Project 2 Report

Ehsan Soltani Abhari - 1003877
Hoang Dang - 1080344
(2 person team)
Workshop 16 Team 06

The Scoring System:

The scoring system was created based upon the Composite Strategy pattern combination. Scoring in the game of Cribbage is done based on a set of rules. Each rule assesses a particular state of the game in some way, and associates with it a numeric value - the score. We designed our program to work with different variations of rules, as well as different rules all together. All of these implementations were contained inside the *Score* package. (Refer to Diagram 2 below)

Scorer - A Strategy pattern was used to implement the rules. The *Scorer* class was created to allow inheritance from all subsequent scoring strategies, providing a layer of polymorphism. The *Scorer* class dictates that all scoring strategies in the game must have an *evaluate()* method that evaluates a hand, and returns an integer score, representing the value of that hand. With this approach, new rules can be added or removed, and existing ones modified, without affecting the rest of the program, as any caller to the class will simply use the generic *evaluate()* method. To build on this, a further inheritance structure was also implemented, where *Scorers* can inherit from each other and reuse functionalities. An example of this extra inheritance is our *RunScorerPlay* (scoring runs during Play) extending upon *RunScorer* (scoring all runs in a hand) and using *RunScorer*'s ability to generate run-combinations to its advantage.

ScorerFactory - To provide package-private access to the concrete *Scorer* strategies, a Singleton Factory pattern was implemented. The *ScorerFactory* creates and holds instances of concrete *Scorers*, meaning the same instances will be returned every time a get method is called. This abstraction allows uniform access to concrete *Scorers* and prevents any accidental construction of concrete *Scorers* elsewhere, while increasing cohesion of the system by delegating all concrete *Scorer* creation to a single class, and also saving resources by limiting arbitrary creation and destruction of concrete *Scorers.*

ScorerComposite - The Composite pattern was incorporated with the Strategy pattern to further strengthen the dynamic scoring ability of the scoring system. Composites of rules were created for different scoring combinations, mainly, for scoring the Play and Show stages of the game. This replaces the alternative of calling upon a list of *Scorers* in different parts of the program which would have introduced coupling between the scoring system and the main system. Instead, the *ScorerComposite* class extends upon the *Scorer*, thus having the same external-facing functionality as any other *Scorer*. Internally, however, each composite evaluates a hand by calling on a list of *Scorers* and using their combined logic to get a hand value, resulting in the abstraction of this logic away from the main system, and increasing cohesion in both the scoring system and the main system.

ScorerCompositeFactory - The Singleton Factory pattern was used again to provide external access to both concrete and composite *Scorers*. This class is the only outward facing interface that allows external access to the *Score* package, it is Singleton to ensure consistent global behaviour. In order to get access to a *Scorer* object, a caller needs to use the *getScorerComposite()* method from this class. The method returns only the correct *Scorer* object required depending on the current game stage (play/score/etc..) as defined in *Cribbage*, meaning the main program no longer needs to know what kind of *Scorer* it needs to create for each stage of the game. By expert, this class knows which *Scorer* combination is suitable for each game stage, thus it was also assigned the responsibility of the creator to generate different composite *Scorer* objects.

The *Scorer* object returned by *ScorerCompositeFactory*, by polymorphism, can either be a concrete or composite object. Any external caller will get access to the same *Scorer* object, given the game stage does not change. Thus, restricting external access to the *Score* package through this class helped us achieve protected variation in our system, as internal changes to the scoring system will not affect external components. The design also resulted in very low coupling with other parts of the main system, as internal components are completely non-reliant on external components, except the game stage.

Is it worth noting that while we could have generated all concrete *Scorers* from *ScorerCompositeFactory*, we abstracted this logic into *ScorerFactory* to make an internal distinction between the creation of concrete and composite *Scorers*, thus reducing responsibilities and increasing cohesion for *ScorerCompositeFactory*.

Properties file - The existing properties file was altered to allow for more parameters to be taken into the program, specifically useful are the different scores used for different scoring rules. While we initially declared these scores as constants in each concrete *Scorer* class, we later decided that these parameters should be defined in the properties file and taken in at the start of the program, to cater for variations in rules. This way, the system contains uniform information throughout, and fewer constants have to be defined in the program. Any variations in existing rules can be easily changed in the properties file.

