Design Document

Team PJA

January - April 2019

Table 1: Team Members

Name	ID Number
Souheil Al-Awar	26558038
David Boivin	40004941
Benson Chan	40046280
Annes Cherid	40038453
Carl Neil Cortes	40016567
Gaoshuo Cui	40085020
Robert Laviolette	27646666
Karim Loulou	40027203
Ke Ma	26701531
Kevin McAllister	40031326

1 Introduction

The primary goal of this project is to develop a computerized version of the card game CodeNames. The game follows the same rule set as the original, with the exclusion of the system for giving clues. Furthermore, the cards and clues were modified from the original to include some simpler word categories.
This design document will outline the details of the Architectural Design, Detailed Design and Dynamic Design Scenarios. This also includes subsections for the individual subsections for each subsystem in the software, as well as a full description of each class.

1.1 Purpose

The purpose of this document is to present the design of the CodeNames game, which is in partial fulfillment of the requirements of COMP 354. It will provide details on the architectural design, subsystem specifications and dynamic design scenarios. The architectural design will describe the software architecture that was chosen for the project, as well as the class diagram. The subsystem specifications will describe each individual subsystem, and the method of communication between them. Finally the dynamic design scenarios will describe important execution scenarios and the interaction between the subsystems during each one.

1.2 Scope

The scope of this document is to provide details on the internal functions of the software, in order to provide a clear description for the current iteration and future additions. The software architecture will be explained in detail to allow future coders to properly understand the code and be able to implement new features in future iterations. The class diagrams, subsystem units and dynamic design scenarios will be created following the UML design.

2 Architectural Design

This section will describe the chosen software architecture, as well as the package diagram designed based on organization of the code. It will also describe the software interfaces between the subsystems, detailing the communication and the passed parameters.

2.1 Software Architecture

The chosen architecture for the CodeNames game is the Model View Controller model (MVC). The MVC architecture is made up of 3 components: the model, the view and the controller, as the name suggests.

The model component is the core of the game, where the game data is stored and manipulated. The model is updated by commands from the controller, and the game logic is processed after each update by another component in the controller.

The view component is the graphical user interface (GUI) in which the user sees and interacts with the program. The view sends messages to the controller by passing messages with information on the chosen input, and is updated after the game processes the selected action.

The control component is the core logic of the game, and is the component that handles and processes all updates to the game. The Controller performs actions based on the messages received from the view, and updates the view based on changes made to the model. The controller also handles errors in the inputs, and prevents crashing from incorrect inputs by the user.

This version of the MVC architecture separates all direct communication between the Model and View, preventing changes from being made without being properly processed by the controller.

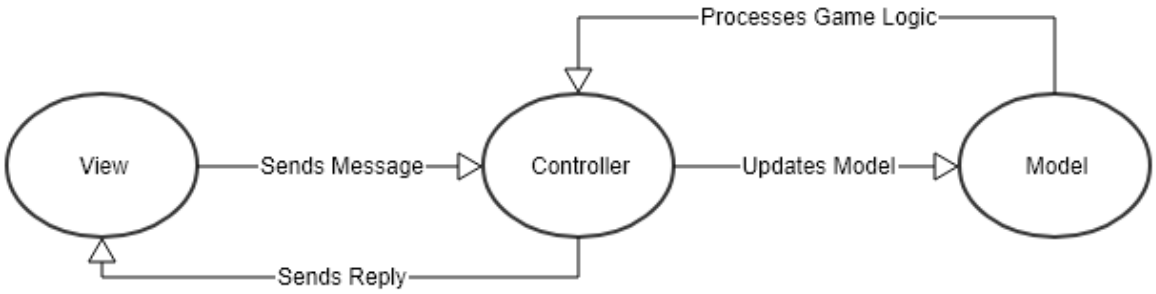


Figure 1: MVC Architecture

2.2 Architectural Diagram

The package diagram outlines the components of each subsystem and well as the relations between the inner units of each package.

