Score calculation is implemented following three design patterns: *Strategy*, *Composite*, and *Facade*.

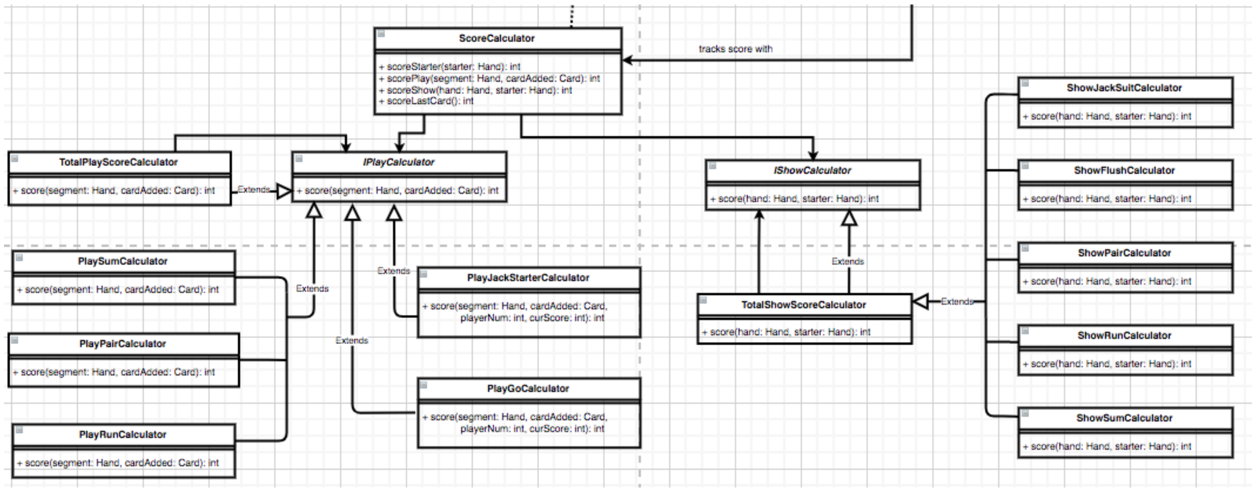

ScoreCalculator
+ scoreStarter(starter: Hand): int
+ scorePlay(segment: Hand, cardAdded: Card): int
+ scoreShow(hand: Hand, starter: Hand): int
+ scoreLastCard(): int
tracks score with
TotalPlayScoreCalculator
+ score(segment: Hand, cardAdded: Card): int
Extends
IPlayCalculator
+ score(segment: Hand, cardAdded: Card): int
PlaySumCalculator
+ score(segment: Hand, cardAdded: Card): int
PlayPairCalculator
+ score(segment: Hand, cardAdded: Card): int
PlayRunCalculator
+ score(segment: Hand, cardAdded: Card): int
Extends
Extends
Extends
PlayJackStarterCalculator
+ score(segment: Hand, cardAdded: Card, playerNum: int, curScore: int): int
PlayGoCalculator
+ score(segment: Hand, cardAdded: Card, playerNum: int, curScore: int): int
IShowCalculator
+ score(hand: Hand, starter: Hand): int
Extends
TotalShowScoreCalculator
+ score(hand: Hand, starter: Hand): int
Extends
ShowJackSuitCalculator
+ score(hand: Hand, starter: Hand): int
ShowFlushCalculator
+ score(hand: Hand, starter: Hand): int
ShowPairCalculator
+ score(hand: Hand, starter: Hand): int
ShowRunCalculator
+ score(hand: Hand, starter: Hand): int
ShowSumCalculator
+ score(hand: Hand, starter: Hand): int

Figure 2. ScoreCalculator class and its subsystem.

Calculating scoring during the play uses the *strategy pattern*. Several classes inherit from the IPlayCalculator abstract class. IPlayCalculator provides the common interface of the score method, which is implemented differently in each child class. For example, PlaySumCalculator only returns scores earned when the cards in the segment sum to 15 or 31, and PlayPairCalculator only returns scores earned when a card's rank matches the rank of another card.

In getting the total score for a play, the *composite pattern* is used. The TotalPlayScoreCalculator class is composed of several scoring strategies and implements the same interface as the atomic scoring strategies, which allows them to be treated identically. Similarly, scoring during the show uses the composite pattern through the IShowCalculator interface.

The ScoreCalculator class acts as a *facade*. It hides all of the complicated implementation details for the composite strategies behind an interface of just four functions: scoreShow, scorePlay, scoreStarter, and scoreGo. This pattern is used to add a level of indirection between the scoring functionality and the rest of the Cribbage program, which does not need to access the information related to score calculation. However, there is one exception - events related to scoring are created and logged inside of each scoring strategy class. This is so that the event logger has access to the score type (e.g. run4, pair2, fifteen, etc). Without this exception, the scoreType would have to be passed back to Cribbage, which would create the event. As is, Cribbage is a bloated class and creating the event here would violate the GRASP principle of *high cohesion*.

**Future Maintenance and Modification**
The strategy pattern allows scoring rules to be pluggable. If the NERD team later wants to add a new scoring rule, such as allow players to earn points for playing a card with the same suit as the previous card, they could achieve this easily by creating a new class (e.g. PlaySuitCalculator) that implements the common interface (i.e. is a child class of IPlayCalculator).

