RECYCLEd CardStock Documentation

Release 0.5

Mark Goadrich

Jul 17, 2023

Contents:

CHAPTER 1

Overview

CardStock is a General Game Playing engine for card games implemented in C#. Games are written in RECYCLE, a card game description language, and then simulations are run with random, simple, and complex AI players. CardStock can then analyze the games to determine heuristics about the games such as fairness, decisiveness, drama, or clarity, and generate transcripts of each simulation for further study.

There are currently 30 games coded in RECYCLE, from genres such as press-your-luck, fishing, adding, matching, draw-and-discard, and trick-taking games.

Attention: We are currently in the process of abstracting and refactoring CardStock to allow for new game functionality and setting up a modular system for a tournament of AI players. Please check back for further progress on these issues.

CHAPTER 2

Requirements

Visual Studio

- Mac
- Windows

CHAPTER 3

Code

Source code available at Github

3.1 RECYCLE

RECYCLE is a game description language that allows for an algorithmic representation of common core mechanisms and elements of card games. RECYCLE stands for REcursive CYclic Card game LanguagE, referring to the primary feature of the language: the recursion of game stages containing cycles of player turns. The language resembles the LISP programming language, often having a large number of nested instructions that control the flow of the games.

The process of writing game rules in a natural language can be fraught with ambiguities, often necessitating clarifications after publication. Encoding a game in RECYCLE can be useful for illuminating the underlying formal structure of a game design, providing insight into avenues for targeted or large-scale refinement, and resolving potential ambiguities.

In RECYCLE, function names are all lowercase, data types are capitalized, and Strings are all caps.

3.1.1 Base Types

There are four main base elements in RECYCLE, *String*, *Integer*, *Card*, and *Boolean*.

String

Strings are used for text within RECYCLE games. They are composed of all capital letters. Strings are the name portion of a *CardCollection*, and the key/value pairs of a *Card*.

```
STOCK HAND PILE TRICK SCORE GREEN SUIT HEARTS
```

The cardatt function also will return a String. It is used to look up the value stored a *Card* for a given key. If the key is not found or the card does not exist, this will return the empty String “”.

```
(cardatt [String] [Card])
```

Integer

Integer literals are contiguous sequences of digits from 0 to 9. Currently only positive integers are supported with literals, but negative integers can be acheived through various actions.

```
0 23 999
```

Standard Integer operators can be applied to Integers to calculate new Integers, with addition, subtraction, multiplication, integer division (//) and modular division (mod).

```
(+ [Integer] [Integer])
(- [Integer] [Integer])
(* [Integer] [Integer])
(// [Integer] [Integer])
(mod [Integer] [Integer])
```

Three functions will return Integers. First, a *Card* can be scored using the values mapped through a *PointMap*.

```
(score [Card] using [PointMap])
```

Similarly, each *Card* in a *CardCollection* can be individually scored with a *PointMap* and then these scores are summed.

```
(sum [CardCollection] using [PointMap])
```

The size of a *CardCollection* can be calculated and returned as an Integer.

```
(size [CardCollection])
```

Integers are stored as part of the game, belonging either to the *Game*, a *Player*, or a *Team*. These **IntegerStorage_** locations are named with a *String*.

```
([Game | Player | Team] sto [String])
```

Card

A card is a set of maps from a *String* key to a *String* value, such as RANK => KING, COLOR => BLUE, and VALUE => FIVE.

A card can never be directly described, but is created through the *CreateDeck* setup and referenced through locations in a *CardCollection*.

```
(top [CardCollection])
(bottom [CardCollection])
([Integer] [CardCollection])
```

Besides using references to individual specific cards in the *CardCollection*, two functions can find either the minimum or maximum card in a collection when given a *PointMap* from the card dictionaries to an integer. If there is a tie, the max or min is decided randomly among all tied cards.

```
(max [CardCollection] using [PointMap])
(min [CardCollection] using [PointMap])
```

Finally, a virtual card (for example from a minimum or union operation) can be converted into an actual card, so that any move operation moves the card in the *CardCollection* to which it belongs.

```
(actual [Card])
```