View Diagram

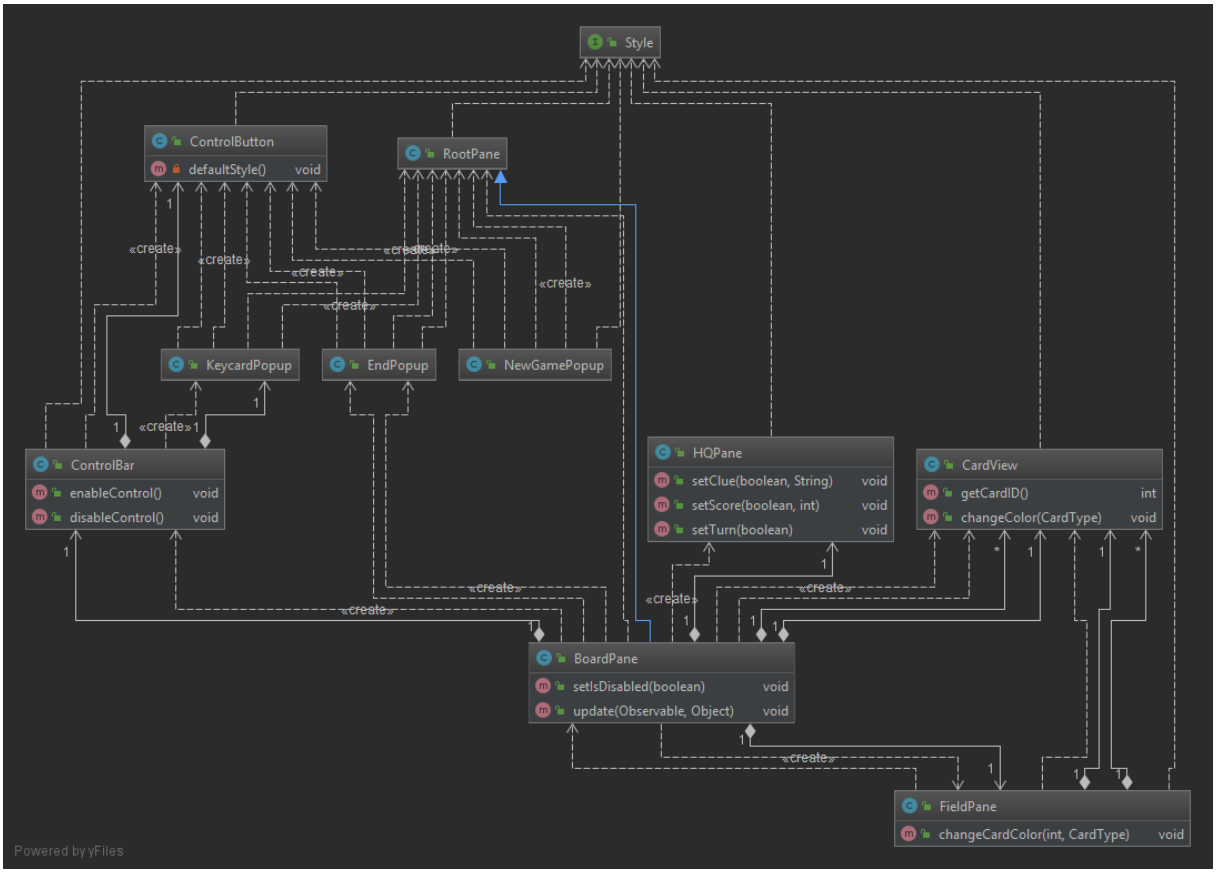


Figure 2: View UML Diagram

The view subsystem contains all the units responsible for displaying the current state of the game as determined by the model, as well as receiving user input to move the game state forward. The view subsystem is the only way the user can interact with the program, restricting the user access to the back end to only the clickable buttons. The view sends messages through the inbox, and receives messages through the outbox, both of which are also used by the controller subsystem.

