TAG: Tabletop Games Framework
Setup and Introduction

Contents

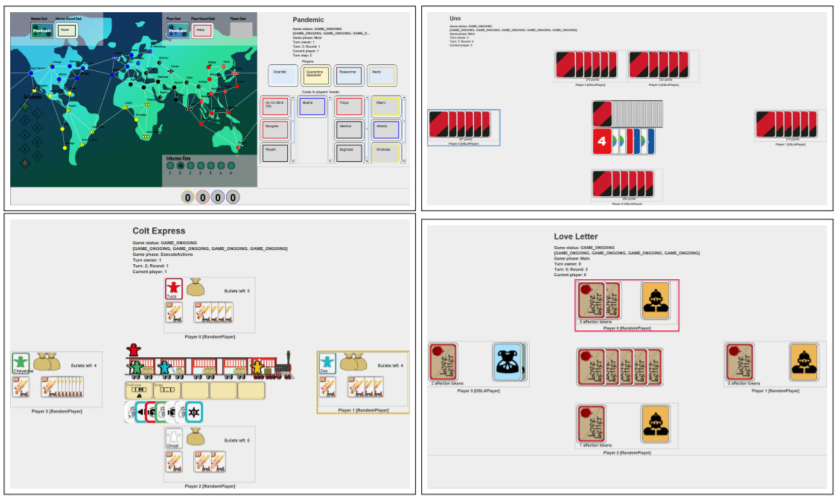
1 Framework Description

The Tabletop Games Framework (TAG)[2] promotes research into general AI in modern tabletop games, facilitating the implementation of new games and AI players, while providing analytics to capture the complexities of the challenges proposed.

We define an **action** as an independent unit of game logic that modifies a given game state towards a specific effect (e.g. player draws a card; player moves their pawn). These actions are executed by the game players and are subject to certain **rules**: units of game logic, part of a hierarchical structure (a game flow graph). Rules dictate how a given game state is modified and control the flow through the game graph (for instance, checking the end of game conditions and the turn order).

At a higher level, games can be structured in **phases**, which are time frames where specific rules apply and/or different actions are available for the players.

All tabletop games use **components** (game objects sharing certain properties), whose state is modified by actions and rules during the game. TAG includes several predefined components to ease the development of new games, such as tokens (a game piece of a particular type), dice (with N sides), cards (with text, images or numbers), counters (with a numerical value), grid and graph boards. Components can also be grouped into collections: an area groups components in a map structure in order to provide access to them using their unique IDs, while a deck is an ordered collection with specific interactions available (e.g. shuffle, draw, etc.). Both areas and decks are considered components themselves.



2 Getting started

The project requires Java with minimum version 8. In order to get the framework installed and running, follow these instructions:

1. Clone the following repository using git in a Unix or Windows terminal:

   git clone `https://github.com/GAIGResearch/TabletopGames`

   You can also download the repository or check it out with SVN. Alternatively, for cloning the repository using a graphical user interface, you can try GitHub Desktop. To make sure you are working with a consistent version of the framework, you can use the latest release instead:

   `https://github.com/GAIGResearch/TabletopGames/releases/tag/v3.1`

2. Set up your IDE to use the project. These instructions allow you to set it up with IntelliJ IDEA, but you can also use any another IDE you feel more comfortable with.
   (a) Open IntelliJ IDEA
   (b) Select 'Create New Project' ... 'from existing sources'
   (c) Select, as project location, the 'TabletopGames' folder downloaded through the previous step.
   (d) Select the **Maven** framework for import. This process should automatically set up the environment and add any project libraries as well.
   (e) Click on Finish (say 'yes' to overwrite the project folder if prompted).
3. Alternatively, open the code directly in your IDE of choice, right-click the `pom.xml` file and setup the project with the **Maven** framework. Make sure `src/main/java` is marked as *sources root.*
4. Verify that TAG can execute in the IDE:
   (a) In the IDE, open the file `src/main/java/core/Game.java` (you can navigate files on the left panel - click on *1:Project* if the folder hierarchy is not displayed).
   (b) Click on menu `Run` → `Run`, and select the *Game* class in the pop up window. Alternatively, right click the file `Game.java` and select the option `Run`.
   (c) It'll take a few seconds for the project to be compiled and then it will run a game.

