Analysis and Design Report

Project Title: River Crossing Game

(with customizable puzzle and solver)

Table of Contents

1. Program Design Constraints

1.1. Hardware Architecture

Since the program is developed under the personal computer (PC) hardware architecture, it cannot be run on portable devices like mobile phones or tablets. Moreover, the PC must be installed with Java runtime environment v.11 or higher in version.

1.2. Internet / Cloud Resources

The programme is designed to be run on a standalone PC. Therefore, any features related to cloud services are prohibited. It limited the use of those public libraries to enhance the program's functions. Only those libraries that can be downloaded as packages and stored on the PC can be applied in the project development.

1.3. User interface

A graphic user interface is not included in the program due to constraints of time schedule and human resources. The user interface is only designed to display in the command line interface (CLI).

1.4. Data storage

As a standalone program, the software architecture retains to be simple. The information of different puzzles (such as rules, and characters) is not using a database structure for storage. Instead, a simple JSON file is applied to store the information of a puzzle. This will be further described in the following section.

1.5. Input file (JSON) format

```
{
    "PuzzleName": "Farmer-tiger-sheep-grass",
    "Description": "Sheep eat grass, Tiger eat sheep if farmer not around. Only farmer can drive the boat",
    "Roles": ["farmer", "tiger", "sheep", "grass"],
    "InitialState": [
        ["farmer", "tiger", "sheep", "grass"],
        []
    ],
    "TargetState": [
        [],
        ["farmer", "tiger", "sheep","grass"]
    ],
    "Travelable": ["farmer"],

    "Rules": [
        {
            "RuleType":"Conflict",
            "Roles":["tiger",["sheep"]],
            "Message": "Tiger ate sheep",
            "Exception": {"RuleType": "Coexist", "Roles":["tiger",["farmer"]], "Exception":{}}
        },
        {
            "RuleType":"Conflict",
            "Roles":["sheep",["grass"]],
            "Message": "Sheep ate grass",
            "Exception": {"RuleType": "Coexist", "Roles":["sheep",["farmer"]], "Exception":{}}
        }
    ]
}
```

Figure 1.1 Example JSON file

The puzzle information is stored as an external JSON file, which can be modified by the user and then import as a customized puzzle.

The attributes meaning are as follows:

Attributes	Meaning
PuzzleName	The puzzle's name that to be shown in the game.
Description	The text description shown in the game, this part should contain the game information player needs to know, such as the rules and win condition.
Roles	Define all the characters/items in this game.
InitialState	Define the initial state of the game, the game will put the roles to lands according to this attribute at the beginning of the game.

TargetState	The win condition of the game. When the current state of the game meets the TargetState, the game is considered winning.
Travelable	Define what roles are travelable. To move a boat, at least one of the roles in the boat needs to be travelable.

Rules	Store a list of Rule objects resembling the program.
	
	`public abstract class Rule {`
	`  ArrayList<String> groupA;`
	`  ArrayList<String> groupB;`
	`  Rule exception;`
	`  String msg = "";`
	`}`
	
	The program will verify all the rules on each player's move. The movement is considered invalid if at least one rule is not satisfied.
	
	For the detail of the rule object, check the next section.

Rule:

The Rule object has the following attributes.

```
{
    "RuleType":"Conflict",
    "Roles":["tiger",["sheep"]],
    "Message": "Tiger ate sheep",
    "Exception": {"RuleType": "Coexist", "Roles":["sheep",["farmer"]], "Exception":{}}
},
```

Attributes	Meaning
RuleType	A string that represents the type of rule. There are currently two Rule types, "Conflict" and "Coexist". Conflict rule: If Role A is together with anyone in Role Group B. Then the rule is considered violated. Coexist rule: If Role A is NOT together with ALL roles in Role Group B. Then the rule is considered violated.

Roles	An array that represents the roles is involved with this rule. The format of this field is supposed to be depending on the RuleType. Typically, for the two existing rules, the first element will be Role A, and the second element will be Role Group B.
Message	The message to output if the rule is violated.
Exception	Also a Rule object. The exception rule is used when there are multiple conditions to check. A rule is only considered satisfied if its exception rule is NOT satisfied. For instance, the following statement "The Tiger will eat the Sheep if the Farmer is not together with the Sheep." can be represented by a Conflict rule between Tiger and sheep, and an exception rule which is Coexist rule between the farmer and sheep.

2. Use Case Diagram

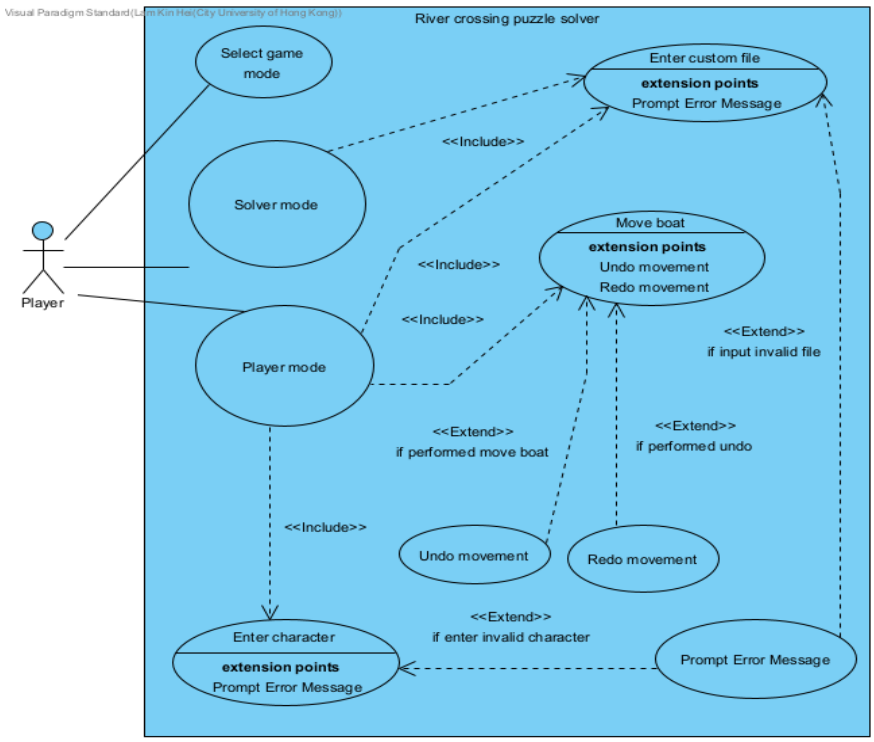


Figure 2.1 Use case diagram of the system

The use case diagram contains one actor, which is the player. Players can select game mode with either player mode or solver mode. If players select player mode, they can play with the default character and control them. In solver mode, the player needs to enter a custom file to run the solver. The system will prompt an error message if the user input is invalid.

2.1. Use Case Specification

2.1.1. Select game mode

Use Case Name:	Select game mode	
Actor(s):	Player	
Description:	This use case describes the process of a player selecting a game mode in the puzzle program	
Reference ID:	PZ-01	
Typical Course of Events:	**Actor Action** Step 1: Initial the selection process. Step 3a: Select the default game mode.	**System Response** Step 2: Show the available game mode on the screen. Step 4a: Enter default gaming mode
Alternate Courses:	Step 3b: Select auto solver mode. Step 4b: Enter solver mode	
Precondition:	The user start the program successfully.	
Postcondition:	The system enters the correct mode.	

2.1.2. Gaming mode

Use Case Name:	Player mode	
Actor(s):	Player	
Description:	This use case describes the process of running a gaming mode.	
Reference ID:	PZ-02	
Typical Course of Events:	**Actor Action** Step 1: Initial the game mode. Step 3: Enter character name Step 4: Enter move boat command	**System Response** Step 2: Ask for user input Step 5: Update the gaming interface. Step 6a: Output the winning message.
Alternate Courses:	Step 6b: the program will output a loss message if input incorrect character	
Precondition:	The user has chosen the gaming mode in the select game mode page.	
Postcondition:	The game return a win or a loose message.	

2.1.3. Enter character

Use Case Name:	Enter character	
Actor(s):	Player	
Description:	The use case diagram describe the process of use enter character	
Reference ID:	PZ-03	
Typical Course of Events:	**Actor Action** Step 1: Initial the game process. Step 3: input the character name to perform a movement.	**System Response** Step 2: Ask the user to input data about the movement. Step 4a: Update the river crossing game interface.
Alternate Courses:	Step 4b: If the input is invalid, an invalid message will be displayed.	
Precondition:	The user selects gaming mode.	
Postcondition:	character name appear on the boat diagram	

2.1.4. Move boat

Use Case Name:	Move boat	
Actor(s):	Player	
Description:	This use case describes the process of a player playing the game by moving the boat in the puzzle program.	
Reference ID:	PZ-04	
Typical Course of Events:	**Actor Action** Step 1: Initial the game process. Step 4: input the character name to perform a movement.	**System Response** Step 2: Display the current river crossing interface and character. Step 3: Ask the user to input information about the movement. Step 5: Display the new status of the river crossing game.
Alternate Courses:	null	
Precondition:	The user has chosen to play in default player mode.	
Postcondition:	River crossing puzzle status is displayed correctly.	