Control Diagram

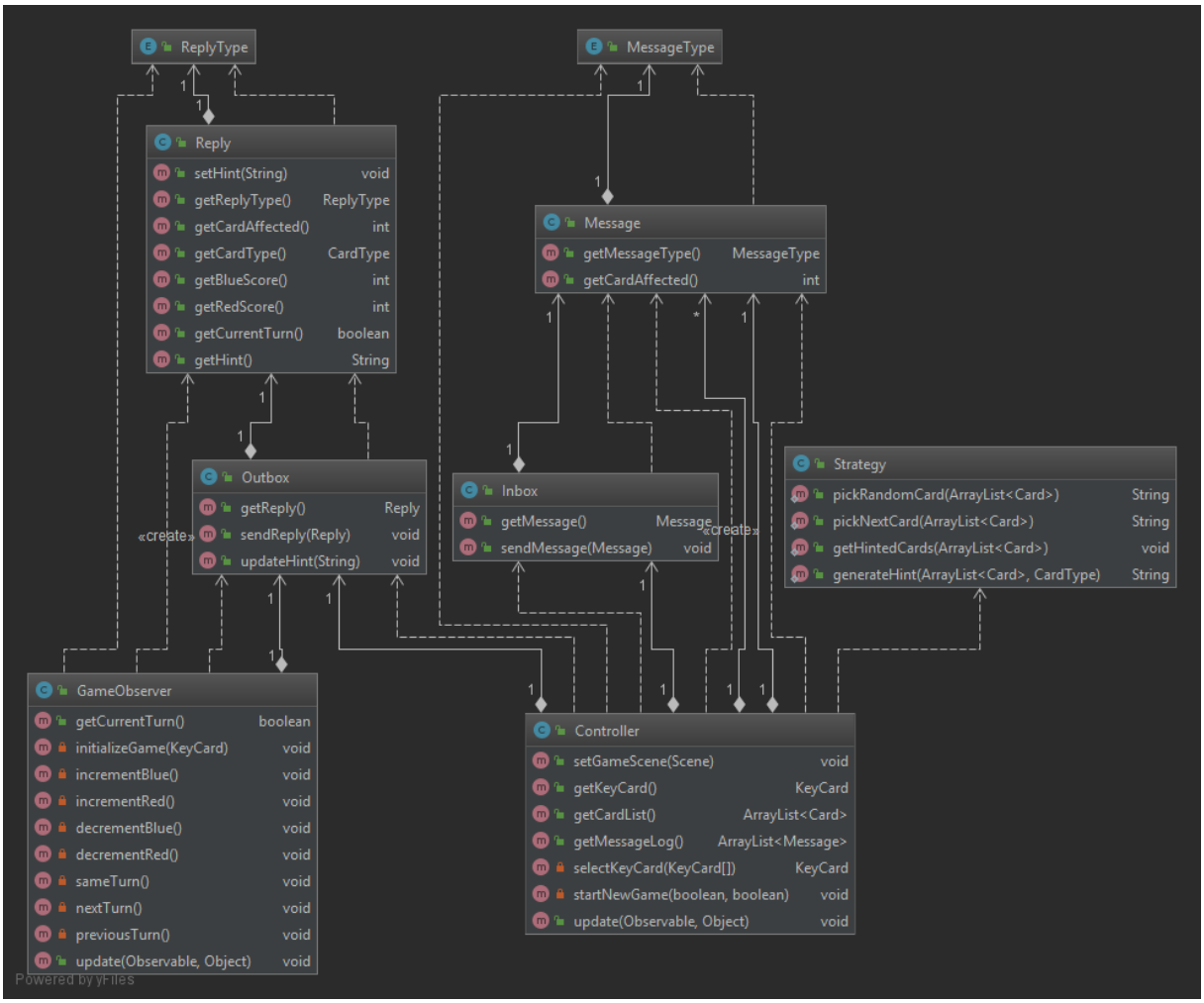


Figure 3: Control UML Diagram

The control subsystem contains all the units responsible for processing and updating the game state, performing changes on the model and notifying the view of the results of updates as determined by the game logic. The control is made up of two major classes: the controller and the game observer, with the former processing user inputs, and the latter handling the game logic to update the state.

Model Diagram

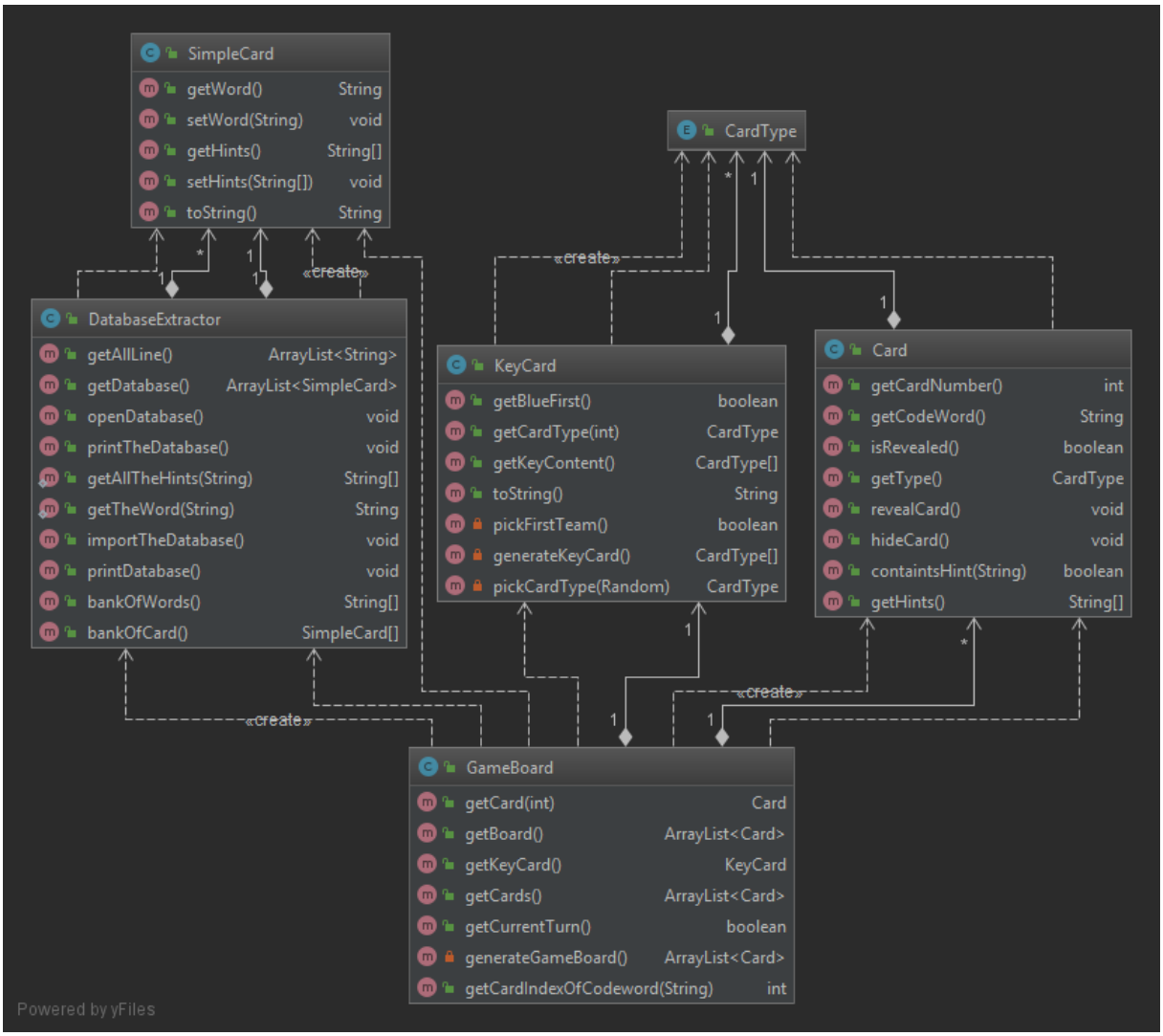


Figure 4: Model UML Diagram

The model subsystem contains all the units responsible for storing the data related to the game, as well as the database of all codewords and clues. The generation of the initial game board and keycard is also handled by the model, with the generation being done during the initial setup and the new game creation command.