Boolean

Booleans in RECYCLE comprise the standard True and False, derived mainly from comparisons between other data types, or conjunctions and disjunctions of other Booleans. They are only evaluated, never explicitly stated as literal True or False.

```
(and [Boolean] [Boolean]+)
(or [Boolean] [Boolean]+)
(not [Boolean])

(> [Integer] [Integer])
(< [Integer] [Integer])
(>= [Integer] [Integer])
(<= [Integer] [Integer])
(== [Integer] [Integer])
(!= [Integer] [Integer])

(== [Card] [Card])
(!= [Card] [Card])

(== [String] [String])
(!= [String] [String])

(== [Player] [Player])
(!= [Player] [Player])

(== [Team] [Team])
(!= [Team] [Team])
```

3.1.2 Owners

There are three main Owners of data in Recycle: the *Game*, each *Player*, and each *Team*. The *Player* and *Team* types are more specific types of *Owners*, allowing different functionality.

Game

The Game holds storage for both *Integer* or *CardCollection* data. These are referenced by a *String* name. For example an *Integer* storage for the number of total chips in the game could be

```
(game sto CHIPS)
```

And a *CardCollection* for the stock of face-down cards would be

```
(game iloc STOCK)
```

Player

As above, a Player tracks storage for both *Integer* or *CardCollection* data. These are referenced by a *String* name. To reference an individual Player, we can directly refer to the turn order of a Player.

```
([Integer] player)
```

Also, based on the current turn within a stage, we can referentially talk to the current, previous, and next player. Turn order is determined clock-wise, and the previous player will always use this turn order. The next player also uses this turn order by default, but could be altered within this stage by a *NextAction* queueing up a different player to go next.

```
(current player)
(previous player)
(next player)
```

A Player can also be found by determining the owner of a *Card*.

```
(owner [Card])
```

Team

As above, a Player tracks storage for both *Integer* or *CardCollection* data. These are referenced by a *String* name. To reference an individual Team, we can directly refer to the turn order of a Team.

```
([Integer] team)
```

Also, based on the current turn within a stage, we can referentially talk to the current, previous, and next team. Turn order is determined clock-wise, and the previous team will always use this turn order. The next team also uses this turn order by default, but could be altered within this stage by a *NextAction* queueing up a different team to go next.

```
(current team)
(previous team)
(next team)
```

Finally, a Team can be found by asking a *Player* what team they are on. A *Player* can only be on one Team at a time.

```
(team [Player])
```

3.1.3 Collections

Many of the base data types can be grouped into *Collections*. Collections provide a way for *Aggregation* to iterate through for actions to be taken or booleans to be processed with each Collection element.

StringCollection

A comma-separated list of *String* primitives is a *StringCollection*.

```
(YELLOW, GREEN, BLUE, RED, WHITE)
```

IntegerCollection

Currently, the only way to write an IntegerCollection is as a range of *Integer* data, starting at a minimum value, and increasing by one up to but not including the maximum value.

```
(range [Integer] .. [Integer])
```

CardCollection

A CardCollection is an ordered list of *Card* objects, found on the *Game*, *Player*, and *Team* objects. These CardCollections can be directly accessed using their **Owner_**, the visibility modifier, and the *String* name for that CardCollection.

```
([Game | Player | Team] (vloc | iloc | hloc | mem) [String])
```

Visibility modifiers can be one of

- vloc: visible to everyone
- iloc: visible to owner, invisible to others
- hloc: invisible to everyone, including owner
- mem: copies of cards in memory, visible to all

The filter function can be used to create a CardCollection subset from another CardCollection. A *Boolean* statement will evaluate as true if an element of the original CardCollection, denoted by a *Variable*, will be included in the filter.

```
(filter [CardCollection] [Variable] [Boolean])
```

A CardCollection can be created through the union of other CardCollections, for example, we can create one CardCollection that will hold all the cards played by players to their individual TRICK CardCollections so that we can determine the highest played card.

```
(union [CardCollection]*)
```

Finally, we can access individual elements of a *CardCollectionCollection* to obtain a CardCollection, following the top, bottom, or index methodology.

```
(top [CardCollectionCollection])
(bottom [CardCollectionCollection])
([Integer] [CardCollectionCollection])
```

CardCollectionCollection

A CardCollectionCollection can be created through the tuples function. This will return subsets of the given *CardCollection*, where the *Card* elements are found to be equal according to a *PointMap*. Only those subsets of size equal to the given *Integer* will be returned.

```
(tuples [Integer] [CardCollection] 'using' [PointMap])
```

PlayerCollection

The current players of the game can be referenced as a PlayerCollection. For all players, simply use the word “player”.