2.1.5. Undo movement

Use Case Name:	Undo movement	
Actor(s):	Player	
Description:	This use case describes the process after the user enters a character name.	
Reference ID:	PZ-05	
Typical Course of Events:	**Actor Action** Step 1: User types a undo command	**System Response** Step 2a: undo the move boat action. Step 3: display updated river crossing interface
Alternate Courses:	Step 2b: if there is nothing to undo, will output a message about 'Sorry, nothing to undo'	
Precondition:	User has performed move boat.	
Postcondition:	river crossing interface is displayed correctly.	

2.1.6. Redo movement

Use Case Name:	Redo movement	
Actor(s):	Player	
Description:	This use case describes a process for redoing a move boat action.	
Reference ID:	PZ-06	
Typical Course of Events:	**Actor Action** Step 1: User sends a redo command to the system	**System Response** Step 2a: the system redo the move boat action Step 3: display updated river crossing interface
Alternate Courses:	Step 2b: If there is no possible redo will display an error message.	
Precondition:	The user just performs the undo move boat function.	
Postcondition:	River crossing interface display correctly.	

2.1.7. Solver

Use Case Name:	Solver mode	
Actor(s):	Player	
Description:	This use case describes how the system responds after getting the custom file.	
Reference ID:	PZ-07	
Typical Course of Events:	**Actor Action**	**System Response** Step 1: Program receives a custom file. Step 2. The system checks the rule and format Step 3: Compute the answer for the river crossing game Step 4a: Display the optimal step for the solution.
Alternate Courses:	Step 4b: Display no solution if the puzzle is not able to solve.	
Precondition:	The user input a custom file.	
Postcondition:	Step output correctly for the user to reference.	

2.1.8. Enter custom file

Use Case Name:	Enter custom file	
Actor(s):	Player	
Description:	This use case describes the process for the user to submit a custom file.	
Reference ID:	PZ-08	
Typical Course of Events:	**Actor Action** Step 2: input the name of the file.	**System Response** Step 1: Show message asking for user input Step 3a: the system gets the file and processes it to the solver function
Alternate Courses:	Step 3b: Ask the user to reinput if the file does not exist Step 4: Process to solver function	
Precondition:	User select solver mode in the selecting page	
Postcondition:	File process to solver function.	

2.1.9. Prompt Error Message

Use Case Name:	Prompt Error Message	
Actor(s):	Player	
Description:	This use case describes the error message after the user inputs invalid information	
Reference ID:	PZ-09	
Typical Course of Events:	**Actor Action** Step 1: User input to the system	**System Response** Step 2: System check input value and type Step 3: Display invalid message and ask user input again
Alternate Courses:	null	
Precondition:	User is running the move boat function or solver function	
Postcondition:	Error message successful delivery to the user.	

3. Class Diagram

The system is divided into 4 main modules: Puzzle, Logic, Solver and UI.

3.1. Puzzle

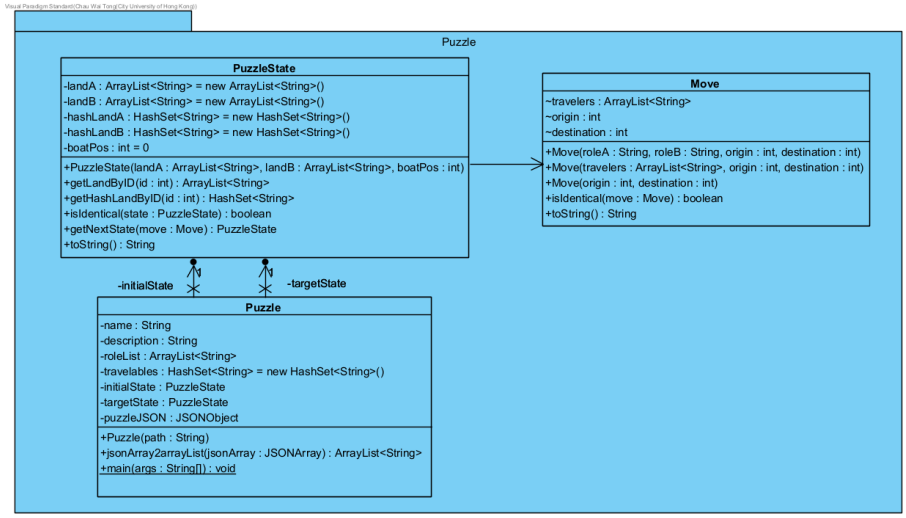


Figure 3.1 Class diagram of the Puzzle Module

The module is for storing the puzzle and in-game information.

Puzzle is for storing information such as roleList, InitialState, and TargetState. This class is also responsible to parse the puzzle JSON files.

PuzzleState is for storing a specific state of the puzzle, which includes the roles in Land A, Land B, and boat position.

Move is for storing a player's move, which includes information like which roles are travelling, origin position and target position.

3.2. Logic module

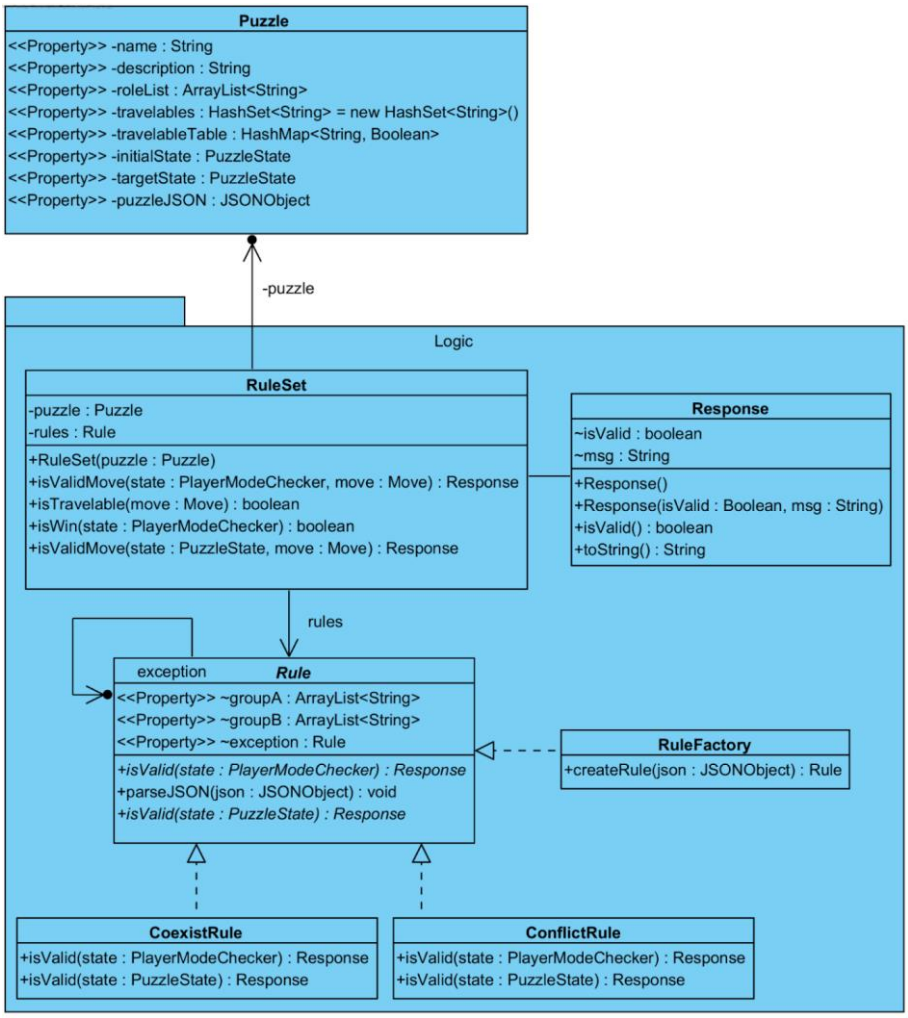


Figure 3.2 Class diagram of the Logic Module

The logic module contains three main classes which are the RuleSet, Rule and Response. Whereas the Rule, RuleFactory, CoexisRule and ConflictRule are written in the format of the factory pattern for fulfilling the open-close principle. The Factory pattern will be further elaborated in the next section - Design Pattern and Principles.

The RuleSet class handle most of the core functions in the Logic module. It mainly handles two objects: the Puzzle and the Rule. Both objects are initialised during the startup of the UI module.