3 TAG Wiki and Documentation

The first step is to inform yourself about each game - how it works, what kind of objects and actions are available, rules, etc. For this, you are recommended to first read the documentation about the framework (introduction, game definition, framework structure, etc.) in the framework's Wiki and descriptive paper:

`http://www.tabletopgames.ai/wiki/`
`https://rdgain.github.io/assets/pdf/papers/gaina2020tag.pdf`

The Wiki includes definitions of terminology to better understand the framework, and information on games currently in the framework. The TAG wiki is generally a good place to find information about the framework.

Next, this document contains exercises for you to get familiarised with the framework. You are welcome to attempt the exercises listed here in the order you prefer, although the suggestion is that you do them in order:

4 Running the framework

There are two main classes to run the framework, and also play the game yourself against the built-in AIs: `core.Game.java` and `gui.Frontend.java`. We will explore other methods in the next lab.

4.1 Game

In the `Game` class (found in the `core` package), you'll find the `main` method which can be run at the end of the file. Here, you can customise the way you run games by modifying some of the code, or supplying different *program arguments*:

Argument	Default	Other values
gameType	Jaipur	games.GameType enum values
useGUI	true	false, true
turnPause	0	positive Integer
seed	System.currentTimeMillis()	Long

Other setup steps are identified in the comments in this method:

4.1.1 Players

A few lines follow to initialise a list of **players**. This variable is a Java `ArrayList` data type, an ordered list. It needs to contain Java objects representing the players that will take part in the game. All objects added to this list need to extend the `AbstractPlayer` class. Multiple copies of the same player can be added.

The size of the list will indicate the number of players for the game. If this doesn't correspond with the number of players accepted by the game being run, an error will occur. The easiest way to check the number of players is in the class `games.GameType.java`. This class contains the descriptors of all the games in the framework which can be run, including:

- The number of players they accept
- A list of categories the game belongs to
- A list of mechanics the game uses
- References to the classes which actually implement the game functionality.

The AI players which currently exist in the framework can be found in the `players` package, and the basic ones can be instantiated for the players ArrayList as follows:

AI Player	Java Construction Code	Optional Constructor Arguments
FirstActionPlayer	`new FirstActionPlayer()`	-
Random	`new RandomPlayer ()`	`java.util.Random`
OSLA	`new OSLAPlayer ()`	`java.util.Random`
RHEA	`new RMHCPlayer()`	random seed, `RMHCParams`, `IStateHeuristic`
MCTS	`new MCTSPlayer()`	random seed, `MCTSParams`, `IStateHeuristic`
HumanGUIPlayer	`new HumanGUIPlayer(ac)`	-
HumanConsolePlayer	`new HumanConsolePlayer()`	-

A short description of these players is as follows (more details can be found in the Wiki at `http://www.tabletopgames.ai/wiki/agents/implemented`):

- **FirstActionPlayer**: always chooses the first action in the list of available action in a game state.
- **Random**: executes a random action at every step.
- **OSLA**: One Step Look Ahead. Looks one step into the future, by simulating the effect of all possible actions in the current game state and picks the one leading to the best next state (according to game-specific heuristic).
- **RHEA**: Rolling Horizon Evolutionary Algorithm[3]. Uses evolutionary algorithms to evolve a sequence of actions that leads to the best state 10 steps into the future, and picks the first action in the sequence.
- **MCTS**: Monte Carlo Tree Search[1]. Incrementally builds a game tree by simulating the effect of different actions starting from the current state up to 10 steps into the future, and picks the next immediate action that statistically leads to a better future state.
- **HumanGUIPlayer**: allows a human to play in the Graphical User Interface (GUI).
- **HumanConsolePlayer**: allows a human to play via the console.

Most of these agents use heuristics to evaluate how good a game state is. These heuristics are most often game-specific (each game state implements these, see Exercise 2), but can also be custom, which are found in the package `players.heuristics`.