Within a stage, players not equal to the current player can be referenced with

```
(other player)
```

Alternately, players can be added to a collection based on *Boolean* attributes assessed on each *Variable* from a PlayerCollection filter.

```
(filter [PlayerCollection] [Variable] [Boolean])
```

TeamCollection

The current teams of the game can be referenced as a TeamCollection. For all teams, simply use the word "team".

Within a stage, teams not equal to the current team can be referenced with

```
(other team)
```

Alternately, teams can be added to a collection based on *Boolean* attributes assessed on each *Variable* from a TeamCollection filter.

Cycle of teams, Denoted with the word "team"

```
(filter [TeamCollection] [Variable] [Boolean])
```

3.1.4 PointMap

A *Card* is a map between *String* and *String* types, therefore we need another data structure to capture *Integer* values of cards for scoring or ranking. A *PointMap* is a map between two *String* pieces and an *Integer*. The first *String* is the *Card* key and the second is the *Card* value. When applied to a *Card*, the points will be a sum of all of the key-value pairs that are found in this *Card*. PointMaps are stored in a *Variable*.

```
(put points [Variable] (([String] ([String])) [Int])
```

3.1.5 Aggregation

One of the powerful things that *Collections* allow is iteration and aggregation. Two keywords, "all", and "any", can be used with collection with varying results. Each element of the **Collection_** will be assigned to a *Variable* that can be used in the final portion.

All

When the final portion of the all aggregation is a *Boolean*, the all will also be a Boolean, constructing an AND over the individual elements.

```
(all [Collection] [Variable] [Boolean])
```

For example, the following is a *Boolean* that will be True if all players have a Hand size of zero.

```
1 (all player 'P
2     (== (size ('P iloc HAND)) 0))
```

When the final element is a **MultiAction_**, the all will become a sequence over the actions, in order of the items in the collection.

```
(all [Collection] [Variable] [MultiAction])
```

We can see this in the following code to move each player's top Trick card to the Discard pile.

```
(all player 'P
    (move (top ('P vloc TRICK))
          (top (game vloc DISCARD))))
```

When the final element is a CardCollection, the all will become a *CardCollectionCollection*.

```
(all [Collection] [Variable] [CardCollection])
```

This can be used to merge each player's individual *CardCollection* elements, such as

```
(union (all player 'P ('P vloc TRICK)))
```

When the final element is an *Integer*, the all will become a sum of those *Integer* elements.

```
(all [Collection] [Variable] [Integer])
```

This is particularly useful for *Integer* storage locations which are part of an **Owner_**.

TODO FIND EXAMPLE!

Any

When the final portion of the any aggregation is a *Boolean*, the any will also be a Boolean, constructing an ON over the individual elements.

```
(any [Collection] [Variable] [Boolean])
```

For example, the following is a *Boolean* that will be True if any player has Points greater than 10.

```
(any player 'P
    (> ('P sto POINTS) 10))
```

When the final element is a **MultiAction_**, the any will become a choice over the actions.

```
(any [Collection] [Variable] [MultiAction])
```

For example, the following is how a player can choose to play any *Card* in their Hand to the current Trick of a trick-taking game.

```
(any ((current player) iloc HAND) 'AC
    (move 'AC
          (top ((current player) vloc TRICK))))
```

3.1.6 Variable

Variables, like *String* constants, must be all caps. They also must begin with a ‘ character.

```
'C 'AC 'COLOR 'P
```

Variables can be implicitly assigned from a **Collection_** inside the filter function or *Aggregation* functions.

There are two other ways to create Variables, with *declare* or *let*.

declare

The declare function must be at the beginning of a RECYCLE program. It is useful for creating program-wide constants or data that cannot be altered through the game. The data can be anything explicitly defined, commonly *String* and *Integer*, or a *StringCollection* type.

```
(declare [Type] [Variable])
```

let

The let function is a local declaration of a Variable, followed by a segment of code where this Variable will be valid.

```
(let [Type] [Variable] [Expression])
```

3.1.7 Action

Actions are the way that RECYCLE allows either the players or the game to update the data structures and rearrange the cards in the game.