Alternatives for Scoring system - In initial discussions, we considered using other designs for our Scoring system. One such design is a Facade-based rule engine, where a complicated sub-system of rules can operate away from the main system, however, its various functionalities are accessible through the Facade. We picked the Composite Strategy pattern over this because the Facade pattern is more useful for subsystems with varying functionality, while our scoring system only has one uniform goal - to score a hand of cards. The other alternative we considered is the Decorator pattern, where multiple rules can be attached to a base rule to create different combinations. We picked the Composite Strategy pattern over this because there was no base functionality defined for the scoring system, each combination works based on a different aspect of the game, thus making it impossible to have a base rule to build upon.

The Logging System:

In designing the logging system, we realised that we needed something to monitor the state of the game, and update the log whenever specific events occur. This requirement fits in perfectly with the idea of publisher-subscriber behind the Observer pattern, as the main game can notify its observers as soon as an event of interest occurs. Interestingly, since the logging system had variations in its functionality depending on the state of the game, we decided it would also be appropriate to incorporate the Composite Strategy pattern into this subsystem. To increase the cohesion of the *Cribbage* class, and reduce coupling between the logging system and the main system, we abstracted all logging functionality into the *Log* package. (Refer to Diagram 3 below)

Logger - we created the abstract *Logger* class, which contained the *update()* method that all children *Loggers* have to implement. As the logging system behaves differently based on the current stage of the game, different logging strategies were implemented in the form of different concrete *Logger* classes to avoid having one bloated class that handles all logging logic; this resulted in a convenient layer of polymorphism between *Loggers*. These concrete *Loggers* overwrite the *update()* method of the base *Logger* class to define its own logging behaviour, making the logging system cohesive, and future changes to logging strategies easy to implement. An example of a concrete *Logger* is the *ShowLogger*, which writes to the log file only if called upon while the game is in the Show stage. With this approach, *Cribbage* can simply notify all of its *Loggers* whenever something of interest happens, without having to provide any information directly in the publishing call. This increases the cohesion of *Cribbage*, and minimises the coupling between *Cribbage* and the logging system.

LogManager - To further reduce responsibilities for the *Cribbage* class, we decided to create a *LogManager*, which would contain all the relevant observer *Loggers* for *Cribbage.* The *LogManager* inherits from *Logger*, and is a composite containing multiple *Loggers*. Whenever *LogManager* is notified of an event, it passes this information on to all of its members, who will perform logging if it's the correct game stage for them. New *Loggers* can be added, and existing ones modified, inside *LogManager*, without affecting how the system works externally due to polymorphism. This adheres to the principle of protected variation, as changes within the logging system would not affect how the *LogManager* operates externally. The design resulted in a highly cohesive logging system that is minimally coupled with *Cribbage*, it provides an indirection between *Cribbage* and the logging system through the *LogManager*, meaning only the *LogManager* needed to observe *Cribbage*.

GameInformation - A big challenge for the logging system was the fact that different types of logging required various information that was spread throughout the main program. We decided against the idea of having our *Logger* take in a bunch of parameters via its *update()* method, as that would create difficulties in having inheritance between *Logger* classes, while also forcing *Cribbage* to pass the correct information to its *Loggers*, increasing coupling between the two. We came up with *GameInformation*, an inner class that contains the source of truth for various pieces of information which often change throughout the game. This inner class holds variables that were previously defined arbitrarily throughout the program, resulting in a small refactorisation of the provided *Cribbage* class. With this source of information, *Loggers* can directly access whichever information is relevant to its functionality upon the publishing call, thus reducing the coupling between *Cribbage* and the logging system.

ScorerCache - Another challenge we faced when designing the Observer pattern for *Loggers* was the fact that some of the information we required were not contained in *Cribbage* at all. Specifically, since we delegated all scoring functionality to the *Score* package, *Cribbage* does not hold any information about how the game was scored. As *Cribbage* was the subject to observe, we decided against also observing *Scorers*, as this would couple the various systems together. The alternative that we went with was based on Pure Fabrication: to create a caching system for the *Scorers*, thus came along the *ScorerCache* class. The *ScorerCache* contains an integer representing a score, a string representing the scoring rule applied, and a combination of cards that resulted in the score. The base *Scorer* class was edited to contain a list of *ScorerCache* objects. This means that after an *evaluate()* call, a *Scorer* object can have 0 to many *ScorerCaches* stored, depending on how many scoring rules were awarded to the hand given. Simply put, the *Scorer* object now holds the most recent history of scoring events.