This module returns an object Response to the caller - UI module. The Response object consists of two attributes: msg: string and isVaild: boolean. Those attributes are the answer given to the caller after the computation according to each game rules setting using CoexisRule and ConflictRule.

3.3. Solver module

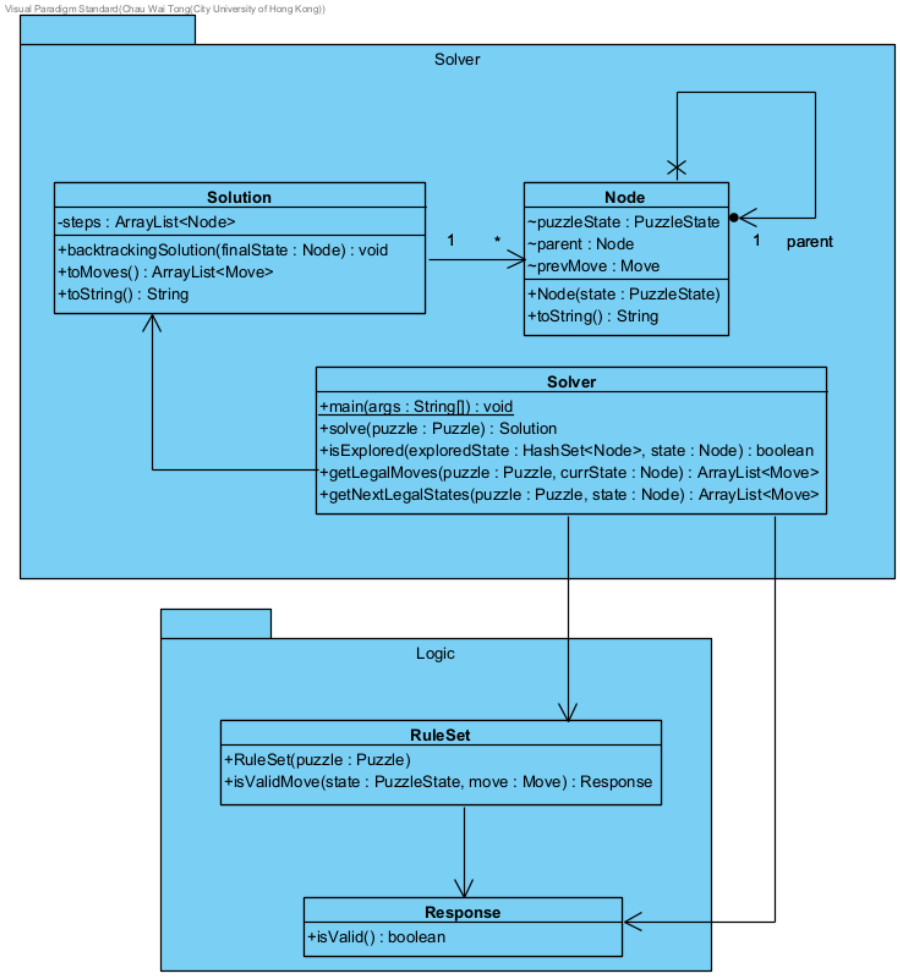


Figure 3.3 Class diagram of the Solver Module

The Solver class is for running the algorithm and outputting the Solution. Other modules invoke this class by creating a Solver object and calling the solve(Puzzle puzzle) method.

The Solution class holds the steps for solving the given puzzle.

The Node class is for storing information while running the algorithm.

3.4. UI module

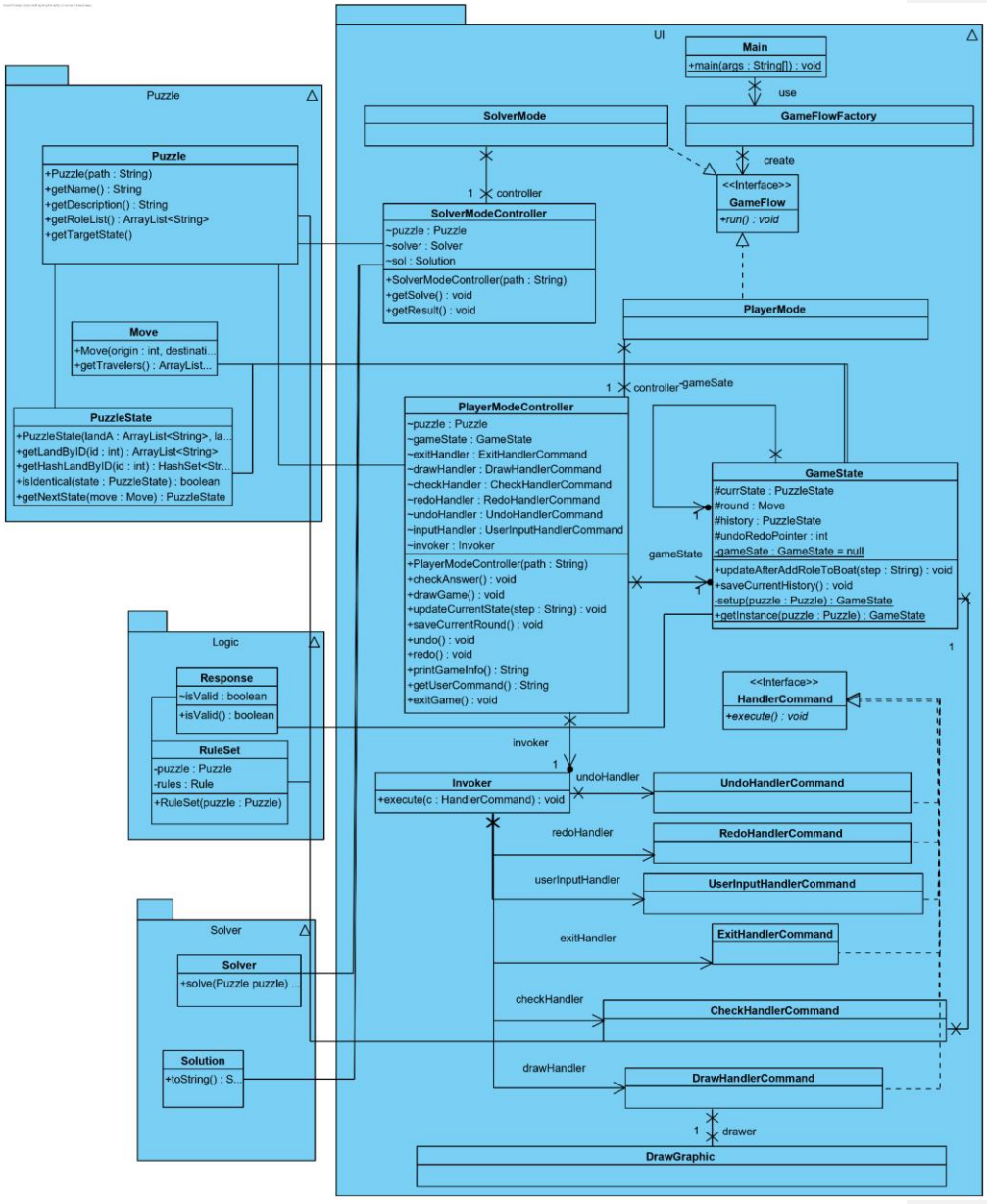


Figure 3.4 Class diagram of the Puzzle UI Module

The UI module will be responsible to interact with the user and other modules.

First, the UI module has a connection with the Puzzle module for storing the game information, because after the user selected player mode or solver mode its loads all the JSON file data into the puzzle class. Next, the UI model also have a connection with the logic model, because the logic model is used to check whether a move is valid in the player mode. The last connection with the UI module is the solver module. It is used to get the puzzle game solution in solver mode.

4. Solver Algorithm

The program will apply breath-first search (BFS) to search for TargetState (defined in the Puzzle JSON file), then backtrack the solution when the TargetState is reached.

The algorithms will utilize the following information:

Let ***PuzzleState*** S be a Class storing the current state of lands, the position of the boat and the previous state. For instance, a PuzzleState S will hold the following information: LandA = {"tiger", "grass"}, LandB = {"farmer", "sheep"}, Boat position = LandA, Previous state = V which also a PuzzleState.

Let ***PendingState*** P, which is a Queue storing valid and unexplored states.

Let ***ExploredState*** E, which is a HashSet storing a set of explored states.

At the beginning, the InitialState (defined in the Puzzle JSON file) will be added to PendingState.

The algorithm will continuously select the first state in PendingState, and explore it (find all valid next states), add those valid states to PendingState, so on and so forth.

To avoid infinite looping, the selected PuzzleStates will be added to ExploredState. The PuzzleStates in ExploredState will be ignored in the later search.

