Daniel Digby Fiona Zhu Jasper Robbins

SWEN30006 – Project 2 Report

Our implementation for the Scoring and Logging of Cribbage revolved largely around creating a scoring subsystem with a Facade Controller, using Composite Strategy for the scoring rules, and Observer pattern for the logging. We represented the information about a game event in a GameEvent class or a subclass of GameEvent, which would be passed back from the scoring subsystem to Cribbage. Within the Cribbage class, an EventManager observes the state of Cribbage. When a rule is validated and returns an event Cribbage notifies the EventManager which will subsequently notify its registered Listeners, of which LoggingListener will receive the events, and print the toString of the event. This allows complete separation of the scoring and logging systems, with Protected Variations for the scoring system and High Cohesion within the logging system.

Scoring

ScoreFacade

For scoring, we decided to create a stable interface around the scoring subsystem using a Facade, ScoreFacade. Cribbage interacts with the scoring subsystem through this facade, which provides Protected Variations for the scoring system by introducing a layer of Indirection, in addition to maintaining Low Coupling between Cribbage and the scoring subsystem.

Rule Configuration & Implementation

The rule configurations were implemented in a way that allowed future extension of rules, as well as easy configuration of the rules applied and the respective points values for each. The configuration of the ruleset is dictated in the “rulesConfig.txt” file, which is then parsed by the logic in the Ruleset class. If a rule configuration is not specified, a set of default rules and point values will be applied. This parsing is done by an instance of Ruleset. This instance is held within ScoreFacade, as ScoreFacade should be the Expert for the rule configuration for the game.

The ruleset instantiated in ScoreFacade can then be used when instantiating the applicable rules for a ruleset for a scoring state (starter, play, show). For storing individual rules, we opted for the Strategy pattern as this seemed to be the clear option, as we had multiple algorithms with varied logic that were still related in their input and outputs. Furthermore, when we considered that we needed to store rules together for the starter, play and show states, it was clear that Strategy could be combined with the Composite pattern to achieve a clean implementation of scoring.

With the Composite Strategy pattern, each rule is represented by a strategy object, which can all be accessed through IRuleStategy. Within this structure, the Composite pattern allows aggregation of multiple applicable strategies into one composite strategy that can be treated the same as a single strategy.

With this implementation, CompositeRuleStrategy only needs to iterate through the Ruleset to add the applicable rules and their value points for the EventType and regional configuration. Thus, CompositeRuleStrategy never has to know about the state of the game or configurations, maintaining Low Coupling and High Cohesion. The creation of rules within the Composite Strategy implementation was done using Pure Fabrication with a Factory, RuleStrategyFactory, to maintain the High Cohesion of ScoreFacade by abstracting creation logic away. We decided to make RuleStrategyFactory not a singleton because it has a one to one relationship with the ScoringFacade and does not need to be accessed anywhere else.

Representing Game & Score Events
We used GameEvent as the superclass to represent the information for an event or action within a game. Game actions include deal, discard, play, starter (start) and show, and the events representing these are StatusEvent and ScoreEvent, which inherit from a base GameEvent. By packaging our loggable events into classes, we can easily pass them around and perform operations on them throughout the code. Furthermore, the polymorphism supplied by this allows us to treat all loggable activity the same, upcasting ScoreEvent and PlayEvent to GameEvent where necessary. The toString() method provided in the GameEvent and subclasses also allows High Cohesion of the classes to represent themselves for logging later on, rather than the logging system needing to parse the events individually to retrieve the information.

When a rule is validated against an action and returns a positive score, the rule strategy will create and return an event representing the score and its details. This method is inherited from the interface IRuleStrategy, and hence it can be called the same way for all strategies. Event objects are instantiated from a Singleton EventFactory. Singleton was used because the factory needs to be accessed from a large range of classes, and Factory was chosen as it abstracts the creation logic of these more complex objects from the strategy classes, maintaining High Cohesion. Although this increases the representational gap, Pure Fabrication in this case is worth it as game events as it allows code reuse between different classes.

We used the Composite pattern again here for ScoreEvent, so that score events created by the strategies can be packaged into one single CompositeScoreEvent. This pattern is necessary as when one action is played multiple rules may be triggered. Hence by packaging these into one object, no modification or extra logic is necessary to handle varied amounts of score events throughout the system. Modification of the toString() and getScore() methods make this possible, and benefit in keeping High Cohesion in Cribbage and LoggingListener, by being able to call getScore() and toString() on a single object, instead of needing to loop through an arraylist.