4.1.2 Game parameters

A path to a JSON file containing a specific parameter configuration for the game can be supplied within the `gameParams` variable. Examples can be found in ‘data/pandemic/param-config-x.json‘ files.

4.1.3 Run

The last step is to choose the run method for your game. The main one is `runOne`:

```
runOne(GameType.valueOf(gameType), gameParams, players, seed, false, null,
    useGUI ? ac : null, turnPause);
```

The arguments it takes are:

- Game to run: the data time of this argument is `GameType`, so you can check the class previously mentioned for the possible values (i.e. games available in the framework).
- Game parameters: path to custom JSON parameter configuration file (if null, default values in the parameters class are used instead).
- Players: players to take part in the game, defined as previously detailed.
- Seed: random seed to use for the run.
- `randomizeParameters`: if true, the parameters of the game will be randomised (more on this next week!); if false, the game will run with default parameters; keep it to **false** for now.
- `listeners`: game listeners can be attached to react to specific events or record data. More on this next week!
- useGUI: If true, the action controller object is supplied, which receives player input for use with the Graphical User Interface. Otherwise the game is run without visuals.
- Turn pause: time to pause between player turns (to slow down GUI runs).

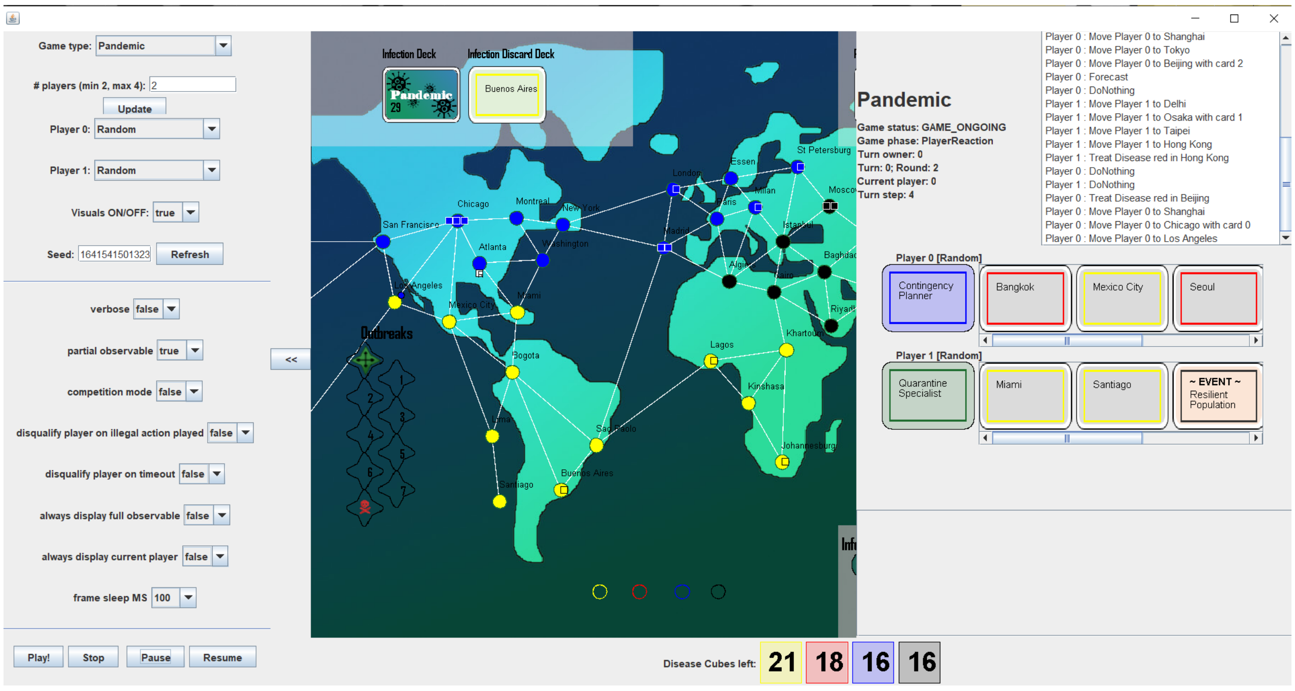
Alternatively, a `runMany` method exists to run multiple games at once with the same set of players. This method will take a list of games as the first argument, instead of just one. Check out the method definition in the `Game` class for more information.