The algorithm will end at one of the following condition:

1. The current selected PuzzleState equals to the TargetState, then start backtracking the solution.
2. No states in PendingState, meaning all valid states are explored, then return no solution.

5. Sequence diagrams

5.1. High level

5.1.1. Select game mode

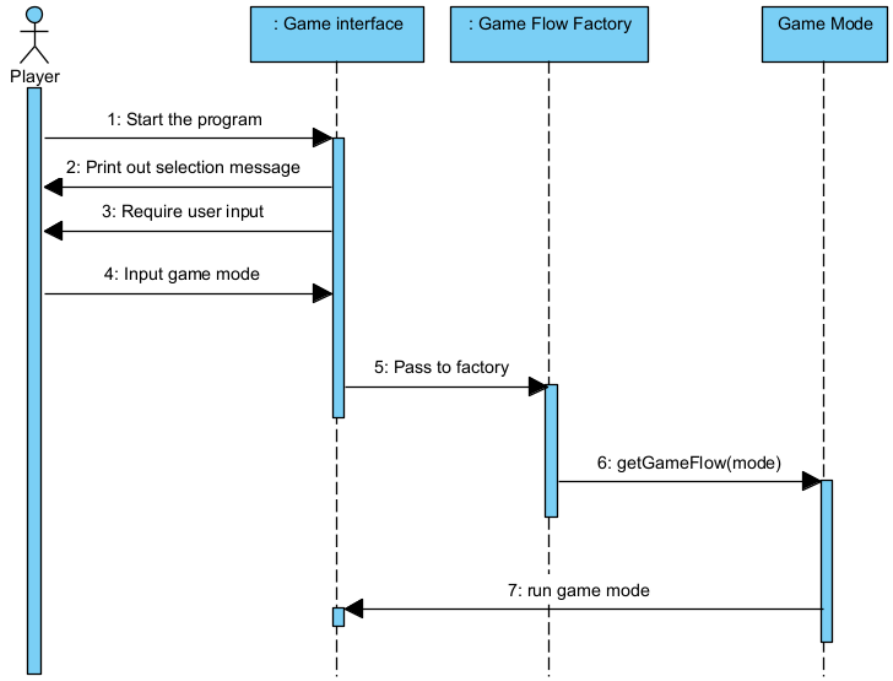


Figure 5.1 Sequence diagram for select game mode

The sequence diagram selects game mode to show the interaction between the user and system. First, the user starts the program, and the game interface will print out a game mode selection message and ask the user to input it after the user inputs the game mode. It will pass it to the game flow factory and get the suitable game mode. And the program will run.

5.1.2. Player mode

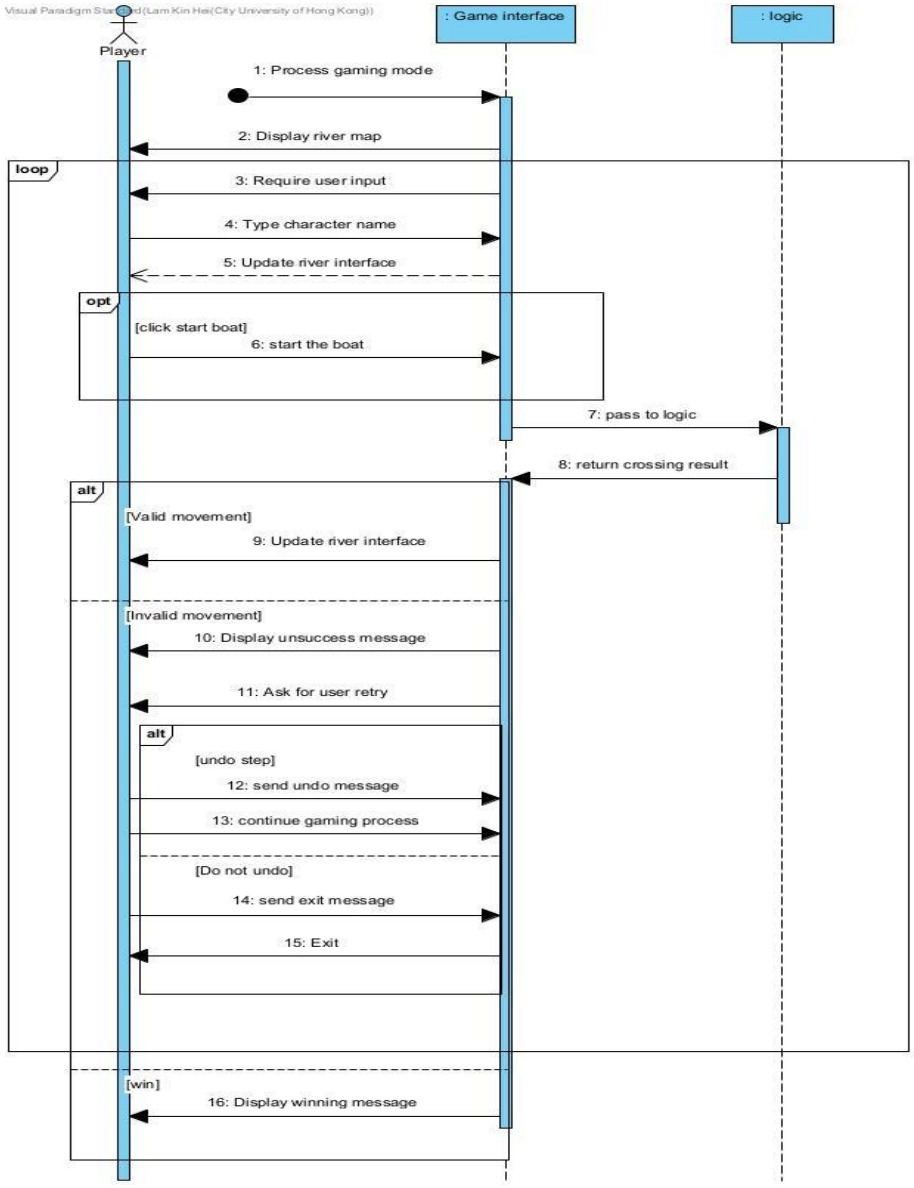


Figure 5.2 Sequence for player mode

The sequence diagram, player mode, is to show the interaction between the user, gaming interface and logic. After the user selects the gaming mode, the gaming mode

is processed. The program will display the river map and ask for user input. After the user inputs the character name, the river interface will update and show a character on the boat. The user can click start boat to perform a movement, the movement will pass to the logic to evaluate and return a message to the game interface. If it is a valid movement, the river interface will be updated. If it is an invalid movement, the system will display an unsuccess message and ask for the user to retry. If the user is willing to retry and send undo message, the gaming process will continue. Else if the user does not select undo function, it will classify as a loss of gameplay and exit the program. Else if the logic module returns a winning message, the program will display the winning message, and the whole gaming mode finished.

5.1.3. Solver mode

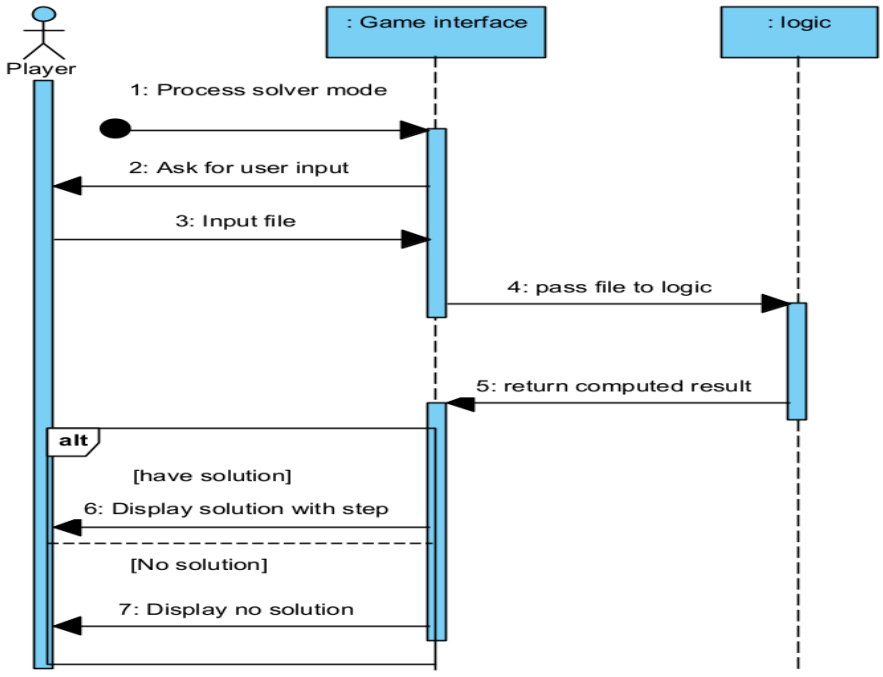


Figure 5.3 Sequence for solver mode

The sequence diagram solver mode is to show the interaction between the user, the gaming interface and the logic. After the user selects solver mode, it will process to the interface. Then the system will ask for the user to input. The user can input a format file to the program to perform computation. In the logic part, it will use different algorithms to compute the solution. If there is a solution and able to perform by the program, it will display the solution with detailed steps. If there is no solution, the program will display no solution on the user screen.