This added design was extremely elegant as *Loggers* do not need to be passed any information from *Cribbage* regarding scoring, but will go straight to the scoring system to ask for this information. The logging system writes logs based on the current game stage; conveniently, the *ScorerCompositeFactory* also provides the same *Scorer* object instance globally based on the game stage; this means that *Loggers* will receive the same *Scorer* object instance that was used to score the current game stage. With *Scorers* being available from the globally accessible Singleton Factory, *Loggers* can easily get their cached scoring information, and then combine these with the information they need from the *GameInformation* class of *Cribbage* to output the correct log message. We were conscious of the fact that this information exchange between the Scoring and Logging systems could result in coupling between the two, and tailored our design to minimize this. A *Scorer* simply keeps track of scores it calculated, and makes this information available, it does not know what class is accessing the information and for what purpose. A *Logger* simply uses the information a *Scorer* makes available to log something of note, it does not know how this calculation came about, and does not interact with the inner-workings of Scorer. Thus we believe this design allows the necessary exchange of information whilst keeping the coupling between 3 complicated systems to an absolute minimum.

Alternatives for Logging system - We considered alternative designs for the logging system, one of which was the implementation of completely different observers, who would take in different inputs and operate completely different to one another. This may be applicable to the case of observing *Scorers* separately from observing *Cribbage*. Such a design can make use of the Facade pattern, abstracting different functionalities into a subsystem and operating them through the Facade object by indirection. Another alternative would be to use the Adapter pattern to adapt the different behaviours of different observer behaviours into a uniform API that can be used globally. However, we determined that both of these designs would require the main system to keep track of the subsystems' behaviours, or make more active calls to arbitrary methods, or pass parameters into the subsystem methods, thus introducing more coupling and reducing cohesion for *Cribbage*. We ultimately found that the Observer pattern in collaboration with the Strategy Composite pattern allows for the most optimal design.

Summary:

We tried to keep changes to the main program to a minimum. As mentioned previously, the biggest change was creating the *GameInformation* class to keep track of various information throughout the game. This allows external systems, which require this information, to access it by interacting with the *GameInformation* class, keeping coupling between our systems and the main program to a minimum.

To calculate the score for a particular hand, the *Cribbage* class simply calls the *getScorerComposite()* method of the *ScorerCompositeFactory*, and then calls *evaluate()* from the *Scorer* object returned. *Cribbage* does not need to worry about which *Scorer* is needed for a particular scenario; through the *GameInformation* class, information about the current phase of the game is automatically obtained by the *ScorerCompositeFactory*, and the correct *Scorer* is provided.

To log a particular event, all *Cribbage* has to do is to call the *update()* method of its *LogManager*, which in turn will call the *update()* method of all its *Loggers*. *Cribbage* does not need to worry about calling the right *Loggers* or giving the necessary information. *Loggers* use the *GameInformation* class and the *Scorer* caching system to automatically obtain all information they require.

Diagram 1

Overview of Systems:

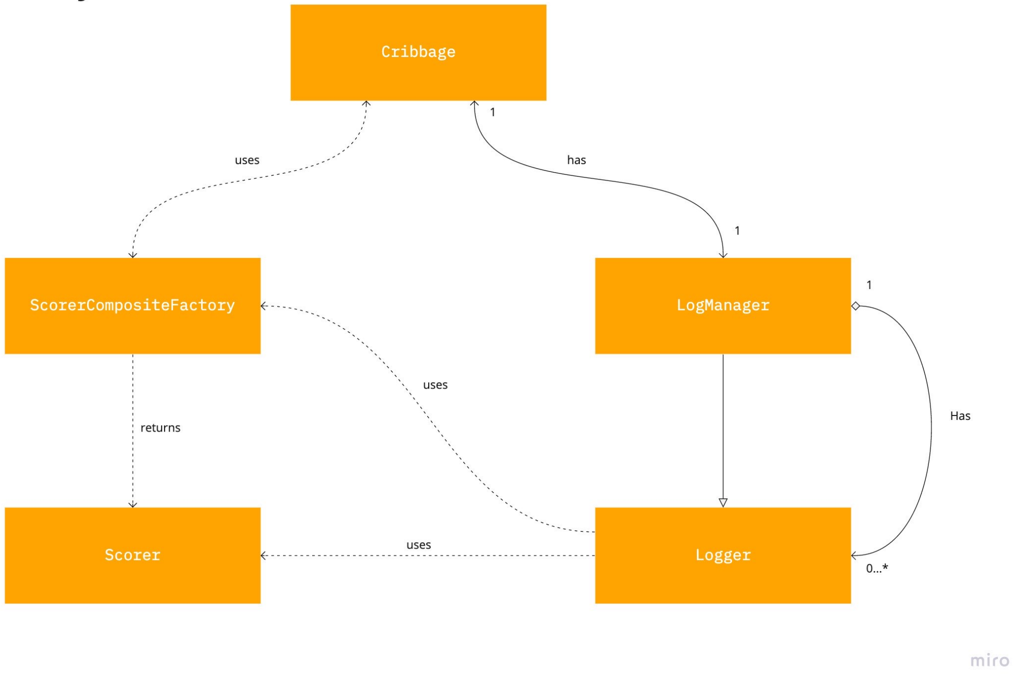


Diagram 2

Scoring System:

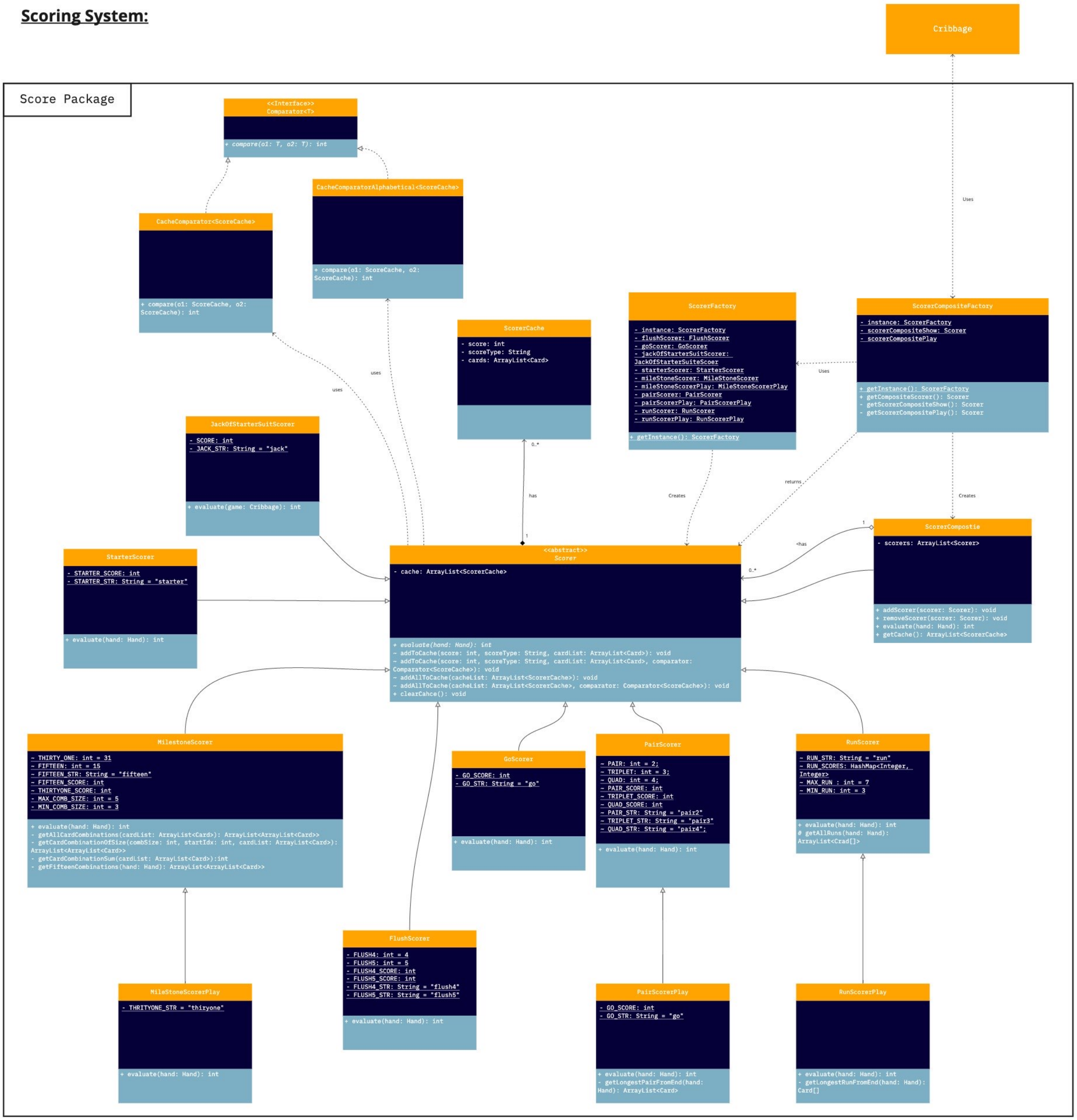
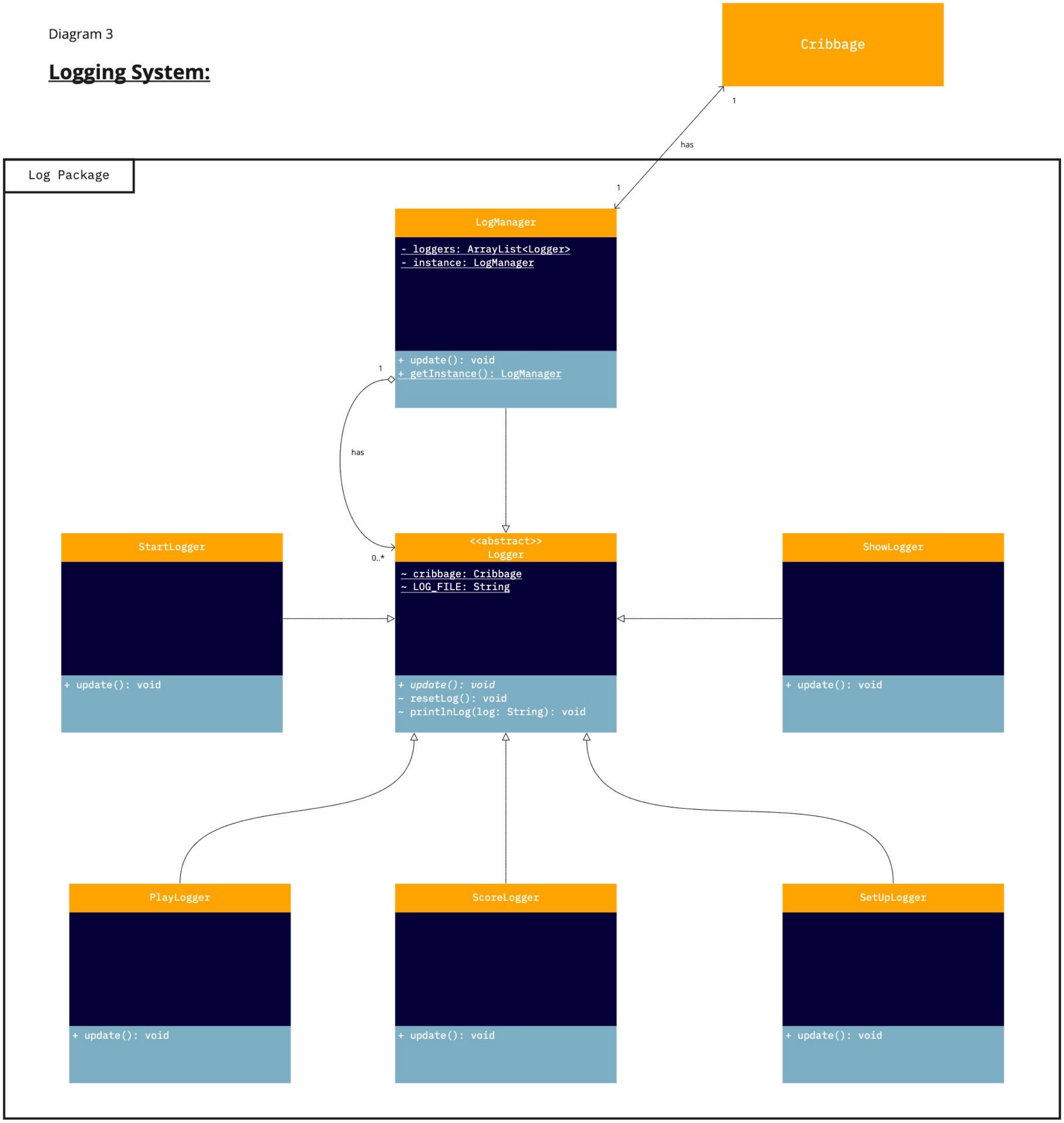