4.2 Frontend

The second (and easier!) way to run the framework is through the `gui.Frontend.java` class. When run, this class produces a rough GUI which can be used to set different options and run games directly (without the need to modify any code):

- Number of players for the run.

- For each player, select the player type (including human player options). Optionally, you can modify the player's parameters.
- Game to be run. Optionally, you can modify the game's parameters (if the game implements *tunable* parameters - more on this next week!).
- Visuals can be toggled on or off.
- Set random seed.
- Set core parameters (including toggling partial observability and competition mode, setting display options or modifying the frame rate).
- Play, pause, resume and stop game buttons.



Game type: Pandemic
# players (min 2, max 4): 2
Player 0: Random
Player 1: Random
Visuals ON/OFF: true
Seed: 1641541501323
verbose false
partial observable true
competition mode false
disqualify player on illegal action played false
disqualify player on timeout false
always display full observable false
always display current player false
frame sleep MS 100

Pandemic

Game status: GAME_ONGOING
Game phase: PlayerReaction
Turn owner: 0
Turn: 0; Round: 2
Current player: 0
Turn step: 4

```
Player 0 : Move Player 0 to Shanghai
Player 0 : Move Player 0 to Tokyo
Player 0 : Move Player 0 to Beijing with card 2
Player 0 : Forecast
Player 0 : DoNothing
Player 1 : Move Player 1 to Delhi
Player 1 : Move Player 1 to Osaka with card 1
Player 1 : Move Player 1 to Taipei
Player 1 : Move Player 1 to Hong Kong
Player 1 : Treat Disease red in Hong Kong
Player 0 : DoNothing
Player 1 : DoNothing
Player 0 : Treat Disease red in Beijing
Player 0 : Move Player 0 to Shanghai
Player 0 : Move Player 0 to Chicago with card 0
Player 0 : Move Player 0 to Los Angeles
```

Player 0 [Random]: Contingency Planner | Bangkok | Mexico City | Seoul

Player 1 [Random]: Quarantine Specialist | Miami | Santiago | ~ EVENT ~ Resilient Population

Disease Cubes left: 21 18 16 16

Note: there may be resolution issues with running this option (which may prevent you from seeing all the options or even running games at all. In this case, please revert to the first option of running the framework - this frontend is a rough solution and still in experimental stages!

5 Game States and Forward Models

Two of the core classes implemented by all games in the framework are the **game state** and the **forward model**. We will explore implementations of these for some of the games next.

5.1 Game State

All game states for all games extend from `core.AbstractGameState`, which provides a lot of basic functionality and templates for further functionality (i.e. methods which can be implemented by subclasses).

A game state contains all information needed to describe the current state of the game, a moment in time: components, other variables or groupings. It does not implement any game functionality, it is simply a container class. Any functions implemented are simple accessor functions, to give wider access to the information contained.

The state is going to be modified by the forward model (game rules) and player actions, which is where the main functionality for a game is actually implemented.

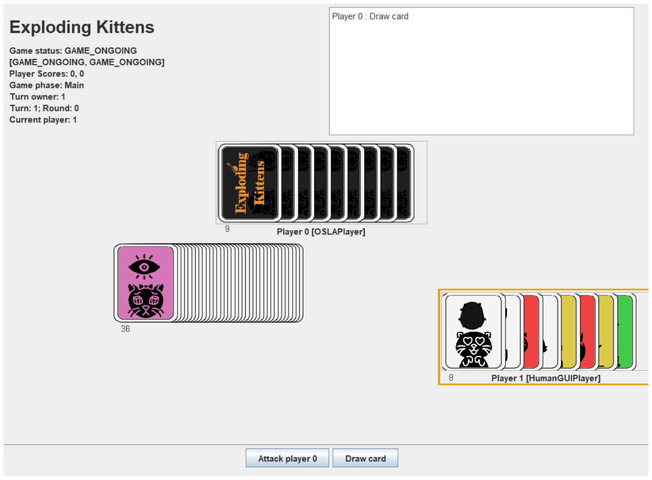
The most important method on a game state is the `copy` method:

```
AbstractGameState _copy(int playerId)
```

This method has 2 purposes:

1. It creates *deep* copies of the game state: AI players can then make simulations within copies without affecting game objects in the original state.
2. If the game is partially observable (some information is hidden from the players, e.g. cards in hand), this method should hide components not visible by the player passed as an argument to this function. **Note:** this argument could be -1, in which case nothing in the state should be hidden!