TeamCreateAction

Teams can be created at any time during the game, and must be created in the initialization section of the game. The following code will make four teams, one for each player, in a four-person game. Players are indexed starting at 0.

```
1 (create teams (0) (1) (2) (3))
```

To add more than one player to a team, write a comma-separated list with each team member. This code will create two teams in a four-person game, where team members are seated opposite each other.

```
1 (create teams (0, 2) (1, 3))
```

ShuffleAction

A *CardCollection* can be shuffled at any time into a new random permutation of the *Card* objects.

```
(shuffle [CardCollection])
```

CardMoveAction

Once created, *Card* objects can be moved from location to location with the move action. The first *Card* must not refer to a memory location, and the second card cannot be a memory or a virtual location.

```
(move [Card] [Card])
```

CardRememberAction

Card objects can also be copied into memory, for example to remember which card was led, or what suit is trump. The second *Card* must refer to a memory location.

```
(remember [Card] [Card])
```

CardForgetAction

Since memory locations hold *Card* objects that are copies, they should not be moved but instead forgotten when they are no longer needed.

```
(forget [Card])
```

IntAction

IntegerStorage locations can be changed in three ways. We can set the storage to be a particular *Integer*, increment the current value by an *Integer*, or decrement the current value by an *Integer*.

```
(set [IntegerStorage] [Integer])
(inc [IntegerStorage] [Integer])
(dec [IntegerStorage] [Integer])
```

NextAction

The order in the current cycle can be altered in two ways. The first is to change the player that is queued to go next. This can be the current player, which will give the player another turn, the previous player to reverse the play direction, or the owner of a particular *Card*, such as the winning card of a trick.

```
(cycle next (owner [Card]))
(cycle next current)
(cycle next previous)
```

SetPlayerAction

Second, the current player in the cycle can be altered immediately with the following similar actions.

```
(cycle current (owner [Card]))
(cycle current next)
(cycle current previous)
```

TurnAction

A player sometimes needs the opportunity to pass. This Action is a way to have the player taken an action that makes no change to the game state.

```
(turn pass)
```

RepeatAction

Actions can be repeated with the repeat action. The action will be repeated for an *Integer* number of times.

```
(repeat [Integer] [Action])
```

One additional way to repeat an action is the "repeat all" for a **MoveAction_**. This will move cards one by one until there are no more *Card* objects in the first location.

```
(repeat all [MoveAction])
```

3.1.8 GameFlow

ConditionalAction

Action blocks can also be prefaced with a boolean condition to make the execution of the action dependent on the current state of the game.

```
([Boolean] [Action])
```

Do

Action blocks can be combined to form a sequence of actions. These can also be *ConditionalAction* objects. The aggregate of these actions is called a *Do*, and can also have nested inside more *Do* blocks. These actions will be executed one after another in order from top to bottom.

```
(do ([ConditionalAction | Action | Do]*))
```

Choice

A *Choice* block is a way to set up options for the player in the game. Instead of operating sequentially, the *Action* objects found to be valid based on their conditions will be grouped and presented to the player, who then must make a choice among them for the game to proceed. A *Do* can be within a *Choice*, giving the player an option of choosing a set of sequential actions.

```
(choice ([ConditionalAction | Action | Do]*))
```

Stage

A *Stage* block activates either a *Player* or *Team* cycle. The components of a Stage will be evaluated, with each member of the cycle becoming the "current" member, until the *Boolean* end condition is met. The order of members in the cycle can be altered with the *NextAction* and *SetPlayerAction* described above.

```
(stage player [Boolean] [Do | Choice | Stage]*)
(stage team [Boolean] [Do | Choice | Stage]*)
```

3.1.9 Setup

Each game begins with a *Setup* section, following any *Declare* statements. The *Setup* section includes a *CreatePlayers* action, a *CreateTeams* action, and at least one *CreateDeck* action. Multiple decks can be added with either multiple *CreateDeck* actions or through a *RepeatAction* containing a *CreateDeck* action.

CreatePlayers

In the *Setup*, each player is created identical, so we only need to know the number of players in the game to create each player. A *Player* is placed into the player cycle.

```
(create players [Integer])
```

CreateTeams

See *TeamCreateAction* above.