The composite approach also helps scoring rules to be pluggable. Any combination of scoring strategies can be easily combined to fit the whims of the NERD team - they just need to add another class.

The facade also provides *protected variations.* Any changes to the scoring rules has no impact on the Cribbage class - it can continue to call methods in the ScoreCalculator's interface without requiring knowledge of those changes.

## **Event Log**

**Problem Overview**
The Logger class logs events that occur within the Cribbage game to the cribbage.log file. A new log file is created at the start of a game and events are then appended to the file as they occur during game play.

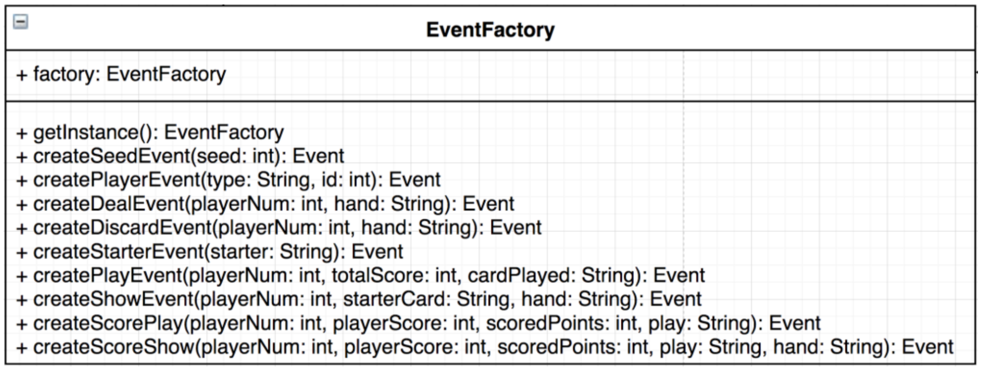
Each type of event is defined as its own concrete class deriving from an abstract Event class. Event classes each have a string attribute of their event type and an overridden toString method, which returns the log output that is respective to that class. Concrete event classes then define their own distinct attributes, such as the hand of cards dealt in a deal or the point value for a score.

Two considerations were necessary for designing the event logging functionality:

1) The logger must log events that have different display requirements to the same file.

2) The events can have different types of triggers, including game play events, such as deals and discards, and score events, such as reaching a segment value of 15.

**Solution**
The first consideration is handled by adding a level of indirection between the Cribbage game and Logger classes. Following the Gang of Four *factory pattern* and GRASP *pure fabrication* principle, the complex creation logic behind generating events is handled through the EventFactory class. EventFactory has specialised methods for the creation of each type of game event.



```
EventFactory
+ factory: EventFactory
+ getInstance(): EventFactory
+ createSeedEvent(seed: int): Event
+ createPlayerEvent(type: String, id: int): Event
+ createDealEvent(playerNum: int, hand: String): Event
+ createDiscardEvent(playerNum: int, hand: String): Event
+ createStarterEvent(starter: String): Event
+ createPlayEvent(playerNum: int, totalScore: int, cardPlayed: String): Event
+ createShowEvent(playerNum: int, starterCard: String, hand: String): Event
+ createScorePlay(playerNum: int, playerScore: int, scoredPoints: int, play: String): Event
+ createScoreShow(playerNum: int, playerScore: int, scoredPoints: int, play: String, hand: String): Event
```

Figure 3. EventFactory class attribute and methods.

As shown in Figure 3, the tailored factory methods handle creation of their eponymous event objects, which each have attributes that vary in number and type. Following the GRASP principle of *low coupling*, Event objects are added to the log during construction, which removes the need for a relationship between Logger and Cribbage.

Furthermore, EventFactory allows for *high cohesion* in Logger and concrete Event classes, as it deals entirely with the logic standardising them into objects that derive from the same abstract class. As a result, the logEvent method in Logger simply takes an Event object and appends the returned value from its toString method to the log.

The second consideration is handled by defining EventFactory as a Singleton class with global visibility, which is accessible through its getInstance method. This is necessary to create events which are triggered during score calculation, such as placing a card which results in a total segment value of 15 or 31.

## Future Maintenance and Modification

The EventFactory class allows the NERD team flexibility in adding new game rules in the future. New scoring event types (in addition to scoring during the Play and the Show) can be added by simply creating a new concrete event class deriving the Event class and creating a corresponding factory method for it. As logging is invoked in Event object constructors, a new event object will handle that responsibility for itself without the need to edit any other classes. New scoring strategies do not require creation of a corresponding event class in order to log them. As long as they inherit from the IPlayCalculator or IShowCalculator classes and call either the createScorePlayEvent or createScoreShowEvent methods (respectively) of those interfaces, they can be logged easily.

## Alternate Implementation Strategy

The Gang of Four *Observer* pattern was considered, as it is commonly used in practice for other programs that log events. In theory, the logger would be observing changes to the game's state published by the Cribbage class. We determined that this pattern would add unnecessary complexity to the program, as events are only logged to one location. While the observer pattern would allow the NERD team to add new log destinations in the future, this would be highly unlikely given the purpose of the program. Therefore, this was not included in the logging strategy for the purpose of reducing complexity.