Game Example: Exploding Kittens In "Exploding Kittens", 2 to 5 players try to avoid drawing an *Exploding Kitten* card while collecting other useful cards. Each card gives a player access to unique actions to modify the game-state, e.g. selecting the player taking a turn next and shuffling the deck. This game features stochasticity, partial observability and a dynamic turn order with out-of-turn actions. The game state keeps an action stack so that players have the chance to react to cards played by others using a *Nope* card. A *Nope* card cancels the most recent effect, but can itself be cancelled by another *Nope* card. Full rules PDF: `https://www.fgbradleys.com/rules/rules2/ExplodingKittens-rules.pdf`.


Exploding Kittens

```
Game status: GAME_ONGOING
[GAME_ONGOING, GAME_ONGOING]
Player Scores: 0, 0
Game phase: Main
Turn owner: 1
Turn: 1; Round: 0
Current player: 1
```

```
Player 0 : Draw card
```

In the game state class, this game tracks the following:

- `playerHandCards`: List of (partial observable) deck of cards in hand for each player.
- `drawPile`: (Partial observable) deck of cards for drawing new cards.

- `discardPile`: Deck of cards for discarding played cards.
- `actionStack`: Stack of actions (which can be cancelled).
- `playerGettingAFavor`: ID of player currently getting a favour (to handle the reactive Favour action card).

Required functions implemented:

- `_getAllComponents()`: returns a list with all Components stored in the game state.
- `_copy(int playerId)`: deep copy of the game state. Further, if the state needs to be hidden for the given player, all cards in players' hands which are unknown to this player are moved to the draw pile. The unknown components in the draw pile are then shuffled, then the opponents are dealt new cards to replace those removed in the first step. This procedure ensures that, since the player doesn't know what cards the opponents hold in hand (or what cards are in the deck), those elements unknown are randomised.
- `_getHeuristicScore(int playerId)`: returns a heuristic score for the current game state, approximating its value for the given player. This heuristic works by assessing the cards in the player's hand (using hard-coded values based on card types) and returns the average card value.
- `getGameScore(int playerId)`: no score in this game, this method always returns 0.
- Equals and hashCode.

Other functions available:

- `killPlayer(int playerID)`: marks a player as dead.
- `toString()`: prints out the game state to the console.
- Getters (public) and setters (protected) for the variables.

5.2 Forward Model

The forward model encompasses the rules of a game. A copy of this model (with a different random seed) is given to all AI players, allowing them to make simulations of games in parallel to the real game being played, following the same rules. All forward models extend from the `core.AbstractForwardModel` class (similar to the game state, this provides functionality and templates for implementations of new games). New games are recommended to extend from the `core.StandardForwardModel` class instead, which oversees framework requirements and exposes an interface more easily handled by new games.

The functionality is split across several important functions:

```
void _setup(AbstractGameState state)
List<AbstractAction> _computeAvailableActions(AbstractGameState gameState)
void _beforeAction(AbstractGameState state, AbstractAction action)
void _afterAction(AbstractGameState state, AbstractAction action)
```

In detail, these methods take care of specific functionality:

- **Setup:** sets up the game state passed as an argument to be the initial state of the game (e.g. creating new decks, loading data, dealing starting hands to players etc.).
- **Compute Available Actions:** returns a list of all actions available in the game state passed as an argument, for the next player to pick from.
- **Before/After action:** applies the main game rules when player decisions are taken. Here, games apply any other required game rules or events not included in action effects, check for turn/round/game end and adjust game state variables appropriately.