5.2. Sequence diagrams Puzzle solver module

5.2.1. Solver.solve()

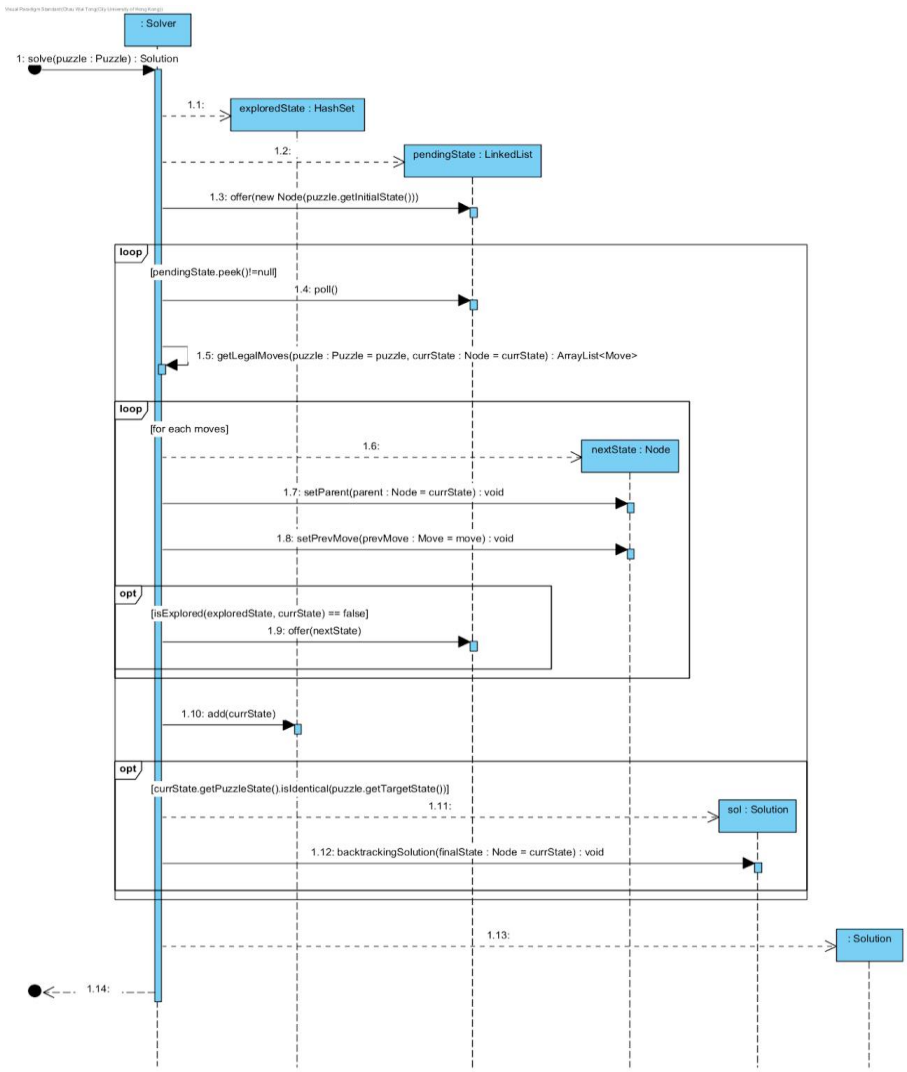


Figure 5.4 Sequence diagram for solver.solve()

Solver.solve(Puzzle) will run the algorithm for finding the solution to the given puzzle. The method will keep exploring the next unexplored valid states for the current state of the game, and add them to pendingState. Then select the state with the lowest depth in pendingState , explore the state, and add it to exploredState. The process continues until the target state is reached.

5.3. Sequence diagrams of Puzzle UI module

5.3.1. UI.SolverMode

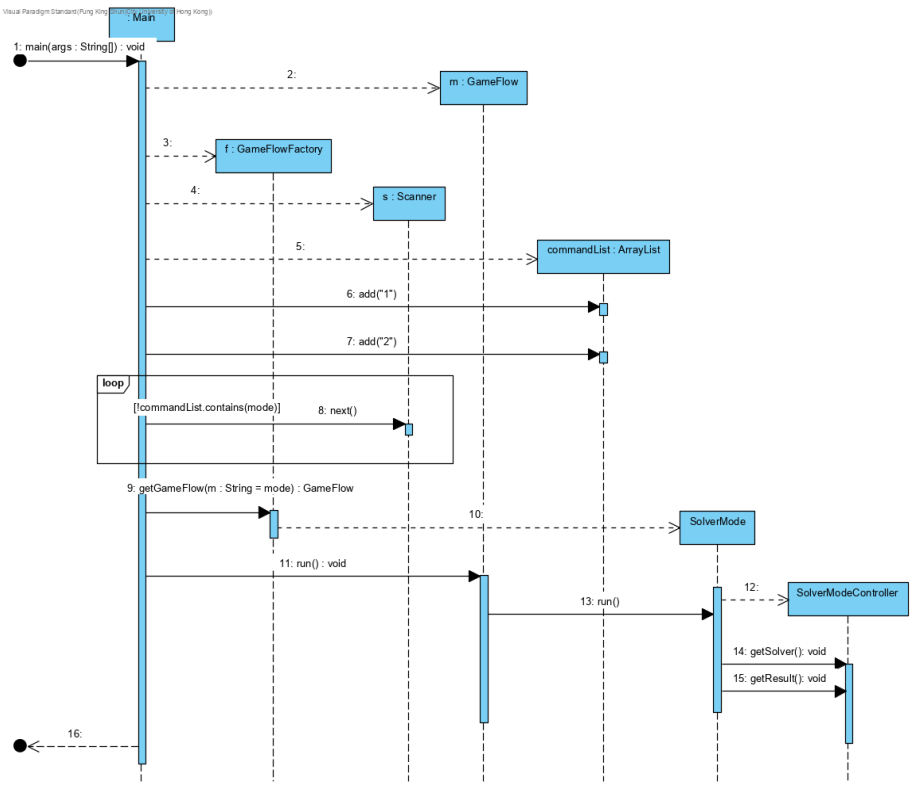


Figure 5.5 Sequence diagram for UI.solverMode

This sequence diagram shows how to control the user input and operate in solver mode. First, the process will start with main and create the associated game factory object. Next, if the user selects the game mode number as 1, it will enter to solver model to create a SolverModelController. Finally, it runs the getSolver() and getResult() method, which part will be passed to the Solver.solve sequence diagram.