2.3 Subsystem Interface Specifications

Specification of the software interfaces between the subsystems, i.e. specific messages (or function calls) that are exchanged by the subsystems. These are also often called “Module Interface Specifications”. Description of the parameters to be passed into these function calls in order to have a service fulfilled, including valid and invalid ranges of values. Each subsystem interface must be presented in a separate subsection.

This section covers the communication between the subsystems, through the use of the Java Observer interface. It also details user interaction interface, and the methods in which the user input is processed by the program.

User Interface System

The user interface is done through the graphical user interface (GUI), and the clickable buttons available on it. The user is able to click buttons for each of the specified actions (Next Move, Undo, Redo, View KeyCard, New Game, Quit), and can manually interact with each individual card to select a specific one. The GUI then sends messages to the Inbox detailing the button clicked, as well as the card affected by the action.

Inbox and Message System

In order to communicate with the controller, messages are sent to the Inbox, which is an observable object, through the use of the sendMessage() function in the Inbox.java class. The sendMessage function takes a Message object, which requires an Enum MessageType and an Integer CardAffected to be specified. When this function is used, the observer is notified of the new Message that is in the Inbox.

Controller and Strategy System

The controller system is an observer object that is attached to the Inbox. This means that whenever the notifyObservers() function is called in the Inbox class, the Controller runs its update() function and processes the passed object, which is a Message object in this case. The controller processes the Message and updates the model according to its contents. The Strategy.java class is used when the passed Message is of type ”NEXT”, in which the chosen strategy (random or sequential) is used to decide which card should be picked. With the implementation of clues, the amount of cards to select from is narrowed down based on the hint, instead of guessing from the entire set.

Card System

The card system is contained in the model package, and contains the information on each individual card that is in play during the current game. Each Card object is observable, and will notify its attached observer when an update is done by the Controller class. Whenever a card is revealed (through "SELECT", "NEXT", "REDO") or hidden (through "UNDO"), the notifyObservers() function is called.

GameObserver System

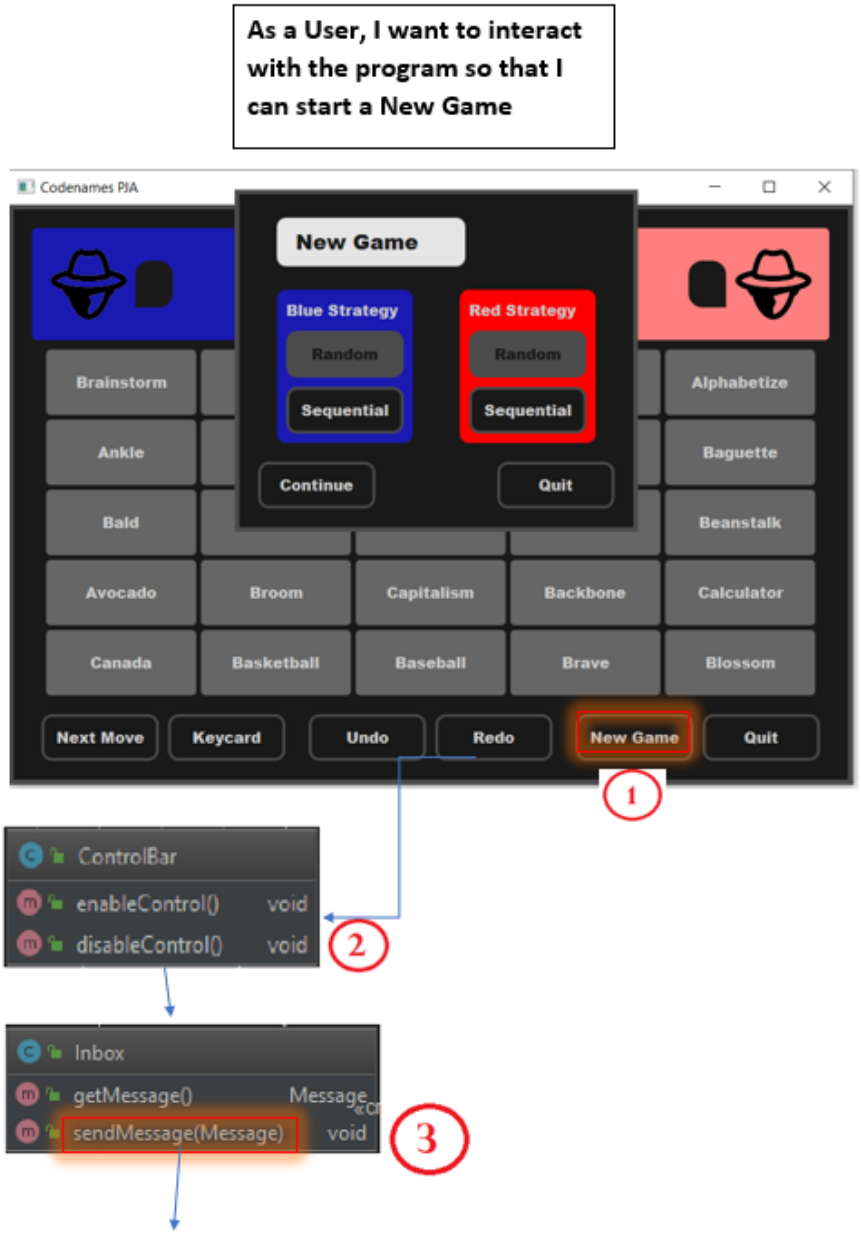
The game observer system is an observer object that is attached to all the active cards in the game state. Whenever the notifyObservers() function is called by a Card, the GameObserver runs its update() function and processed the passed object, which is a Card object in this case. The GameObserver then runs the related game logic based on the affected card and the current turn, updating the scores for each team, determining the next turn, and ending the game if a victor is decided by action of the current turn. After processing the game logic, the GameObserver sends a Reply to the Outbox through the sendReply() function.