Cribbage calls EventFactory to create GameEvents for any action such as deal, discard or play (PlayEvent upcasted). When the *score()* method in ScoreFacade is called, we pass in the *segment* object containing information about the current state of the game and action played. This is where the responsibility of cribbage ends in terms of creating events. The Cribbage class never needs to know about how Events are scored or created as the logic is abstracted away to the scoring system, coordinated by the ScoreFacade FACADE. The applicable composite rule class is then selected and applied to the action, without any excess logic needed other than identifying which group of rules to apply. This maintains High Cohesion in the ScoreFacade class, another benefit of the Composite Strategy pattern used.

Once a composite rule class is applied to the action, any score events will be passed back from a strategy through to Cribbage as a GameEvent, which allows them to be treated the same as the other GameEvents created by Cribbage. This choice was guided by High Cohesion for the Cribbage class. Having a standard type for all events also allows the logging system to deal with all loggable activity, whether it be a different type of event (Status or Score) or multiple events (CompositeScoreEvent) in a uniform manner.

Logging
For the logging portion of the implementation, we opted for the Observer pattern for multiple reasons. Firstly, the Observer pattern seemed appropriate, as it allows logging to occur dynamically as play is occurring, whenever an action happens, and this integrates well with the scoring system which does its work dynamically as well. Observer also allows High Cohesion within Cribbage to now add much logic, and Low Coupling between Cribbage and the classes that receive the information.

For our implementation, we used Indirection and Pure Fabrication to create an EventManager class to further abstract the logic of the Observer pattern from Cribbage. This achieves Low Coupling with the receiving objects of the information, and maintains High Cohesion of Cribbage by not needing to maintain a list of observers.

We also made the decision to couple EventManager to Cribbage rather than to the IRuleStrategy interface. This calls back to how our scoring system passes back GameEvents to Cribbage, because we strongly considered another option where IRuleStrategies would create GameEvents and then pass them directly to the EventManager, who was listening to either these objects, or EventFactory, instead of Cribbage. This was considered for High Cohesion of Cribbage, as well as the Low Coupling of not having to pass a GameEvent object back through from each strategy through ScoreFacade to Cribbage. However, the decision to instead listen to Cribbage was made for the purpose of Low Coupling between the Logging / Observer subsystem and the scorer subsystem. With our current implementation, the scoring and logging systems are completely independent of each other, and we deemed this Low Coupling and Protected Variations between the systems more important.

Our Observer Pattern for logging monitored the state of Cribbage in terms of GameEvents, the class created to represent an action in a game. This choice was made as this activity is the only part of the Cribbage that seems loggable. For our listeners, we implemented an interface for CribbageEventListener that our LoggingListener implements. We chose to add an interface here to account for potential future extensions of other listeners, such as if we were going to record analytics throughout the game, we could have a AnalyticsListener class.

For the implementation of our logging listener, this was made extremely simple by virtue of our effective design in the scoring system. As we have fully encapsulated all loggable activity within the GameEvent class, including the ability to store multiple ScoreEvent objects within one composite, we can parse all these objects in the same way. This allows our LoggingListener to simply take a GameEvent when one is generated, and print this to the logging file with the toString() method.