5.3.2. UI.PlayerMode

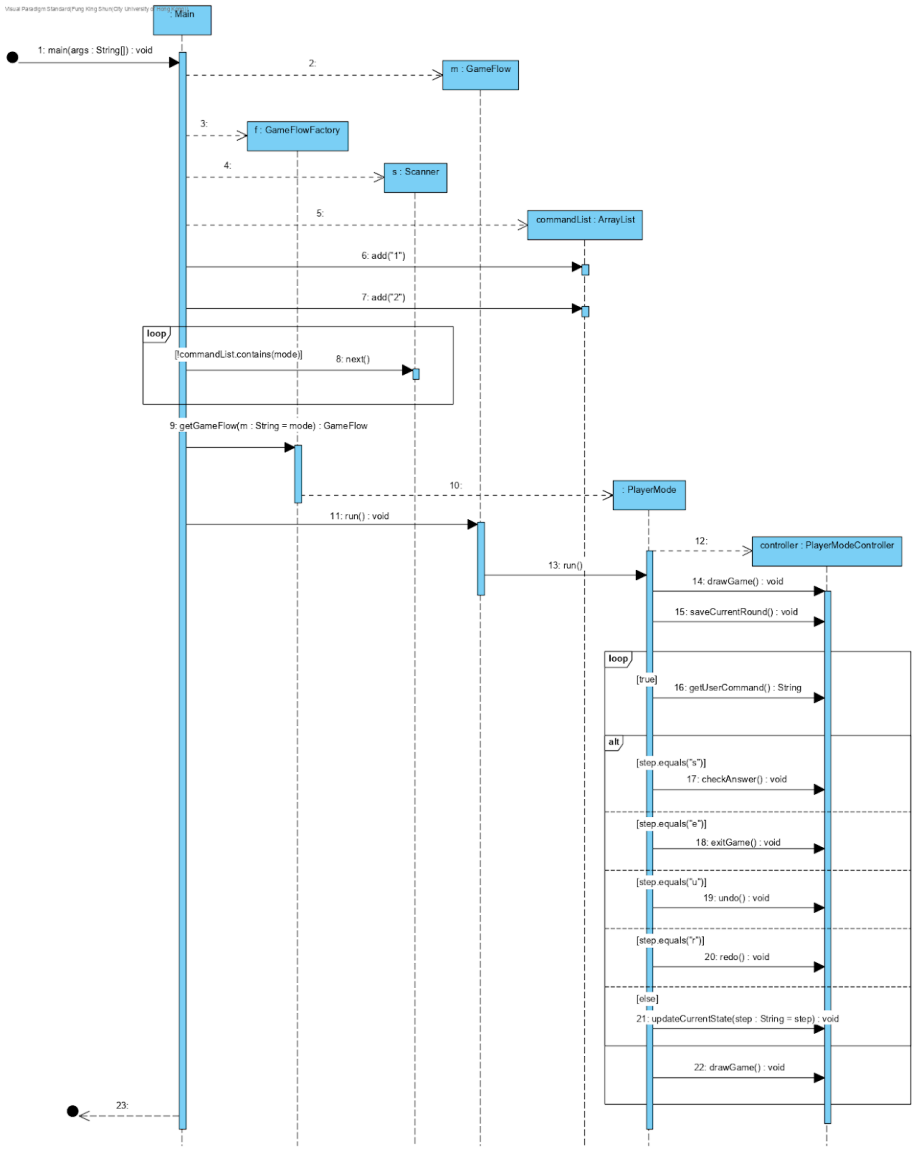
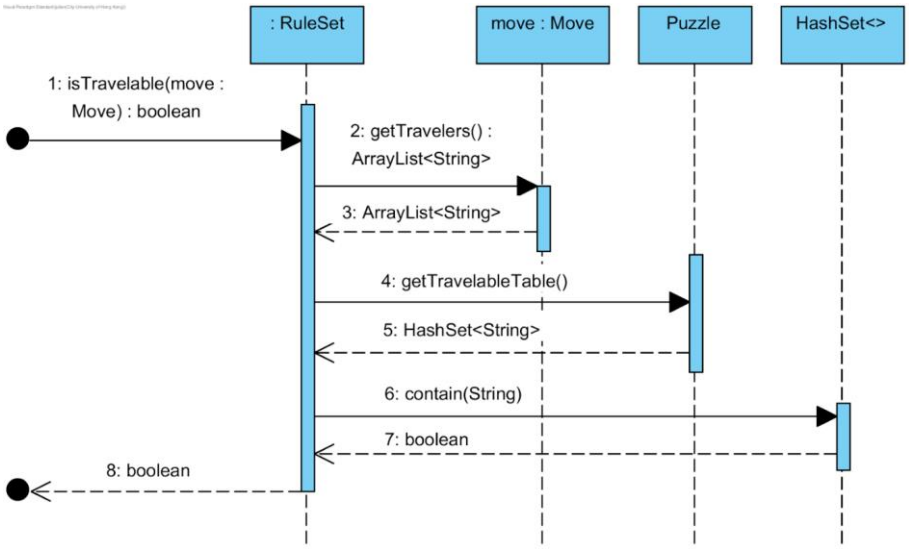


Figure 5.6 Sequence diagram for UI.PlayerMode

This sequence diagram is shows how to control the user input and operate as player mode. First, the process will start with main and create associated game factory object. Next, if the user select the game mode number as 2, it will entry to player model to create a PlayerModelController. After passed the process passed into PlayerModelController it will calls the drawGame() and saveCurrebtRound() method to display the UI and save the game record once which this part is initialization the game. Next, user can input different command like undo, redo, role name, exit, check answer, it is keeping in a loops until end game or game over.

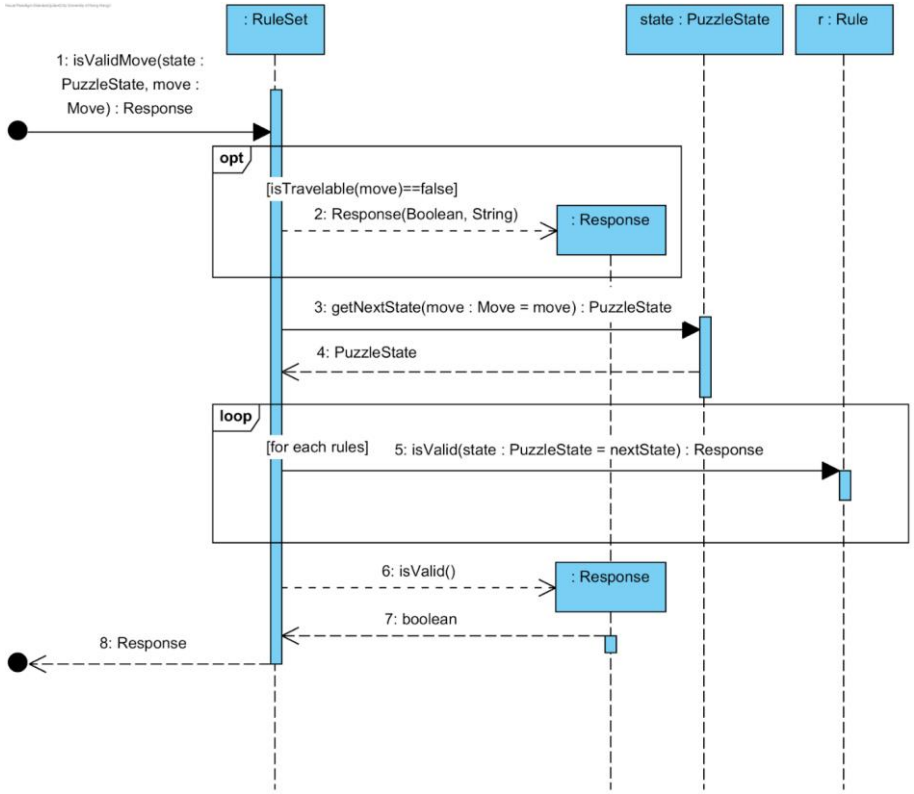
5.4. Sequence diagrams of Puzzle Logic module

5.4.1. isTravelable(Move move): Boolean



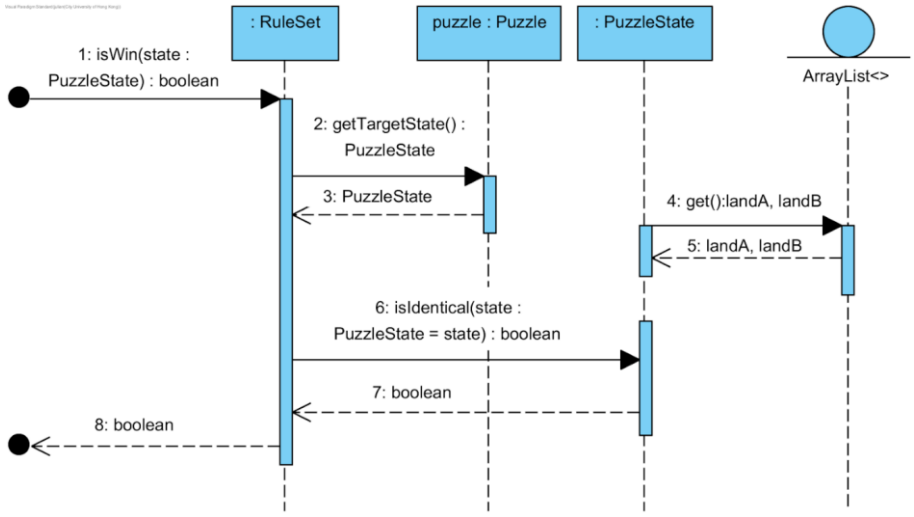
The isTravelable() method responds with a boolean object to the call from the UI module. This process checks the situation by comparing the travellers' list from the Move object and the rule list from the puzzle object. The algorithm checks whether the traveller's name/s is/are present in the rule list through the HashSet function.

5.4.2. isValidMove(PuzzleState state, Move move): Response



The isValidMove() method returns a response object to the caller - UI module. The method runs only when the answer of isTravelable() is True. This process checks the answer by comparing the travellers' list from the Move object and the rule list from the puzzle object. The algorithm compares whether or not the traveller's name/s with the role names in the destination land is/are in conflict. If not, then continuously compare the roles in the origin land location. The response object consists of a string attribute and a boolean attribute.