CreateDeck

Card objects are created and placed in a *CardCollection* with the *CreateDeck* function. A single key can be listed for a *Card*, followed by all values of that attribute, and a *Card* will be made with one of each value. The following code will make three *Card* objects.

```
(create deck (game iloc STOCK) (deck (COLOR (RED, BLUE, GREEN))))
```

More complicated decks can be made by adding more keys. If multiple keys are listed, each with their own values, then a *Card* will be created for each permutation of these values. Here, will will create six *Card* objects, RED-SMALL, RED-LARGE, BLUE-SMALL, BLUE-LARGE, GREEN-SMALL, and GREEN-LARGE.

```
(create deck (game iloc STOCK) (deck (COLOR (RED, BLUE, GREEN))
                                     (SIZE (SMALL, LARGE))))
```

Keys can also be nested inside as values inside other key lists. The following code is used to make a full 52 *Card* deck, with RANK, COLOR, and SUIT keys.

```
(create deck (game iloc STOCK) (deck (RANK (ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
                                           EIGHT, NINE, TEN, JACK, QUEEN, KING))
                                     (COLOR (RED (SUIT (HEARTS, DIAMONDS)))
                                            (BLACK (SUIT (SPADES, CLUBS))))))
```

3.1.10 Scoring

The Scoring section of the game details how to determine the ranking of players once the game is complete. The scores can be based on any *Integer* in the game, commonly an IntegerStorage, but possibly the size of a CardCollection or any other Integer that can be calculated. The (current player) can be used to denote that this scoring will be applied to each player in the game.

You can either sort these scores from highest to lowest so that the max is regarded as the best score, or lowest to highest so that the min is the best score.

```
(scoring max [Integer])
(scoring min [Integer])
```

3.1.11 Example

The following is an example game written in RECYCLE called Agram. Agram is a simple Nigerian trick-taking card game for 2 to 6 players. Players are dealt six cards from a reduced French deck, and play six tricks. To win a trick,

players must follow the suit of the lead player with a higher card; there is no trump suit. The object of the game is to win the last trick.

```
;; Agram
;;
;; https://www.pagat.com/last/agram.html

(game
 (declare 4 'NUMP)
 (setup
  (create players 'NUMP)
  (create teams (0) (1) (2) (3))

  ;; Create the deck source
  (create deck (game iloc STOCK) (deck (RANK (THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN))
                                       (COLOR (RED (SUIT (HEARTS, DIAMONDS)))
                                              (BLACK (SUIT (SPADES, CLUBS))))))
  (create deck (game iloc STOCK) (deck (RANK (ACE))
                                       (COLOR (RED (SUIT (HEARTS, DIAMONDS)))
                                              (BLACK (SUIT (CLUBS)))))))


 ;; Shuffle and deal each player 6 cards
 (do
     (
      (shuffle (game iloc STOCK))
      (all player 'P
           (repeat 6
                   (move (top (game iloc STOCK))
                         (top ('P iloc HAND)))))))

 ;; players play a round 6 times
 (stage player
        (end
         (all player 'P
              (== (size ('P iloc HAND)) 0)))

        ;; players play a hand once
        (stage player
               (end
                (all player 'P
                     (> (size ('P vloc TRICK)) 0)))

               (choice
                (

                 ;; if following player cannot follow SUIT
                 ;;   play any card, and end your turn
                 ((and (== (size (game mem LEAD)) 1)
                       (== (size (filter ((current player) iloc HAND) 'C
                                         (== (cardatt SUIT 'C)
                                             (cardatt SUIT (top (game mem LEAD))))))
0))
                  (any ((current player) iloc HAND) 'AC
                       (move 'AC
                             (top ((current player) vloc TRICK)))))