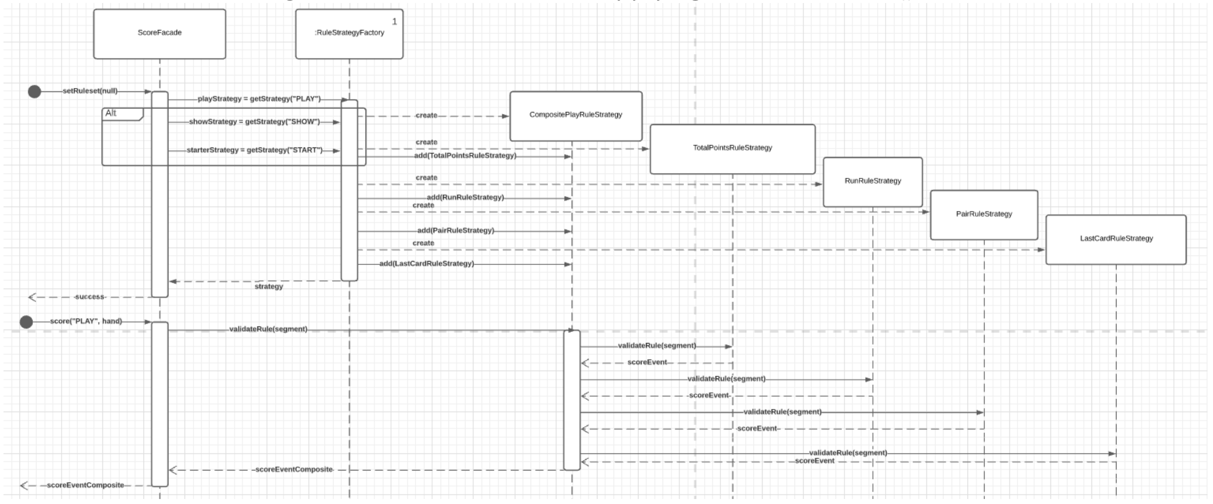
We made the decision to encapsulate the task of representing a GameEvent as a string within the class itself as stated prior, rather than include this logic in the LoggingListener class. This choice was made to keep High Cohesion and Low Coupling for both the LoggingListener and the GameEvent classes, however this came at a trade-off of being able to easily change the logging format.

There was however another implementation we considered, where GameEvent was a standalone class, and we utilised the Decorator pattern to decorate the log output of the event. In this approach, the EventFactory would instantiate GameEvent objects, and then decorate it with each relevant field as an additional layer to be unpacked for logging. This would allow the LoggingListener to parse the object and print it in the desired output without having a fixed toString() method be predetermined by the GameEvent. Although this design would make it easier to change the logging method, simply by configuring the Decorator settings to include or not include a layer to be logged, we decided against this. It appeared to be overdesign, as well as it abstracted the logging responsibility to the EventFactory class, rather than the individual objects themselves. We therefore kept the toString representative of the events to maintain High Cohesion also, and solved this problem by inheritance with the event classes.

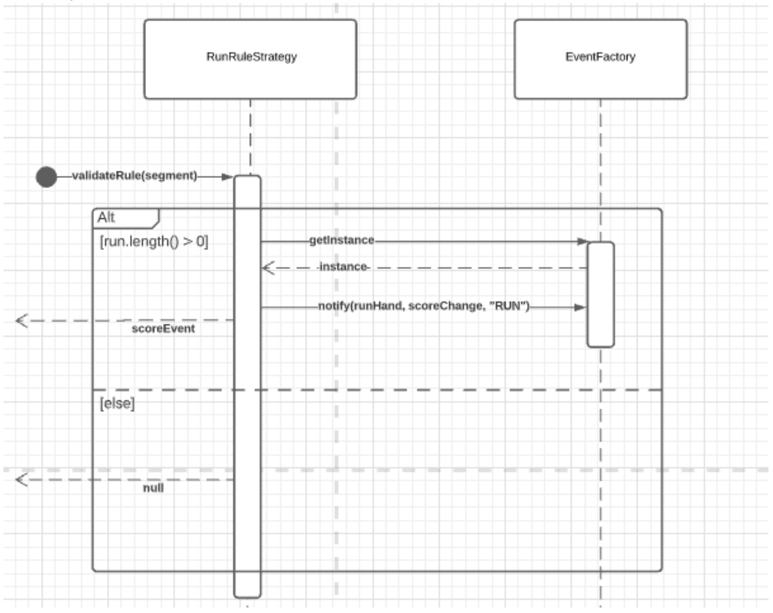
TESTING

We rigorously tested the system, unit testing all strategies, as well as integration testing the integration of the composite scoring system/factory with the scoring facade and ruleset. Testing both parsing in and creation of IRuleStrategies, as well as their use. Our unit tests are no longer functional as we had to make modifications to Cribbage in order to set up testing which have since been removed in our final product.