Game Example: Tic Tac Toe 2 players alternate placing their symbol in a N $\times$ N grid until one player completes a line, column or diagonal and wins the game; if all cells in the grid get filled up without a winner, the game is a draw. This is the simplest game included in the framework, meant to be used as a quick reference for the minimum requirements to get a game up and running.

The forward model implements the required functions as follows:

- **Setup:** sets up the grid board, according to the size defined in the games parameters (more on this next week).
- **After action:**

1. Checks for game end (a line filled with the same player's symbol, or the grid filled with no winner), and assigns game status and player result accordingly.
2. If the game has not ended, moves to the next player.

- **Compute Available Actions:** actions in this game are to place the player's symbol in an empty cell on the grid, so a list of all such actions is returned for the next player to choose from.

6 Actions and Components

Two other core elements required for a game are: **actions**, for players to be able to interact with the game, and **components**, which make up the game state and are modified by actions and other game rules (applied by the forward model). We will explore some key aspects of these next.

6.1 Action

Actions are things the players can do in the game (e.g. move a pawn, play a card, roll a die). In general, in tabletop game rules descriptions, these are usually listed under a special category for player actions as well. All actions extend from the `AbstractAction` class.

The functionality for actions is split across several important functions, similar to the forward model:

- **Execute**: changes the given game state by applying the effect of the action in the game (e.g. change the position of the current player's pawn from one square on the board to another).
- **Copy**: returns an exact (deep) copy of the action.
- **Equals and HashCode**: required for action comparisons (the HashCode function always returns an integer unique to the object, given its current state).

When writing Actions, there is one important aspect to keep in mind:

- **No references**: there should be no references stored in the action classes (unless you are absolutely certain they are immutable). Due to the fact that game state copies are created often for simulations, object references will not point to the correct objects in all the copies. If component references are required, you can store the component ID in the action class instead, and then retrieve the correct object instance from the given game state using the `getComponentByID` method.

Complex actions (e.g. multi-step actions that require several decisions to be taken) can be implemented via the `Extended Sequence` framework. You can find more information on this in Lecture 1 or in the Wiki: `http://www.tabletopgames.ai/wiki/conventions/IExtended`.

Core actions available in the framework:

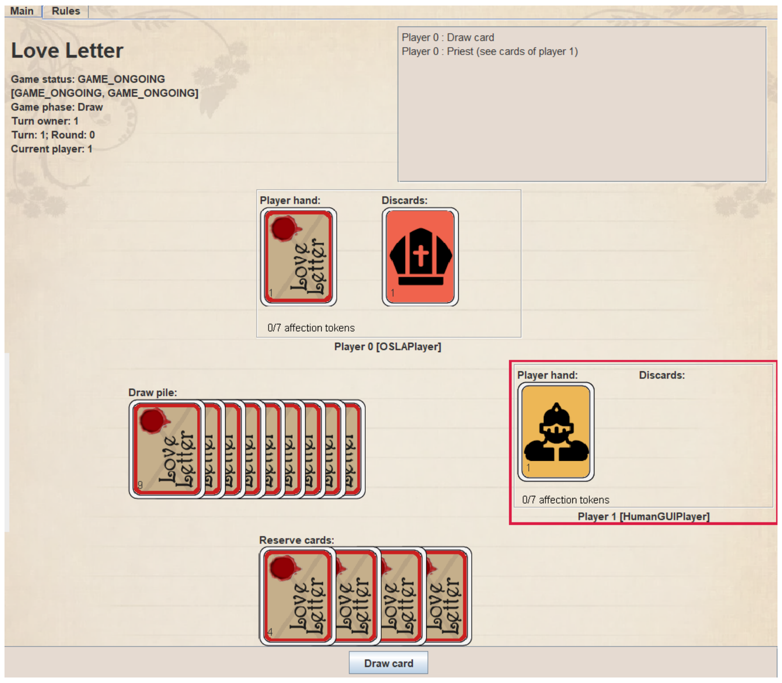
Action	Effect
DoNothing	does nothing (equivalent to "pass")
DrawCard	moves a card from one deck to another (given by component ID), specifying the card's source and destination index
DrawComponents	moves several components (given by IDs) from one deck to another
RearrangeDeckOfCards	changes the order of cards in a deck
RemoveComponentFromDeck	removes a component from a deck
ReplaceComponents	move first n components from one deck to another, then draw the same number of components to replace them
SetGridValueAction	sets the value of a cell in a grid board (given by component ID)