```

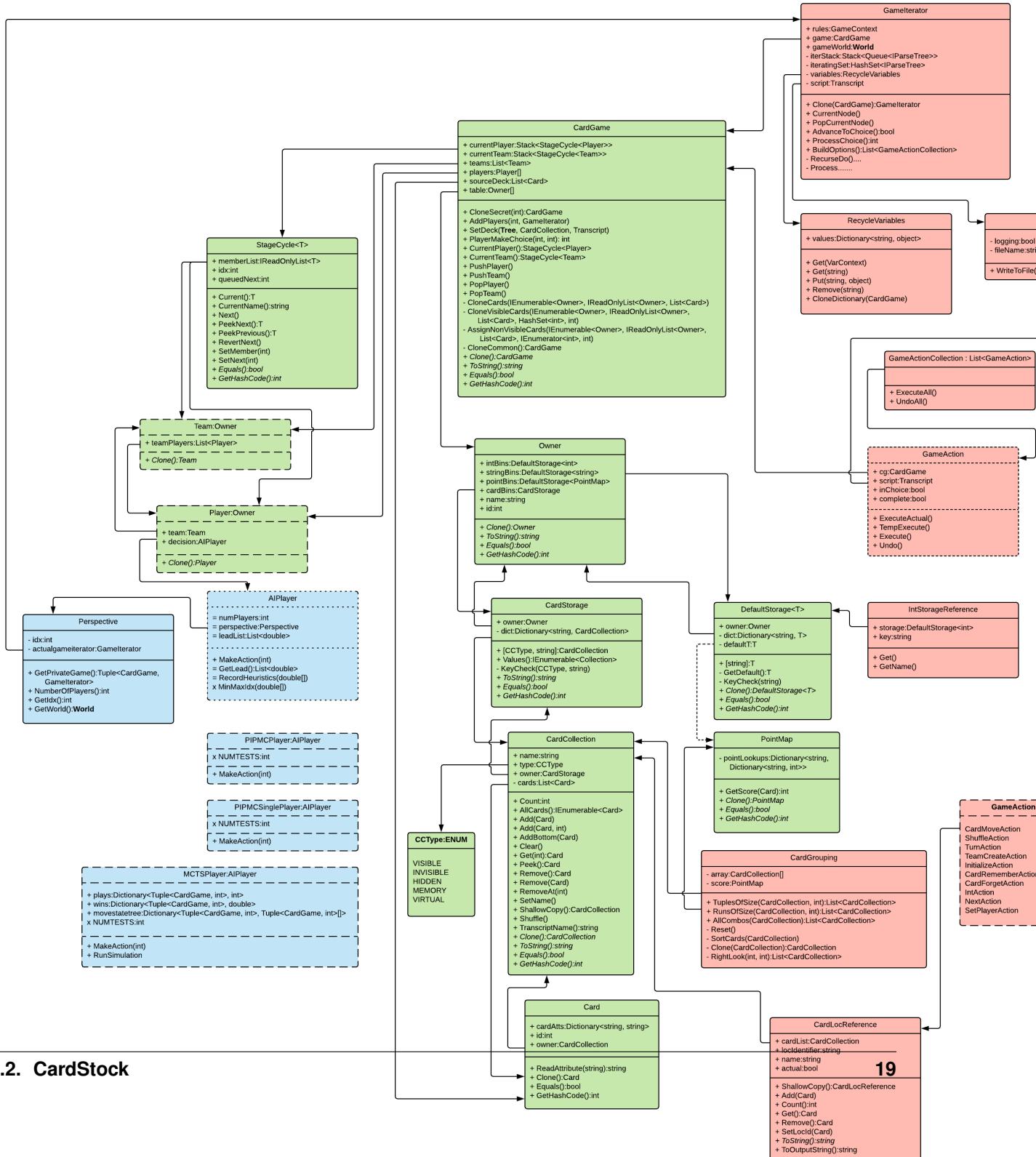
(continues on next page)