ScoreFacade - Initialising rulesets with setRuleset, applying rules with score()

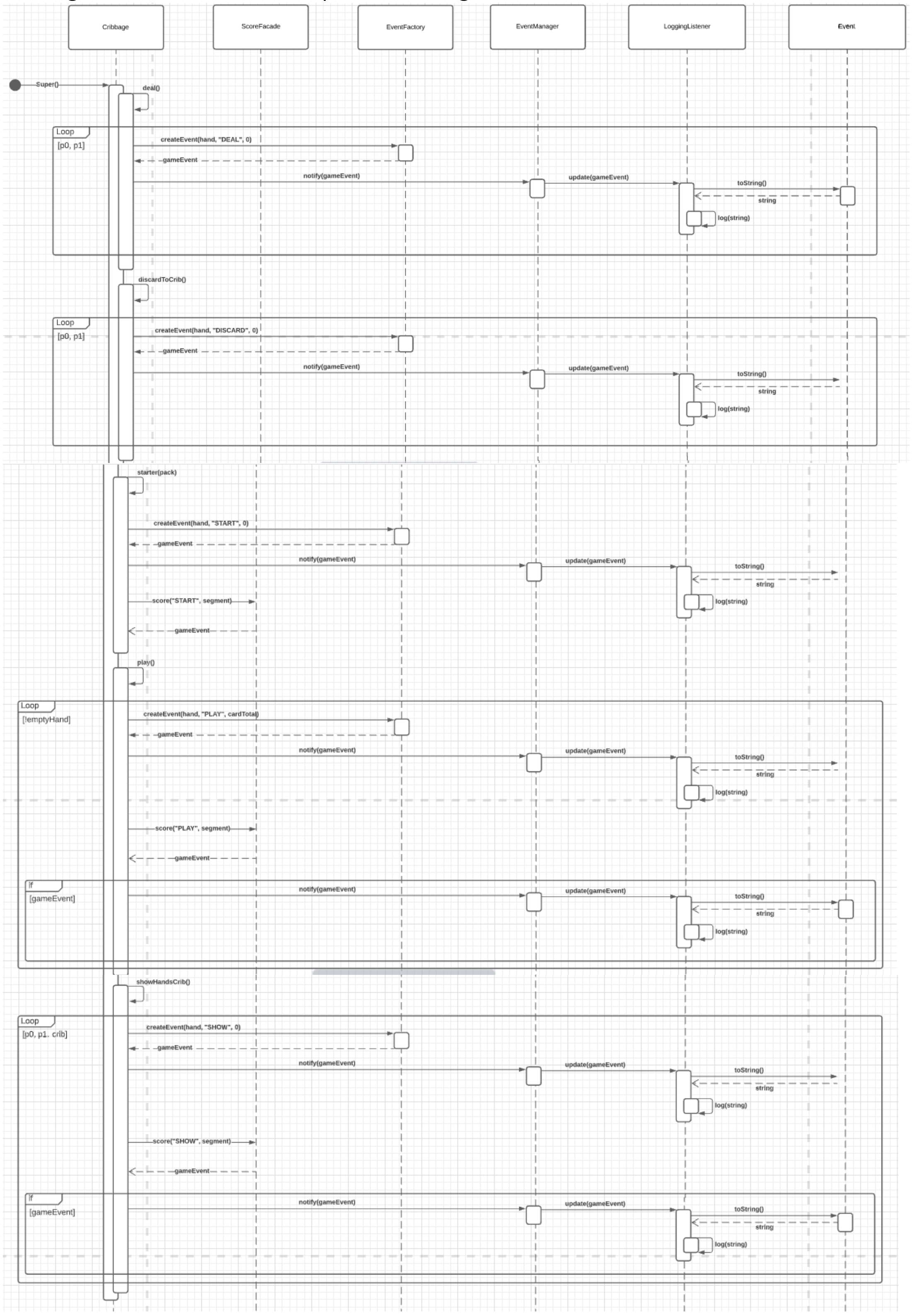


Example of validateRule() call