Cribbage
Uses
Score Package
<<Interface>> Comparator<T>
+ compare(o1: T, o2: T): int
CacheComparatorAlphabetical<ScoreCache>
+ compare(o1: ScoreCache, o2: ScoreCache): int
CacheComparator<ScoreCache>
+ compare(o1: ScoreCache, o2: ScoreCache): int
uses
uses
ScorerCache
- score: int
- scoreType: String
- cards: ArrayList<Card>
0..*
has
1
ScorerFactory
- instance: ScorerFactory
- flushScorer: FlushScorer
- goScorer: GoScorer
- jackOfStarterSuitScorer: JackOfStarterSuiteScoer
- starterScorer: StarterScorer
- mileStoneScorer: MileStoneScorer
- mileStoneScorerPlay: MileStoneScorerPlay
- pairScorer: PairScorer
- pairScorerPlay: PairScorerPlay
- runScorer: RunScorer
- runScorerPlay: RunScorerPlay
+ getInstance(): ScorerFactory
Uses
ScorerCompositeFactory
- instance: ScorerFactory
- scorerCompositeShow: Scorer
- scorerCompositePlay
+ getInstance(): ScorerFactory
+ getCompositeScorer(): Scorer
- getScorerCompositeShow(): Scorer
- getScorerCompositePlay(): Scorer
Creates
returns
Creates
ScorerCompostie
- scorers: ArrayList<Scorer>
+ addScorer(scorer: Scorer): void
+ removeScorer(scorer: Scorer): void
+ evaluate(hand: Hand): int
+ getCache(): ArrayList<ScorerCache>
1
<has
0..*
JackOfStarterSuitScorer
- SCORE: int
- JACK_STR: String = "jack"
+ evaluate(game: Cribbage): int
StarterScorer
- STARTER_SCORE: int
- STARTER_STR: String = "starter"
+ evaluate(hand: Hand): int
<<abstract>> Scorer
- cache: ArrayList<ScorerCache>
+ evaluate(hand: Hand): int
~ addToCache(score: int, scoreType: String, cardList: ArrayList<Card>): void
~ addToCache(score: int, scoreType: String, cardList: ArrayList<Card>, comparator: Comparator<ScoreCache>): void
~ addAllToCache(cacheList: ArrayList<ScorerCache>): void
~ addAllToCache(cacheList: ArrayList<ScorerCache>, comparator: Comparator<ScoreCache>): void
+ clearCahce(): void
MilestoneScorer
~ THIRTY_ONE: int = 31
~ FIFTEEN: int = 15
~ FIFTEEN_STR: String = "fifteen"
~ FIFTEEN_SCORE: int
~ THIRTYONE_SCORE: int
- MAX_COMB_SIZE: int = 5
- MIN_COMB_SIZE: int = 3
+ evaluate(hand: Hand): int
- getAllCardCombinations(cardList: ArrayList<Card>): ArrayList<ArrayList<Card>>
- getCardCombinationOfSize(combSize: int, startIdx: int, cardList: ArrayList<Card>): ArrayList<ArrayList<Card>>
- getCardCombinationSum(cardList: ArrayList<Card>):int
- getFifteenCombinations(hand: Hand): ArrayList<ArrayList<Card>>
GoScorer
- GO_SCORE: int
- GO_STR: String = "go"
+ evaluate(hand: Hand): int
PairScorer
~ PAIR: int = 2;
~ TRIPLET: int = 3;
~ QUAD: int = 4;
~ PAIR_SCORE: int
~ TRIPLET_SCORE: int
~ QUAD_SCORE: int
~ PAIR_STR: String = "pair2"
~ TRIPLET_STR: String = "pair3"
~ QUAD_STR: String = "pair4";
+ evaluate(hand: Hand): int
RunScorer
~ RUN_STR: String = "run"
~ RUN_SCORES: HashMap<Integer, Integer>
~ MAX_RUN : int = 7
~ MIN_RUN: int = 3
+ evaluate(hand: Hand): int
# getAllRuns(hand: Hand): ArrayList<Crad[]>
FlushScorer
- FLUSH4: int = 4
- FLUSH5: int = 5
- FLUSH4_SCORE: int
- FLUSH5_SCORE: int
- FLUSH4_STR: String = "flush4"
- FLUSH5_STR: String = "flush5"
+ evaluate(hand: Hand): int
MileStoneScorerPlay
- THRITYONE_STR = "thiryone"
+ evaluate(hand: Hand): int
PairScorerPlay
- GO_SCORE: int
- GO_STR: String = "go"
+ evaluate(hand: Hand): int
- getLongestPairFromEnd(hand: Hand): ArrayList<Card>
RunScorerPlay
+ evaluate(hand: Hand): int
- getLongestRunFromEnd(hand: Hand): Card[]

Diagram 3

Logging System:



miro