(continued from previous page)

```
                ;; if following player and can follow SUIT
                ;;   play any card that follows SUIT, and end your turn
                (any (filter ((current player) iloc HAND) 'T
                             (== (cardatt SUIT 'T)
                                 (cardatt SUIT (top (game mem LEAD)))))
                     'C
                               ((== (size (game mem LEAD)) 1)
                                (move 'C
                                      (top ((current player) vloc TRICK)))))

                ;; if first player, play any card, remember it in the lead spot, and end your turn
                ((== (size (game mem LEAD)) 0)
                 (any ((current player) iloc HAND) 'AC
                      (do
                          (
                           (move 'AC
                                 (top ((current player) vloc TRICK)))
                           (remember (top ((current player) vloc TRICK))
                                     (top (game mem LEAD))))))))

        ;; after players play hand, computer wraps up trick
        (do
            (
             ;; solidfy card precedence
             (put points 'PRECEDENCE
                  (
                   ((SUIT (cardatt SUIT (top (game mem LEAD)))) 100)
                   ((RANK (ACE)) 14)
                   ((RANK (TEN)) 10)
                   ((RANK (NINE)) 9)
                   ((RANK (EIGHT)) 8)
                   ((RANK (SEVEN)) 7)
                   ((RANK (SIX)) 6)
                   ((RANK (FIVE)) 5)
                   ((RANK (FOUR)) 4)
                   ((RANK (THREE)) 3)))

             ;; determine who won the hand, set them first next time
             (forget (top (game mem LEAD)))

             (cycle next (owner (max (union (all player 'P ('P vloc TRICK))) using 'PRECEDENCE)))

             (all player 'P
                  (move (top ('P vloc TRICK))
                        (top (game vloc DISCARD))))

             ;; if that was the last round, give the winner a point
             ((all player 'P
                   (== (size ('P iloc HAND)) 0))
              (inc ((next player) sto SCORE) 1)))))

 (scoring max ((current player) sto SCORE)))
```

3.2 CardStock

RECYCLEd CardStock
A General Card Game Playing Engine



3.2.1 CardEngine

The data structure that holds the game state data

3.2.2 FreezeFrame

GameIterator

Transcript

3.2.3 Heuristics

Scorer

World

Heuristic

MeaningfulMoves

Variance

Fairness

Drama

Decisiveness

Clarity

Coolness

3.2.4 Running Experiments

Open "CardStockXam" project. Write up your game in RECYCLE. Alter the Program.cs class to create an Experiment object for your game. Run the program in either Release or Debug mode. Choose "Release Mode" to only see the results, or "Debug Mode" to see all game actions (better logs in the future).

3.3 AIPlayer

3.3.1 RandomPlayer

Random players are useful in many situations for CardStock. They can be deployed for initial playtesting to ensure a game is robust to all player choices and unbiased toward a particular seating arrangement. They can also be employed by more advanced *AIPlayer* implementations within Monte Carlo simulations.

In *RandomPlayer*, the *MakeAction* method simply returns a uniformly randomly chosen int within the range of 0 to the number of choices given to the player.

3.3.2 PIPMCPlayer

Perfect Information Pure Monte Carlo Players are the current default AI players for CardStock.

For each potential move in **MakeAction_**, the player simulates a set number (defined in the NUMTESTS static variable) of random games. A unique subgame data structure and subgame iterator are created for each test. The subgame iterator then constructs all the possible GameActionCollection moves with the BuildOptions method, the current move actions are all executed, and then the choice is popped from the subgame iterator.

All players in the subgame are assigned to make choices randomly, and then the game is played out to completion. Each player is assigned a score based on the inverse of their rank in the player list, where 1st place is the winning player. These ranks are accumulated and averaged across all tests, and the index of the move where the current player earned the highest rank is selected and returned.

3.3.3 MCTSPlayer

Monte Carlo Tree Search

3.3.4 ISMCTSPlayer

Information Sets Monte Carlo Tree Search

AIPlayer is an abstract class to be subclassed for all of the AI. It will know the number of players in the game, have a *Perspective* which privatizes the hidden aspects of the game from this player, and a List that tracks the player's estimates of their current game position.

3.3.5 Perspective

A *Perspective* is a wrapper around a **GameIterator_** and an int which is the current player's index in the game cycle. An *AIPlayer* can ask the *Perspective* for their view of the game through the **GetPrivateGame_** method. This will return a Tuple of a CardGame and *GameIterator*, where all the private *Card* information hidden from this player is shuffled and replaced.

3.3.6 MakeAction

The *MakeAction* method is the critical method that needs to be overridden in any subclass of AIPlayer. When a choice is found in the game, the number of potential moves will be passed in. The *AIPlayer* is expected to return an int which is the index of their chosen move.

If your *AIPlayer* is being used in the *Heuristics* portion of CardStock, then within *MakeAction*, you should also create an array of doubles with the estimated inverse rank of the player for each possible move and pass this to RecordHeuristics for processing.