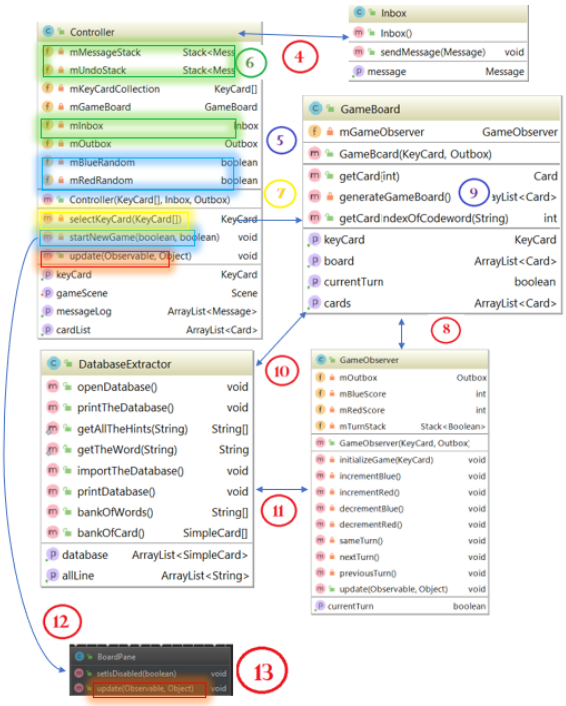
Outbox and Reply System

In order to send information back to the GUI, the GameObserver invokes the sendReply() function in the Outbox.java class. The sendReply function takes a Reply object, which requires an Enum ReplyType, an Integer CardAffected, an Enum CardType, an Integer BlueScore, an Integer RedScore, and a Boolean blueTurn. When this function is used, the observer attached to the Outbox, which is the GUI in this case, is notified of the new Reply.

3 Dynamic Design Scenarios

We have chosen for user-goal level use cases 2 scenarios: The "New Game" and "Next Move" scenarios. We want to explain how a new game is generated and ready to be played as well as the next move to progress on the game. The two user stories are: As a User, I want to interact with the program so that I can start a New Game and As a user I want to click on the next move button so that I can progress on the game.





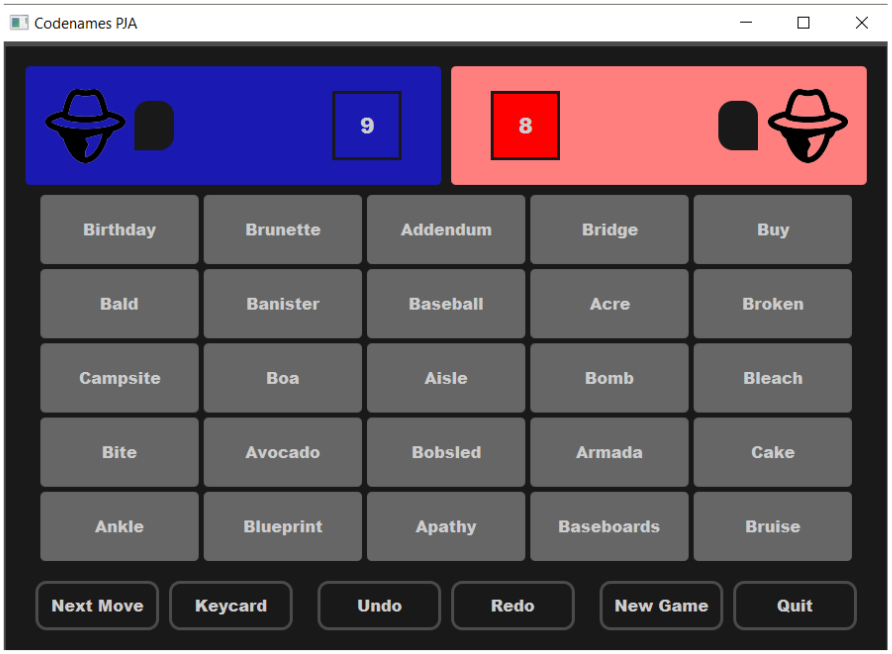


Figure 5: User Story: New Game

Code Sequence for NEWGAME:

1. User Clicks "New Game" button and selects the desired strategy for each team, then clicks "Ok"

2. ControlBar.java sends Message Object to Inbox.java via Inbox.sendMessage(Message m)

3. Inbox.java sets its internal message to the passed Message and notifies the Observer

4. Controller.java is notified of the change in Inbox.java and calls its update() function

5. The update() function runs Controller.startNewGame(Boolean blueRandom, Boolean redRandom) based on the selected strategies)

