Lets Play Cards

Software Architecture Document

Gabriel Forgues

Surbhi Gupta

Justin Kat

Chandani Patel

Jessica Makucka

Table of Contents

1. System Overview

In the first version of Lets Play Cards (LPC), which is a desktop application of single- and multi-player card games, the popular casino game Blackjack is going to be implemented. The LPC system is going to be implemented using a three-tier architecture system:

- Presentation Tier
- Logic Tier
- Database Tier

The three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently as requirements change since the presentation, application processing (logic), and data management are logically separate processes.

Presentation tier (includes the GUI)

The presentation tier is the topmost level of the application. This tier displays information, through the GUI, that is connected to the Main Interface whose methods act as the boundary objects between the user and the system.

Application tier (logic or data access tier)

The logic tier is pulled out from the presentation tier and, as its own layer, it controls the application's functionality; it represents the game logic which contains the necessary classes and methods needed to create a game (for this version, it's blackjack).

Data tier (database or storage tier)

This tier consists of the database and the storage. Here, information is stored and retrieved through the means of a text file. This tier keeps data neutral and independent from the application and presentation tiers.

Overview Diagram:

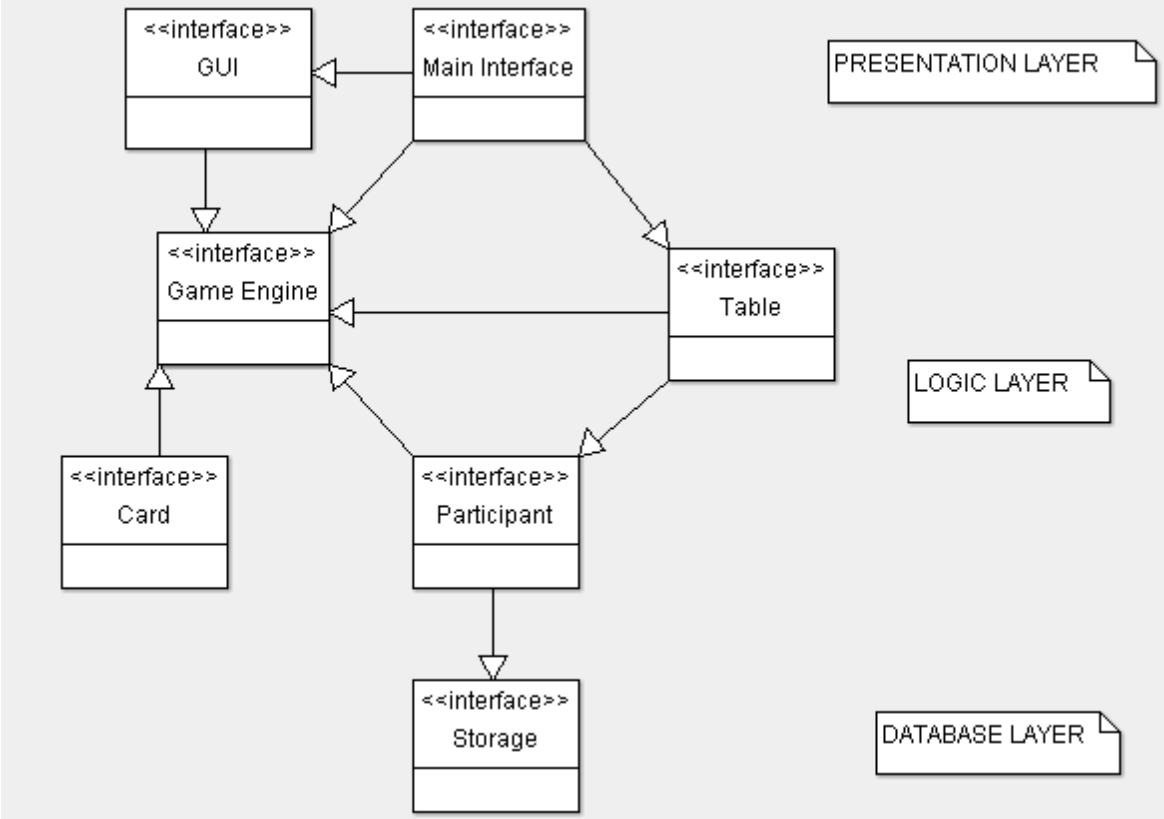

Figure 1 System Architecture Overview

2. Sub-system views

2.1 Main Interface

Decomposition Description: Interfaces the main menu's functionality

- Select Game
- Play Game
- Setup Game
- Setup Player
- Show Game Statistics

Dependencies:

-Depends on the GUI which acts as the boundary object between the user and the main interface