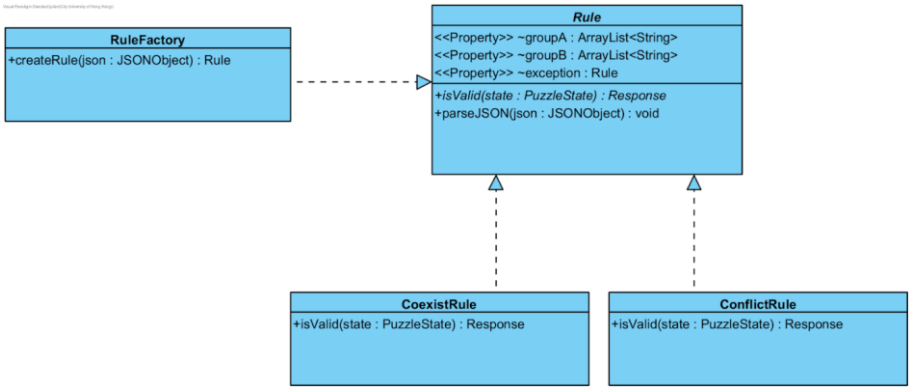
5.4.3. isWin(PuzzleState state): boolean



The isWin() method reply a boolean object to the caller - UI module by comparing the current state and the target state. The process returns a true value when both states are the same.

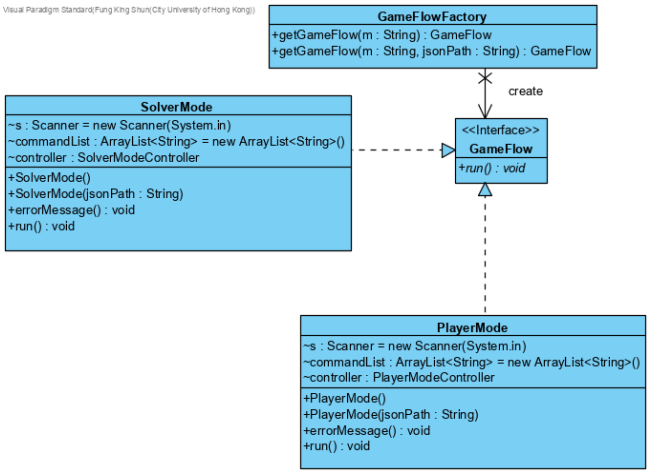
6. Design Pattern and principles

6.1. Factory pattern (Puzzle Logic module)



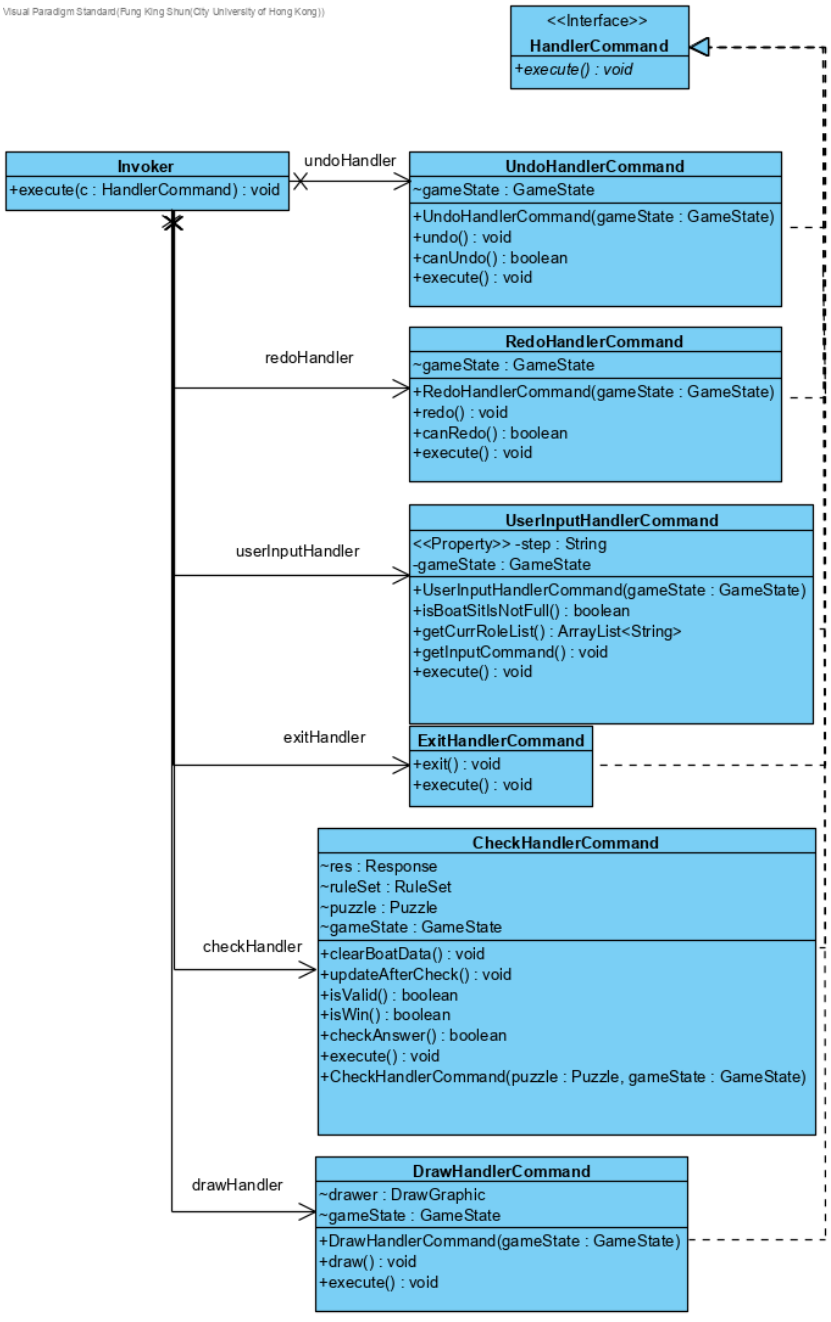
A new logic algorithm must be added to the module in case new game rules beyond the CoexisRule and ConflicRule can handle. The purpose of the Factory pattern designed in the logic module is simply the addition of new game rules. Such a pattern enables the logic module to open for any new game extension but does not necessarily modify any core code.

6.2. Factory pattern (UI module)



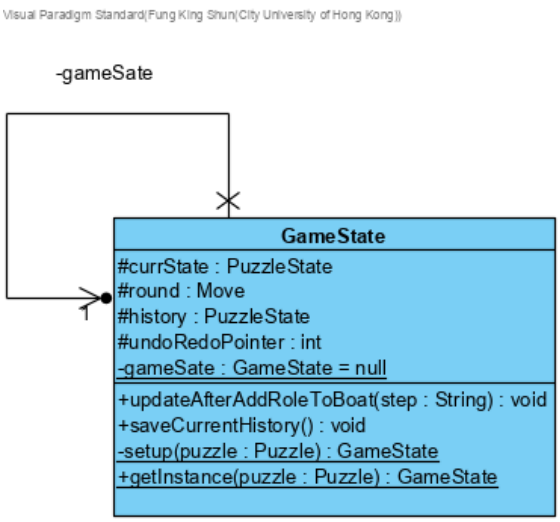
This is the factory pattern implemented in UI part, the idea is we will base on the user command input to create correspond game mode. This part is control by GameFlowFactory using two getGameFlow() method. Also, solver model and player mode is implement the GameFlow. So, after user selected the game mode, we just need to call the run() method to start the game.

6.3. Command pattern



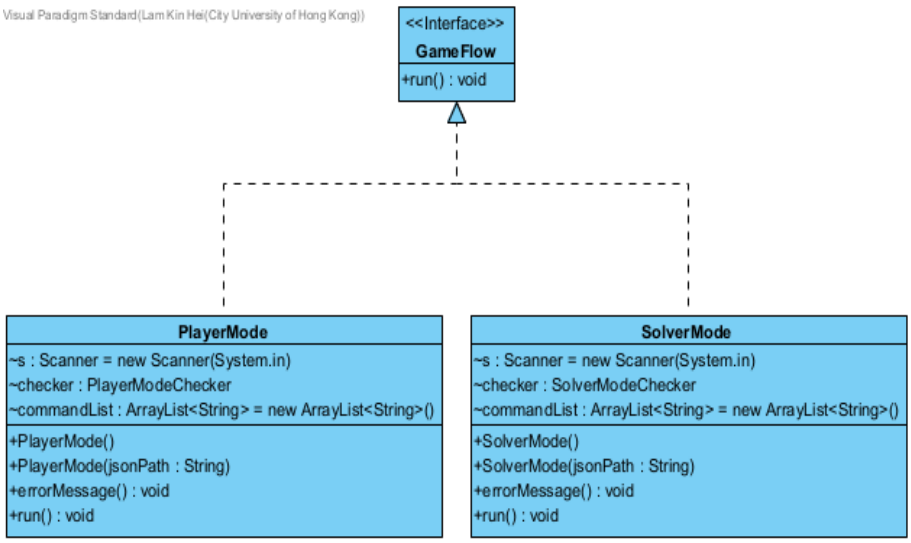
This is the command pattern we used in UI model. Because we need to handle different user input like redo, undo, role name, exit, chack answer, the design should be one class doing one command. Therefore we have different handler command shuch as UndoHandlerCommand, RedoHandlerCommand, UserInputHandlerCommand, ExitHandlerCommand, CheckHandlderCommand and DrawHandlerCommand. To run one of the command, it just nedd to use the invoker call execute() method like invoker.execute(DrawHandlerCommand), than it will run DrawHandlerCommand class execute() method run the the draw function.