6. startNewGame() clears the MessageStack, UndoStack and Message Log

7. startNewGame() generates a new GameBoard.java object with a keycard selected from the collection via Controller.selectKeyCard(KeyCard[] collection)

8. GameBoard.java creates a new GameObserver.java object

9. GameBoard.java creates a new board via generateGameBoard() and is returned an ArrayList of 25 random cards

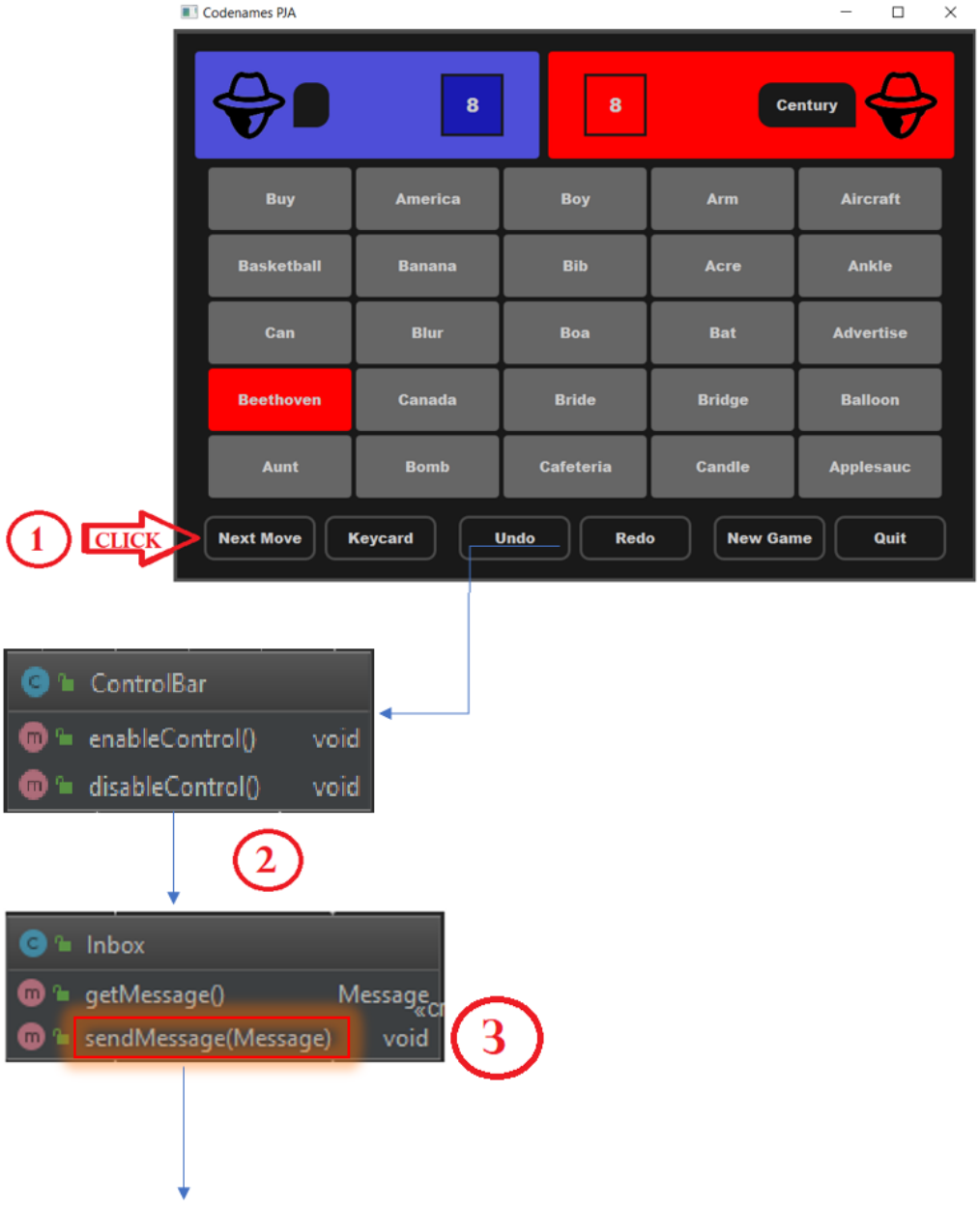
10. generateGameBoard() creates a new DatabaseExtractor.java object and selects 25 codewords at random and generates an ArrayList of 25 Card.java objects

11. generateGameBoard() attaches the GameObserver.java as an Observer for each card.

12. startNewGame() generates a new BoardPane.java object based on the new GameBoard.java object and sets the GameScene to the new BoardPane.java object

13. startNewGame() unbinds the old BoardPane.java object from the Outbox and attaches the new BoardPane.java object to it.

AS A USER I WANT TO CLICK ON THE NEXT MOVE BUTTON SO THAT I CAN PROGESS ON THE GAME.