Interface Description:

setGame() - this sets the game to be loaded by Game Engine

playGame() - this launches the game set by setGame()

setupGame() - this function presents many choices to the user that are implemented in the Table interface such as

- Adding players to the game or table
- Choosing a difficulty level
- Loading a saved game

setPlayer() - this function asks the user to load an existing profile or create a new one

viewStats() - this function allows a player to see his stats

2.2 Observer/GUI Interface

View:

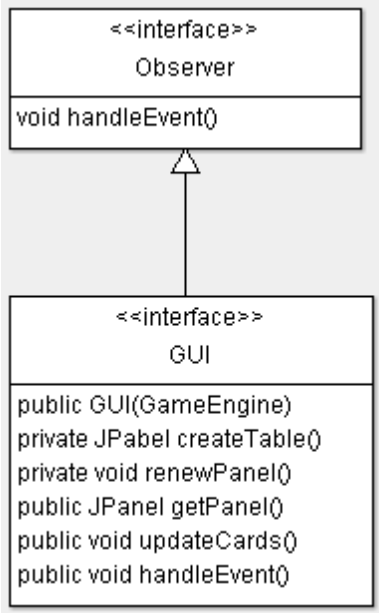

Figure 2 Observer/GUI Interface view

Description: The GUI for the system will implement the Observer interface to allow for updates throughout the game.

Dependencies: GameEngine

Interface description: Specific modifications to the GUI class will be necessary depending on what type of game is being played. This is because for each type of game, the display will likely be different in almost all cases and hence must be modified to properly display certain aspects relevant to the particular game. The primary area necessary for modification will be createTable() while everything else such as renewPanel() or updateCards() will require minimal or no necessary modifications at all.

Observer Interface:

Attributes:

Functions:

Void handleEvent() : Action performed

GUI Interface

Attributes -

private GameEngine ge

private JPanel table

Functions -

public GUI(GameEngine ge)

private JPanel createTable() : reset panel for reinitialization and layout cards on table

private void renewPanel() : updates the panel

public JPanel getPanel()

public void updateCards() : call to update game's card info

public void handleEvent() : will override handleEvent() from the Observer interface to handle events and calls for updates (updateCards(), renewPanel())

2.3 Game Engine & Card Subsystem

View:

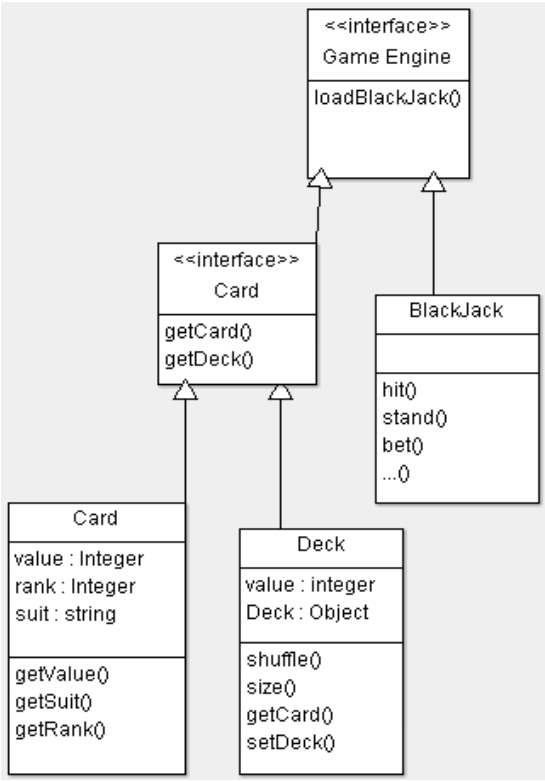

Figure 3 Game Engine and Card Subsystem View

Description: The Game Engine subsystem will include methods to call on the Game class. Each game will have a class and method to call on that class in the Game Engine. For now, we will just have one for the Blackjack game. The Game Engine subsystem will be composed of an interface, the Card interface and Blackjack Class. The Card subsystem will include methods to call on the Card class and Deck class. This model provides flexibility. For example, if we would like to create another game, we would create a method in the interface to call on the game class and create the game's class with all its functions.

Dependencies: For the Game Engine interface to be completed, the Card interface and Game class must be completed.

Interface Description: The GameEngine interface will include all methods used to access the Card class and Blackjack class. The most important methods include:

loadBlackjack():call on the Blackjack class

The Blackjack class will include all methods pertaining to the game of blackjack. The most important methods include:

hit():receive a single card.

stand():leave hand as it is

bet(): player can make a bet on their hand before any other options

The Card interface will include methods to call on the Card class and the Deck class.

The Card class will include final attributes of the card which are: the value of the card, the rank of the card and the suit of the card. The Card class will include all methods that deal with the card. The most important methods include:

getValue: get the value of the card

getSuit: get the suit of the card

getRank: get the rank of the card (this can be different for each game)

The Deck class will include all methods that deal with the card. The most important methods include:

Shuffle(): randomize the deck

setDeck(): put all 52 cards into a single deck

getCard(): get a single card from the deck

size(): return the size of the deck

2.4 Participant Subsystem

View:

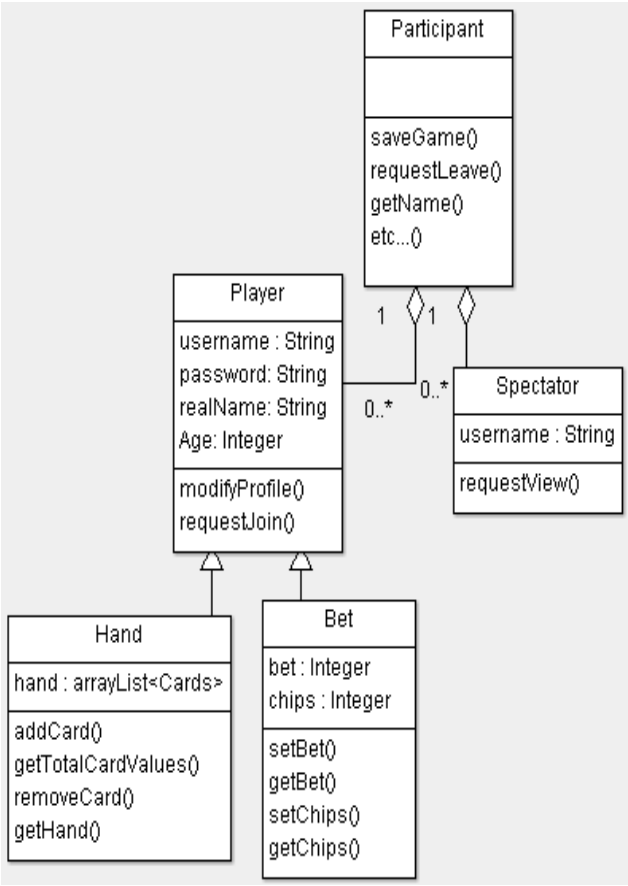

Figure 4 Participant Subsystem View

Description: The Participant subsystem will include the Player and Spectator objects. Each of these objects share common functionalities such as

- Save Game
- Request Leave
- Play Game
- Close Game
- Get Name

The Participant subsystem is connected to the Game and Table subsystems. Also the Participant subsystem uses data from the Storage subsystem to implement functionalities such as save game, get name.