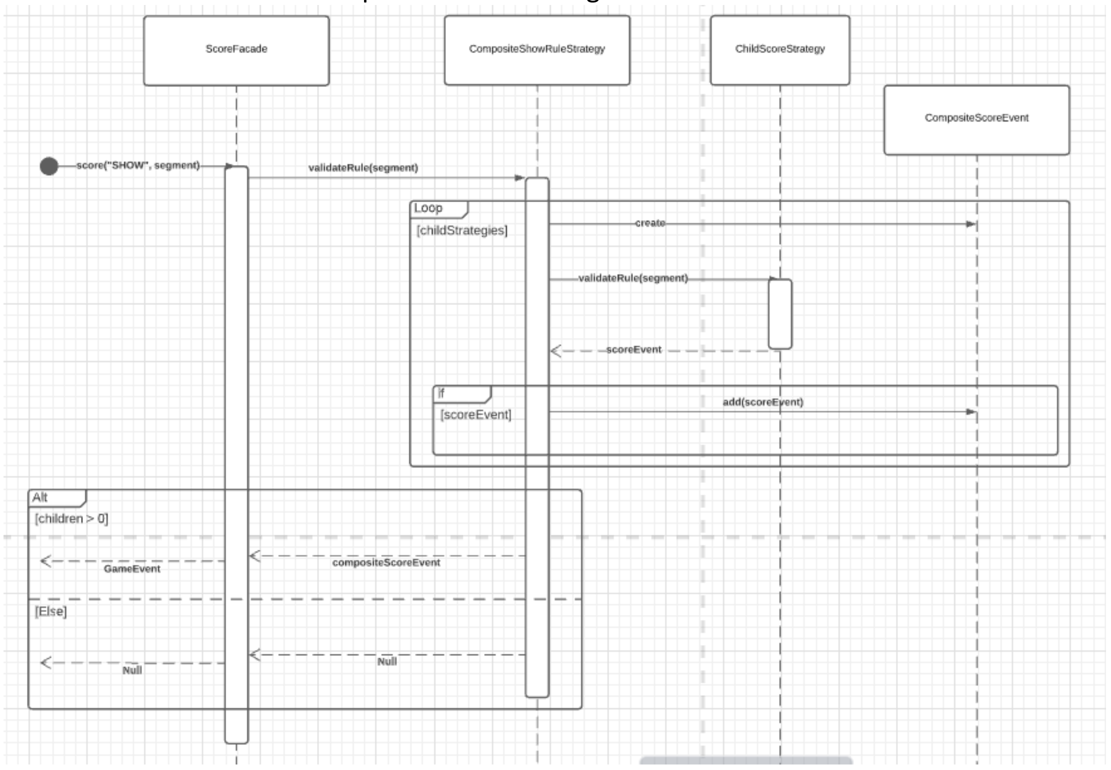


Daniel Digby Fiona Zhu Jasper Robbins

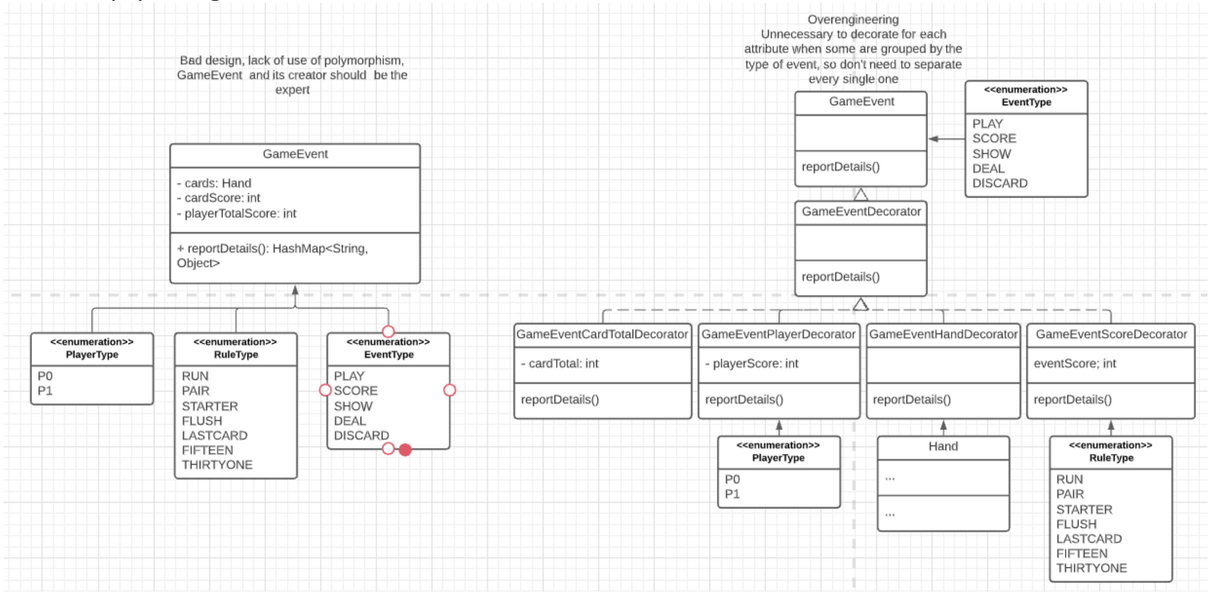
Cribbage: Calls to ScoreFacade upon different game events