Game Example: Love Letter In "Love Letter", 2 to 4 players start the game with one card each, representing a character, a value and a unique effect. A second card is drawn at the start of each turn, one of which must be played afterwards. After the last card of the deck is drawn, the player with the highest valued card wins the current round. Players can be eliminated throughout the round, and the round also ends when only one player remains, who wins. A player wins the game after winning 5 rounds. Full rules PDF: `https://www.fgbradleys.com/rules/rules2/LoveLetter-rules.pdf`

The actions implemented for this game are as follows:

- `DrawCard`: draws a card for the current player (used at the start of each turn).
- One action per card type in the game: `Guard`, `Priest`, `Baron`, `Handmaid`, `Prince`, `King`, `Countess` and `Princess` - all have different effects when played, according to game rules. These actions will be available to the players depending on the cards they hold in hand.

As an example, the `PrincessAction` class extends the `core.actions.DrawCard` class (using the default functionality to discard the card once played), adding on top the effect of playing the Princess card, which eliminates the player that played it (setting the status of this player in the given game state in the `execute` function).



Love Letter
Game status: GAME_ONGOING
[GAME_ONGOING, GAME_ONGOING]
Game phase: Draw
Turn owner: 1
Turn: 1; Round: 0
Current player: 1

Player 0 : Draw card
Player 0 : Priest (see cards of player 1)

0/7 affection tokens
Player 0 [OSLAPlayer]

0/7 affection tokens
Player 1 [HumanGUIPlayer]

6.2 Component

Components are unit parts or concepts of a game. All components extend the `core.components.Component` class and they always call super constructor: this assigns a unique ID to each component in the game.

Important method: `copy`. Every component needs to be able to create a deep copy of itself. The copy needs to have the same component ID (different objects across game state copies can be matched by this ID). It's generally good practice to not store object references here either, but since the copies are going to be deep and keeping component IDs intact, this is not a strict requirement.

Core components available in the framework:

Component	**Definition**
Component	base class for all components
Counter	used for counting; has a minimum, maximum and current value
Dice	used to roll dice; has a number of sides, and stores the last value rolled
Token	used for game pieces that need to move around in physical space; has a type
Card	base class for cards
FrenchCard	definition for French cards (with number/type and suite, i.e. diamonds, hearts, spades and clubs)
BoardNode	defines a node in a graph board, with information about its neighbours
GraphBoard	a collection of board nodes
GridBoard	a grid containing any other components
Area	a map collection, mapping from component ID to component object
Deck	a list collection of other components; typical methods for drawing, shuffling etc.
PartialObservableDeck	a deck where the components inside have different visibility settings

Game Example: Dominion "Dominion" is a deck-building game in which players start with basic cards, and then buy cards with better values and abilities to add to their deck. It has the common structure of Euro-games of first needing to build an 'engine', and then using that engine to get victory points. A short-sighted strategy of buying victory points immediately will do very badly, against human players at least, and good agents need to learn the virtues of delayed gratification. Full rules pdf: `https://www.riograndegames.com/wp-content/uploads/2016/09/Dominion-2nd-Edition-Rules.pdf`.

This game has 2 different components implemented within which all instances are implemented:

- `DominionCard`: a card of a specific type (enumeration defined separately to reflect all card types in the game. Each card type maps to actions which are applied when the card is played, similar to "Love Letter". Each card type has several properties as well, which can be queried from the card itself (e.g. victory points, cost, reactions required etc.).
- `Gardens`: a special type of card (extends `DominionCard`, with different rules for giving victory points.

7 Running The Framework (Advanced)

Last week we saw 2 basic ways of running the TAG framework: the `Game` class, and via the `Frontend`. More detailed information from a run can be obtained by running the framework via 2 other entry points discussed in this section: the `evaluation.GameReport` and `evaluation.RoundRobinTournament` classes. Both are most easily run via command line (or with program arguments from an IDE).