Dependencies: Storage subsystem. The participant subsystem uses Storage to save games and get name and or statistics.

Interface description: The Participant interface will include all methods used by a Player/Spectator object. Basically these methods will be commands for any Player or Spectator. They include:

- saveGame()makes a call to request the Table object that the Player belongs to to get all necessary data to save into the Database (Table object must validate the call request is by a Player who is the current TableOwner)
- requestLeave() makes a call to request the Table object to release the Player/Spectator from belonging to the Table object
- playGame()makes a call to the main interface to begin game play (validate as TableOwner required through Table interface)
- closeGame()makes a call to the Table object that the Player belongs to (must validate as TableOwner required through Table interface) to stop game play and close the Table
- getName()will allow the Table object to query and push necessary data of the Player/Spectator object to the Database

Player function would also include requestJoin() which interfaces with the main interface to assign the Player object a specific Table object to belong to.

Spectator objects may also requestView() which interfaces with the main interface to assign the Spectator object a specific Table object to belong to.

2.5 Table Subsystem

Description: Sets the game up by organizing the table of the game. The table subsystem also adds and removes participants (players and spectators) from the table/game. The TableOwner starts the game, and at any time during the game, he can save it, and close the game.

Dependencies: Table needs the participant subsystem to be implemented in order to complete the Table class.

Interface Description: The most important methods of the Table interface include:

setDifficulty() : not applicable for black jack, but sets the games difficulty

requestJoin() : TableOwner accepts or rejects the player's request to join the table

requestView() : TableOwner accepts or rejects the spectator's request to view the table

requestLeave() : an existing player, spectator, and TableOwner may leave the table

setTableOwner() : TableOwner selects another player as the new TableOwner

playGame() : the TableOwner starts the game

saveGame() : saves the game status

closeGame() : exits game

2.6 Storage Subsystem

View:

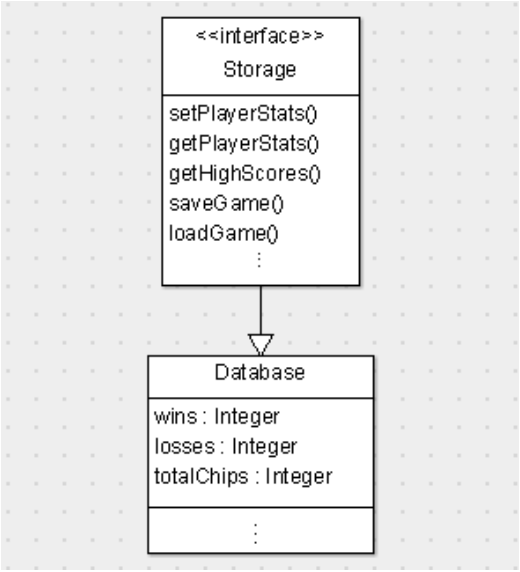

Figure 5 Storage Subsystem View

Description: The Storage subsystem will include all the game's stored data. This includes all the player statistics, along with saved game data. The Storage subsystem will be composed of an interface and a database. This model provides flexibility, since only the interface would need to be modified if changes are made in the database architecture.

Dependencies: None

Interface Description: The Storage interface will include all methods used to access the database. The most important methods include:

setPlayerStats() : saves a Player's statistics into the database

getPlayerStats() : retrieves a Player's statistics from the database

getHighScores() : retrieves the 5 highest scores from the database

saveGame() : store player-specific game information depending on game (blackjack: bet value and hands in play)

loadGame() : retrieve player-specific game information from the database

The database will be implemented as a flat text file. This simple structure will be sufficient to store all the simple information we need, with minimal complexity. This data includes specific player statistics for each game (e.g. Blackjack wins/losses, total chip amount, etc.). When players select to save game, the database will also store saved game progress. For the blackjack game, this information includes the current bet amount and cards in their hand.