ScoreFacade: Creation of Composite Score Strategies

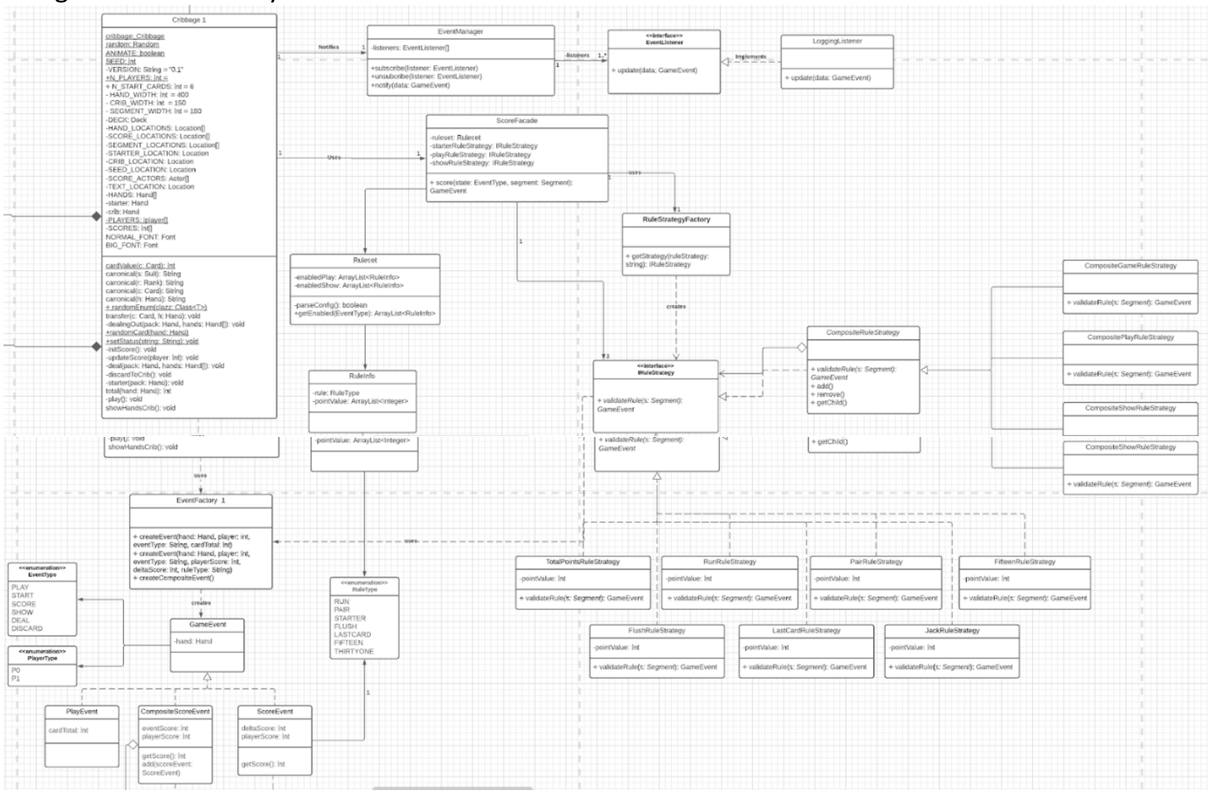


Alternatives considered for GameEvent - Decorator pattern for logging, single event with multiple null / empty string values