7.1 The Game Report

We will look at the `GameReport` first. This class provides support for running multiple games with the same AI player, including options to:

- Run for one game, a list of specific games, or every game in the framework.
- Run for a specified number of players.
- Add one or more objects that implement the `IGameListener` interface, or one or more `AbstractMetric` objects, to log details of the games as they proceed. The default here will generate data on game characteristics like branching factor, the size of the action space, proportion of hidden information and so forth.

For full details on usage use the `--help` option on the command line or check the TAG wiki.

To run this class with your desired settings, set up program arguments in IntelliJ (see Appendix A), or run via command line. All settings for this class need to be defined as arguments to the class (rather than modifying the class itself as seen previously). All arguments have default options enabled, so you can simply run the class without any arguments as well.

See the Appendix for example arguments for the run.

7.2 The Game Listener

Several classes make up the Game Listener framework used for collecting game data. These are explained below:

`IGameListener`: listens to a Game (with a classic Observer pattern) for several events. Generally speaking, each `IGameListener` will have a list of `AbstractMetric` that it extracts from the game, to record interesting information about the event, as well as an associated `IStatisticsLogger` which decides how the data is recorded. Default implementation (recommended to use): **evaluation.listeners.GameListener**.

`evaluation.metrics.AbstractMetric`: extracts a single piece of information (Attribute) an Event object (including event type, state, action, player). The information returned can be a String, Integer, Boolean, Double, or another type of object. Specifies which events it responds to, and may be parameterised (AbstractParameterizedMetric).

`IMetricCollection`: interface for classes which include metrics as inner classes, including functionality to retrieve all possible metric objects belonging to this collection. Often game-specific.

`IStatisticLogger`: This is responsible for *how* to record data. Two default implementations (`SummaryLogger` and `FileStatsLogger`) either log summary data to the console, or write detailed data to a file (one row per event). Implementations can easily be added to send data to a database, or a GUI, for example.

`evaluation.metrics.Event.GameEvent`: defines the current set of game events that can be listened to:

```
ABOUT_TO_START, GAME_OVER, ROUND_OVER, TURN_OVER, ACTION_CHOSEN,
    ACTION_TAKEN, GAME_EVENT
```

Default generation of these events has been added to the core framework at appropriate places, and they can be overridden as required for specific games. Games may also raise a `GAME_EVENT` type whenever they wish to signal a custom event taking place.

For an example of some game-specific metrics used in reporting, see `games.dominion.stats.DominionMetrics`, or `evaluation.metrics.GameMetrics` for general-purpose metrics.

7.3 Tournaments

The `evaluation.RoundRobinTournament` class provides a convenient way to run a tournament between multiple AI players for a single game. This allows you to specify:

- The game
- The number of players
- The directory containing a set of JSON files, one for each of the agents to be included in the tournament. Details here: `http://www.tabletopgames.ai/wiki/agents/agent_JSON`.
- The number of games per match-up between players
- Whether to run in ‘exhaustive’ or ‘random’ mode. The former runs a number of games for every possible player combination across every possible position. The latter runs a total number of games, randomising the players (and their positions) in each - it is more tractable for larger numbers of players, and/or if each game takes a long time to run.
- Optionally, one or more `IGameListener`s, or one or more `IMetricCollection`s, to log details of the games as the proceed.

For full details on usage use the `--help` option on the command line.

References

[1] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Pérez Liébana, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. In *IEEE Trans. on Computational Intelligence and AI in Games*, volume 4, pages 1–43, 2014.

[2] Raluca D. Gaina, Martin Balla, Alexander Dockhorn, Raul Montoliu, and Diego Perez-Liebana. TAG: a Tabletop Games Framework. In *Proceedings of the AIIDE workshop on Experimental AI in Games*, 2020.

[3] Raluca D. Gaina, Sam Devlin, Simon M Lucas, and Diego Perez-Liebana. Rolling Horizon Evolutionary Algorithms for General Video Game Playing. *IEEE Transactions on Games*, 2021.