Controller
setGameScene(Scene) void
getKeyCard() KeyCard
getCardList() ArrayList<Card>
getMessageLog() ArrayList<Message>
update(Observable, Object) void
4
6
Outbox
mHint String
Outbox()
sendReply(Reply) void
updateHint(String) void
reply Reply
Return String hint
5
Strategy
storedHintedCards ArrayList<Card>
storedHint String
NULL_CARD_ERROR String
pickRandomCard(ArrayList<Card>) String
pickNextCard(ArrayList<Card>) String
getHintedCards(ArrayList<Card>) void
generateHint(ArrayList<Card>, CardType) String
7
8
9,10
GameObserver
mOutbox Outbox
mBlueScore int
mRedScore int
mTurnStack Stack<Boolean>
GameObserver(KeyCard, Outbox)
initializeGame(KeyCard) void
incrementBlue() void
incrementRed() void
decrementBlue() void
decrementRed() void
sameTurn() void
nextTurn() void
previousTurn() void
update(Observable, Object) void
currentTurn boolean
12
Card
getCardNumber() int
getCodeWord() String
isRevealed() boolean
getType() CardType
revealCard() void
hideCard() void
11
13
Outbox
getReply() Reply
sendReply(Reply) void
14
Outbox
mHint String
Outbox()
sendReply(Reply) void
updateHint(String) void
reply Reply
15
BoardPane
setIsDisabled(boolean) void
update(Observable, Object) void
16

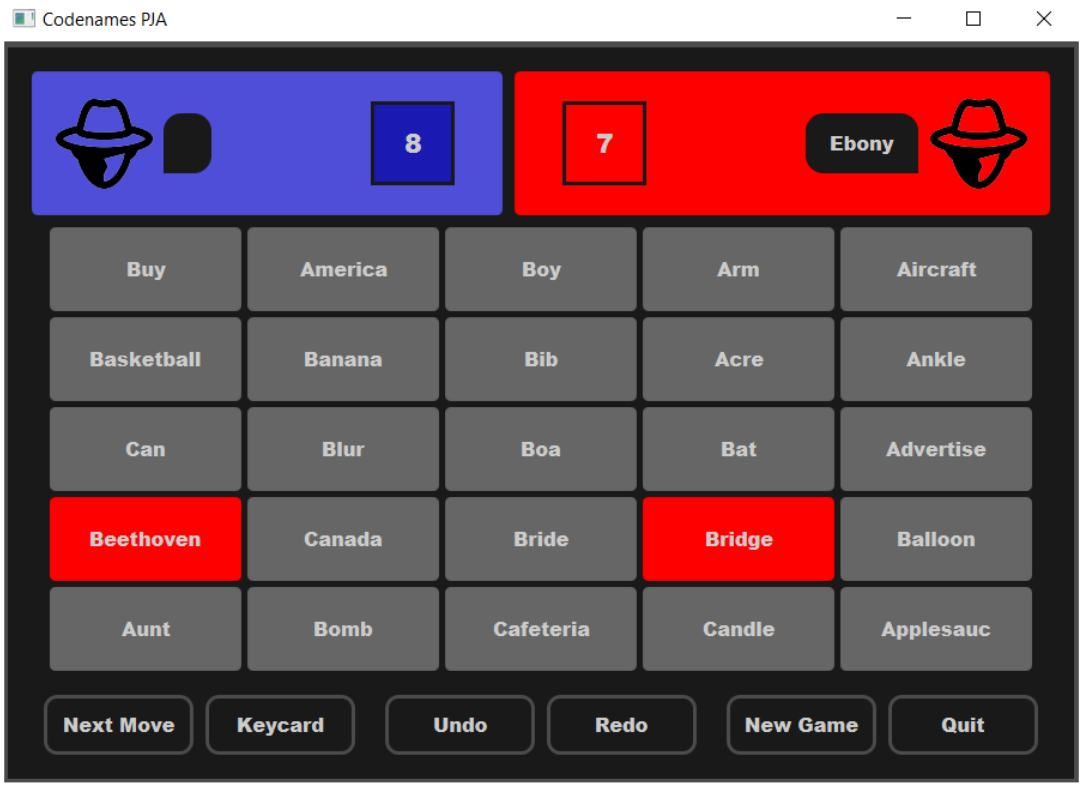


Figure 6: User Story: Next Move

Code Sequence for NEXT:

1. User Clicks "Next Move" button

2. ControlBar.java sends Message Object to Inbox.java via Inbox.sendMessage(Message m)

3. Inbox.java sets its internal message to the passed Message and notifies the Observer

4. Controller.java is notified of the change in Inbox.java and calls its update() function

5. The update() function runs and calls generateHint() in Strategy.java and is returned a String hint

6. Controller.java updates the hint in Outbox.java via Outbox.updateHint(String hint)

7. The update() function then calls Strategy.getHintedCards() and Strategy.java sets its internal ArrayList of Cards to cards related to the hint

8. The update() function then calls either Strategy.pickNextCard() or Strategy.pickRandomCard() depending on the Strategy type set the current team

9. Strategy.java then selects a card from the available list and repeats until an unrevealed card is selected, and returns the card number

10. Controller.java reveals the Card with the returned card number.

11. Card.java notifies the Observer when the card is revealed.

12. GameObserver.java is notified when a Card is revealed and runs its update() function.

13. GameObserver.java updates the game scores and current turns then sends a Reply to Outbox.java via Outbox.sendReply(Reply r)

14. Outbox.java sets its internal reply to the passed Reply, sets the hint in the Reply to its internal hint (set in step 6) and notifies the Observer