Daniel Digby Fiona Zhu Jasper Robbins

Design Model of Full System


Cribbage 1
cribbage: Cribbage
random: Random
ANIMATE: boolean
SEED: int
-VERSION: String = "0.1"
+N_PLAYERS: int =
+ N_START_CARDS: int = 6
- HAND_WIDTH: int = 400
- CRIB_WIDTH: int = 150
- SEGMENT_WIDTH: int = 180
-DECK: Deck
-HAND_LOCATIONS: Location[]
-SCORE_LOCATIONS: Location[]
-SEGMENT_LOCATIONS: Location[]
-STARTER_LOCATION: Location
-CRIB_LOCATION: Location
-SEED_LOCATION: Location
-SCORE_ACTORS: Actor[]
-TEXT_LOCATION: Location
-HANDS: Hand[]
-starter: Hand
-crib: Hand
-PLAYERS: Iplayer[]
-SCORES: int[]
NORMAL_FONT: Font
BIG_FONT: Font
cardValue(c: Card): int
canonical(s: Suit): String
canonical(r: Rank): String
canonical(c: Card): String
canonical(h: Hand): String
+ randomEnum(clazz: Class<T>)
transfer(c: Card, h: Hand): void
-dealingOut(pack: Hand, hands: Hand[]): void
+randomCard(hand: Hand)
+setStatus(string: String): void
-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
-play(): void
showHandsCrib(): void
Notifies
1
1
EventManager
-listeners: EventListener[]
+subscribe(listener: EventListener)
+unsubcribe(listener: EventListener)
+notify(data: GameEvent)
-listeners
1..*
1
<<interface>>
EventListener
+ update(data; GameEvent)
Implements
LoggingListener
+ update(data: GameEvent)
Uses
1
1
ScoreFacade
-ruleset: Ruleset
-starterRuleStrategy: IRuleStrategy
-playRuleStrategy: IRuleStrategy
-showRuleStrategy: IRuleStrategy
+ score(state: EventType, segment: Segment): GameEvent
uses
1
1
RuleStrategyFactory
+ getStrategy(ruleStrategy: string): IRuleStrategy
creates
1
3
Ruleset
-enabledPlay: ArrayList<RuleInfo>
-enabledShow: ArrayList<RuleInfo>
-parseConfig(): boolean
+getEnabled(EventType): ArrayList<RuleInfo>
RuleInfo
-rule: RuleType
-pointValue: ArrayList<Integer>
-pointValue: ArrayList<Integer>
<<interface>>
IRuleStrategy
+ validateRule(s: Segment): GameEvent
+ validateRule(s: Segment): GameEvent
CompositeRuleStrategy
+ validateRule(s: Segment): GameEvent
+ add()
+ remove()
+ getChild()
+ getChild()
CompositeGameRuleStrategy
+ validateRule(s: Segment): GameEvent
CompositePlayRuleStrategy
+ validateRule(s: Segment): GameEvent
CompositeShowRuleStrategy
CompositeShowRuleStrategy
+ validateRule(s: Segment): GameEvent
uses
EventFactory 1
+ createEvent(hand: Hand, player: int, eventType: String, cardTotal: int)
+ createEvent(hand: Hand, player: int, eventType: String, playerScore: int, deltaScore: int, ruleType: String)
+ createCompositeEvent()
uses
TotalPointsRuleStrategy
-pointValue: int
+ validateRule(s: Segment): GameEvent
RunRuleStrategy
-pointValue: int
+ validateRule(s: Segment): GameEvent
PairRuleStrategy
-pointValue: int
+ validateRule(s: Segment): GameEvent
FifteenRuleStrategy
-pointValue: int
+ validateRule(s: Segment): GameEvent
FlushRuleStrategy
-pointValue: int
+ validateRule(s: Segment): GameEvent
LastCardRuleStrategy
-pointValue: int
+ validateRule(s: Segment): GameEvent
JackRuleStrategy
-pointValue: int
+ validateRule(s: Segment): GameEvent
<<enumeration>>
EventType
PLAY
START
SCORE
SHOW
DEAL
DISCARD
<<enumeration>>
PlayerType
P0
P1
creates
GameEvent
-hand: Hand
<<enumeration>>
RuleType
RUN
PAIR
STARTER
FLUSH
LASTCARD
FIFTEEN
THIRTYONE
1
PlayEvent
cardTotal: int
CompositeScoreEvent
eventScore: int
playerScore: int
getScore(): int
add(scoreEvent: ScoreEvent)
ScoreEvent
deltaScore: int
playerScore: int
getScore(): int