3. Analysis

We have chosen to go with a three-tier architecture design. The three components as shown in 'Figure 1 System Architecture Overview' shows the breakdown into the Presentation, Logic, and Database Layer.

We believe breaking down the system into the appropriate subsystems will help to facilitate reusability, flexibility, and maintainability and in the interest of high cohesion and low coupling.

The GUI subsystem will depend on the GameEngine subsystem as different games will want the display to be modified accordingly to suit the specific game play style and hence the GUI will require modifications to the layout with regards to the differences. Perhaps the way in which specific events are handled specific to the game play will need to be modified accordingly for the GUI as well. Otherwise, the updating side of the GUI will not require much modifications if any as it will implement the Observer interface to easily update necessary changes from events. The Main Interface subsystem handles the interactions between the GUI, GameEngine, and Table. The Main Interface will need to what game is chosen, what the game play style is for example the difficulty level, so small tweaks may be necessary to accommodate the provision of game specific features. On the whole it should also not require major reimplementation as the main functions it's supposed to carry out is rather generic to support any game.

The Table subsystem depends on GameEngine and the Participant subsystem. It also implements the Observable interface in order to accommodate requests such as request view, join, and leave. This will make it easy to add and remove observers as necessary and to notify any changes to each of the observers in a systematic manner. It will be responsible for setting up a game and so it will require corresponding modification depending on the game play style of a given game provided by the GameEngine. On the whole it should not require much modification if the game's game play setup is rather generic. Functionalities within the Table subsystem should be pretty standard for example in terms of a TableOwner handling Participant requests to join, view, or leave a given Table object.

The Participant subsystem will interact with the GameEngine and Table Interface as it is necessary for example for a given Participant to trigger events such as requests to the Table and so on. While the Participant interface will not require modification, extensions such as to accommodate specific Player implementations will naturally be rather game dependent. This is because the way in which a given player interacts in a game depends on the actual game play of the particular game – and therefore this subsystem resides as part of the logic layer. The Spectator implementation should not require change as it will ultimately be a view-only object

much like a Player but with much more limited functionalities. Views should be able to present the Table regardless of what game is currently being played.

The purpose of the database layer is to store game data. It does not depend on the logic or presentation layer and hence it is a component on its own in a completely separate layer. Having a simple structure of storing the data into a text file will help decrease complexity and in turn promote desired qualities such as reusability, flexibility, and maintainability. Such a setup can easily accommodate storing data such as name of game, chip amounts, win/loss numbers, lists of participants, etc with no necessary knowledge of how a particular game works.

The GameEngine subsystem will encompass both the Card subsystem and Game subsystem. The Card subsystem will not require changes as Card objects are pretty generic across all types of card games so this subsystem will have high reusability. Depending on how an actual game references different Card objects, this will be handled by the Game subsystem. Therefore a Game subsystem will hold all the logic of the specific game and the GameEngine will be able to take on(load) different Games objects with less modification necessary on the GameEngine subsystem as a whole.