15. BoardPane.java is notified when Outbox.java is updated and runs its update() function.

16. BoardPane.java updates the View based on the contents of the Reply, then waits for another user input

A Description of File Format: Tasks

Member Name	**Task**
Annes Cherid	Documenter
David Boivin	Quality Assurance
Karim Loulou	Organizer
Kevin McAllister	Documenter
Souheil Al-Awar	Quality Assurance
Benson Chan	Coder
Carl Neil Cortes	Organizer
Gaoshuo Cui	Coder
Robert Laviolette	Coder
Ke ma	Coder

B Description of File Format: Persons

B.1 Kevin McAllister (40031326)

Date	Task	Duration
Feb. 20	GROUP MEETING	2.0 Hours
Feb. 27	GROUP MEETING - Explained direction and general code implementation for iteration 2	1.5 Hours
Mar. 5	GROUP MEETING - Explained code implementation to coders and divided tasks between them	1.75 Hours
Mar. 12	Completed UML Diagrams and first two sections of the Design Document	2 Hours
Mar. 13	GROUP MEETING - Consolidated all code parts for different members, bugfixed code and clarified parts of the Design Document	0.75 Hours
Mar. 17	Finished Design Document with Annes' additions and logs from all team members	5 Hours
	Total	13 Hours

B.2 Benson Chan (40046280)

Date	Task	Duration
Mar. 5	Attended coders group meeting	1 hours
Mar. 6	Full group meeting worked on iteration 2 code	3 hours
Mar. 9	Looked around the code to understand it	2 hours
Mar. 11	Received code from Gaoshuo, Ke and Robert put it all together for march 13 meeting	1.5 hours
March 13	Attended full group meeting, finalized the project	2 hours
	Total	9.5 Hours

B.3 Annes Cherid (40038453)

Date	Task	Duration
Feb. 27	Group Meeting	1.75 Hours
Mar. 5	Group Meeting	1.5 Hours
Mar. 13	Group Meeting	0.75 Hours
Mar. 3, 6, 14, 16, 17	Individual Work	11 Hours
	Total	15Hours

B.4 David Boivin (40004941)

Date	Task	Duration
Mar. 17	Wrote tests for iteration 2	10.5 Hours
	Total	10.5 Hours

B.5 Ke Ma (26701531)

Date	Task	Duration
Mar. 5	Coders Group Meeting	1 Hours
Mar. 06	Full group meeting worked on iteration 2 code	3 Hours
Mar. 10	Fixed some bugs on algorithm	4 hours
Mar. 13	Attended full group meeting, finalized the project	2 hours
	Total	10 Hours

B.6 Souheil Al-awar (26558038)

Date	Task	Duration
Feb. 27	LAB - Got the mechanics explained to me, got introduced to jUnit and testing which is my role.	1.5 Hours
Mar. 5	MEETING - Heard more of the mechanics of the game and set myself some homework.	2 Hours
Mar. 6	Watched some youtube videos about jUnit and testing.	2 Hours
Mar. 6	LAB - Looked at last iterations code.	1.5 Hours
Mar. 9	Did some testing and wrote some notes.	1.5 Hours
Mar. 13	LAB - Revised my notes, got some questions answered and did some more local testing.	3.5 Hours
	Total	12 Hours

B.7 Gaoshuo Cui (40085020)

Date	Task	Duration
Feb. 20	GROUP MEETING	1.5 Hours
Feb. 22	Familiar with the Previous code	2.5 hours
Feb. 25	Make sure about main objective for Iteration 2	1.5 hours
Mar. 05	Group Meeting	2 hours
Mar. 06	Programming and Group Meeting	3.5 hours
Mar. 07	Programming about my part	2 hours
Mar.12	Debug	1 hours
Mar.13	Group Meeting	1 hours
	Total	15 Hours

B.8 Carl Neil Cortes (40016567)

Date	Task	Duration
Feb. 27	Attended Group meeting and took notes of the meeting	2.5 hours
Mar. 4	Uploaded doodle sheet to slack to decide on next meeting	0.25 hours
Mar. 5	Attended Group meeting and took notes of the meeting	2 hours
Mar. 6	Attended Group meeting and took notes of the meeting	1.5 hours
Mar. 13	Attended Group meeting and took notes of the meeting	1 hour
Mar 17	Finished diary and completed group diary	3 hours
	Total	10.25 Hours

B.9 Robert Laviolette (27646666)

Date	Task	Duration
Mar. 5	Group Meeting	2 Hours
Mar. 6	Programming Implementation	2.5 Hours
Mar. 7	Group Meeting	2.5 Hours
Mar. 13	Group Meeting	2 Hours
Mar. 17	Document Work	1.5 Hours
	Total	10.5 Hours

B.10 Karim Loulou (40027203)

Date	Task	Duration
Feb. 20	GROUP MEETING - Group Meeting	2 Hours
Feb. 21	Organizing iteration 2	2 Hours
Feb. 23	Organizing a meeting	1 Hour
Feb. 27	Group Meeting	1.5 Hours
Mar. 5	Work on web scraper	3 Hours
Mar. 5	Group Meeting	2 Hours
Mar. 13	Group Meeting	2 Hours
Mar. 14	Teach JUnit to Souheil	0.75 Hours
Mar. 16	Created new word database with hints	5 Hours
	Total	24.25 Hours