4. Workload Estimation

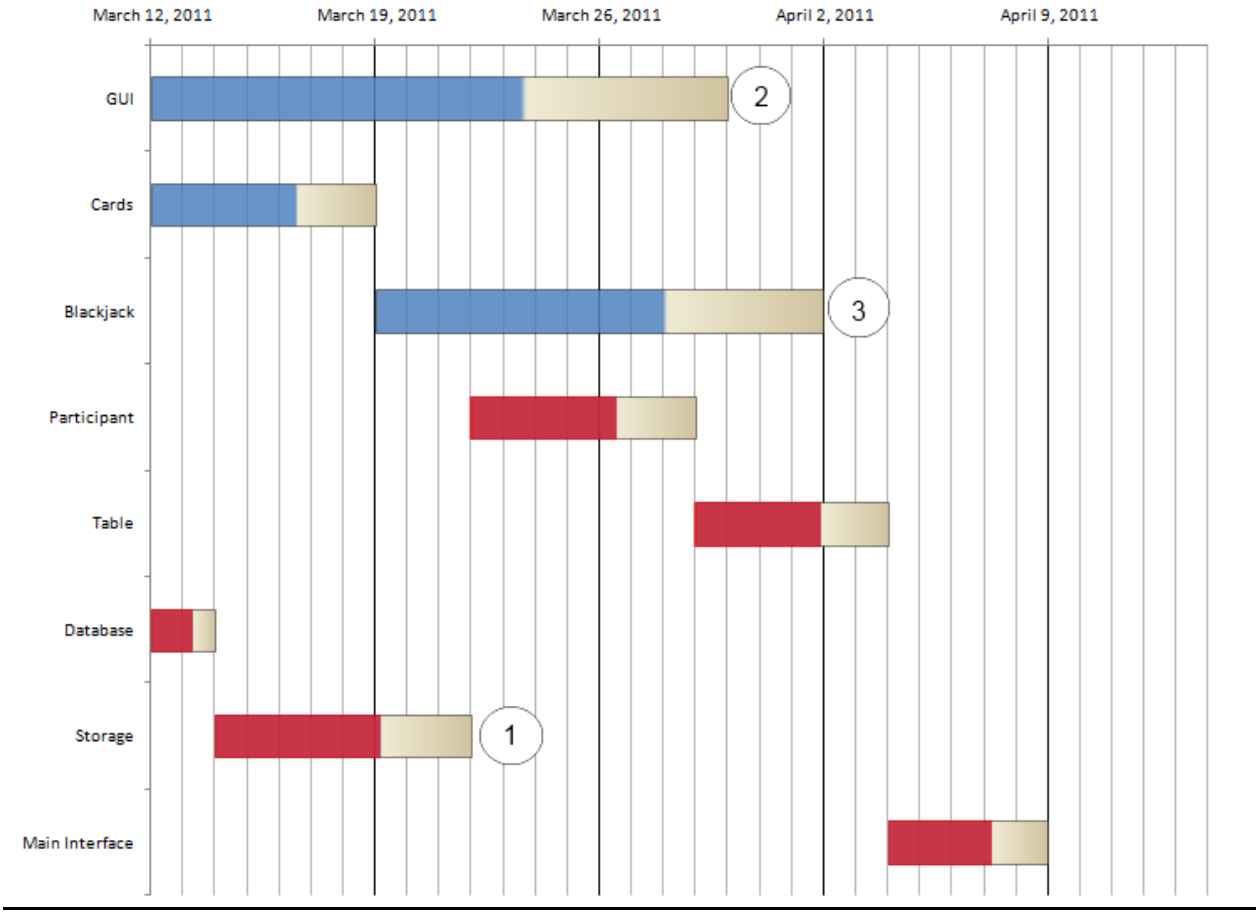

Figure 6 Workload Estimation Gantt Chart

Blue: Development time, Red: Critical Path, Grey: Testing time

Milestones indicated with circled numbers

The workload was broken down into different subsystems, according to their dependencies. Development is projected to begin on March 12th 2011. The final deadline is expected to be met on April 9th, 2011. Days are approximated to roughly 2 hours of work, for a total of 134 work hours. Development for the parallel tasks will be split among the five members.

Until milestone 1:

GUI: 2 members

Cards & Blackjack: 2 members

Database & Storage: 1 member (until completion)

From milestone 1 to 2:

GUI: 1 member (until completion)

Blackjack: 2 members

Participant: 2 members (until completion)

From milestone 2 to 3:

Blackjack: 2 members (until completion)

Table: 3 members (until completion)

After milestone 3:

Table testing & Main interface: 5 members

This breakdown of subsystems will allow most of the project to be done with several tasks in parallel. Important milestones are marked when essential components of our software architecture are completed, and when many team members will transition between tasks. Upon completion of the 3rd milestone, all members will start working together to finalize the product. This includes final testing of the Table subsystem, and implementation of the Main Interface which will link important interfaces together.