6.4. Singleton pattern



The singleton pattern it used in UI model of player mode. Since in player mode we have different handler command, and some of the command have relationship like undo, redo, check answer. Their should be share with same game informate object and this object is GameState class. We use history and undoRedoPointer to process the game state and round and currSate is saving the current game information.

6.5. Open close principle



The open close principle is used for the gameflow. OCP stands for open for extension butclosed for modification. Since the gameflow isnt limited to two mode, there may have extra gameflow in the future. By adapting open close principle, it is still available for adding new game mode. Developer dont need to modify the original code which can have a better code maintaince.

7. Game interface

7.1. Select Game mode

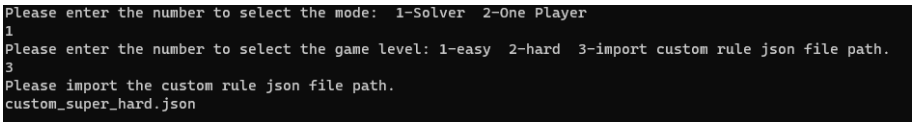


Figure.7.1 Selecting game mode and puzzles

There are two modes for our program, Player mode and Solver mode.

There are two puzzles included by default, which is the Easy and Hard puzzle. To extend the gaming method, users are able to input a custom puzzle JSON file to play with. For example, user can input a custom_super_hard mode to play the river crossing game or solver. If the JSON formatted correctly, the usage is just same as the perious easy and hard mode.

7.2. Player mode

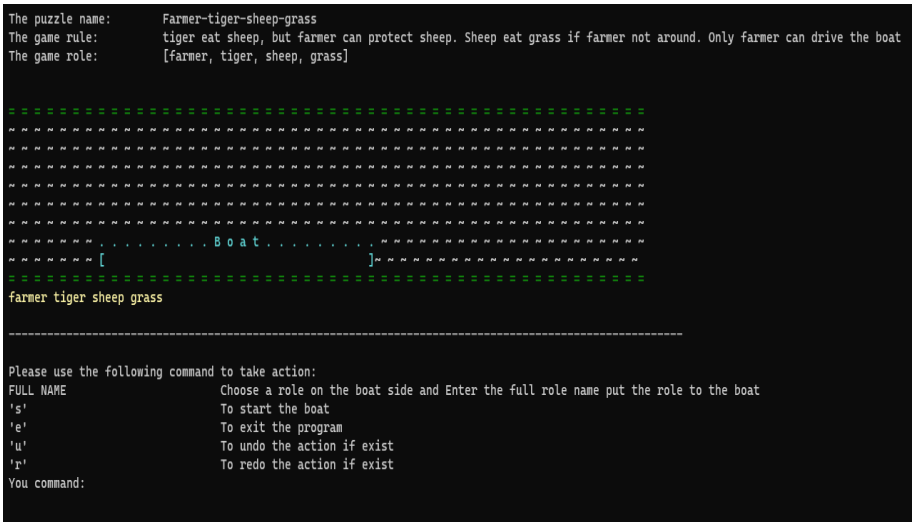


Figure 7.2 Player mode interface

The above diagram is the player mode interface. The name and rule of the puzzle are displayed at the top of the screen to guide the user. The middle part is the river with a boat, and the characters (game roles) are displayed on the land at the bottom of the screen.

The user needs to input the character name to let them get into the boat. Then the user can type 's' to start the boat. If the user inputs some wrong character name to get into the boat, they can input 'r' to redo the process. The character will return from the boat back to the original land. When all characters cross the river successfully, the player wins the game.

7.3. Solver mode

```
Please enter the number to select the mode:  1-Solver  2-One Player
1
Please enter the number to select the game level: 1-easy  2-hard  3-import custom rule json file path.
2

Initial State:
Land A: farmer tiger sheep grass (boat)
Land B:

Step 1
Move [farmer, sheep] from LandA to LandB
Land A: tiger grass
Land B: farmer sheep (boat)

Step 2
Move [farmer] from LandB to LandA
Land A: tiger grass farmer (boat)
Land B: sheep

Step 3
Move [farmer, tiger] from LandA to LandB
Land A: grass
Land B: sheep farmer tiger (boat)

Step 4
Move [farmer, sheep] from LandB to LandA
Land A: grass farmer sheep (boat)
Land B: tiger

Step 5
Move [farmer, grass] from LandA to LandB
Land A: sheep
Land B: tiger farmer grass (boat)

Step 6
Move [farmer] from LandB to LandA
Land A: sheep farmer (boat)
Land B: tiger grass

Step 7
Move [farmer, sheep] from LandA to LandB
Land A:
Land B: tiger grass farmer sheep (boat)
```

Figure 7.3 Solver mode interface

The above diagram shows the interface of the river crossing solver. The user can get the solution by running this function. Other than the solution of the easy and hard modes. The user can also input a custom JSON file to output a solution. The solution will detail show the step of the river crossing step, like how the characters move from the initial state of origin land A to destination land B. If there is no solution for the custom file, it will display no solution to the user.

8. Tools required

Commented [GU1]: explain

In the whole project cycle, we used different tools to support the development and management.

Functionality	**Tools**
Version control	Git, GitHub
Development	Eclipse Java Development Kit
Testing	Junit5
Documentation	Google docs Visual ParaDigm GanttProject draw.io Bugzilla

8.1. Development

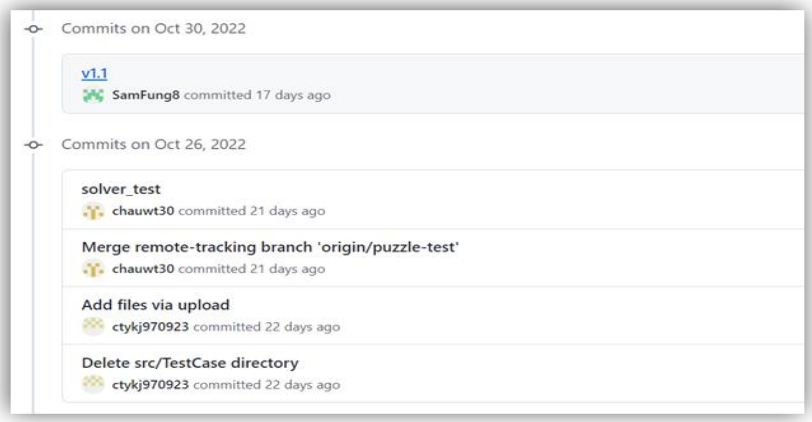
8.1.1. Eclipse IDE and Java Development Kit

```
import java.util.*;

public class Main{

    public static void main(String[] args) {
        GameFlow m;
        GameFlowFactory f = new GameFlowFactory();
        Scanner s = new Scanner(System.in);

        if(args.length > 0) {
            if(args[0].equals("-s")) {
                String jsonPath = args[1];
                m = f.getGameFlow("1", jsonPath);
                m.run();
            }else if(args[0].equals("-p")) {
                String jsonPath = args[1];
                m = f.getGameFlow("2", jsonPath);
                m.run();
            }
        }else {
            ArrayList<String> commandList = new ArrayList<String>();
            commandList.add("1");
            commandList.add("2");

            String mode;
            do{
                System.out.println("Please enter the number to select the mode:  1-Solver  2-One Player");
                mode = s.next();
            }while(!commandList.contains(mode));

            m = f.getGameFlow(mode);
            m.run();
        }

        s.close();
    }
}
```

For the development part, we use Java as the programming language with java development kit version 11. This version is the same among all team member to ensure the syntax consistency. We use Eclipse as the IDE as it has library for Junit testing.

8.2. Version control

8.2.1. Git and GitHub



To ensure better cooperation between team members, we use git and GitHub for version control. As it supports feature branch workflow, once the team member wants to extend the functionality, they just need to create a new branch for development, which do not affect the main program development. It also supports distributed development, once the programmer finish the code, they can just commit it. There is no need to keep sending the code though email to decrease the development time. At last GitHub support version tracking, since the development of river crossing puzzle is complicated, the algorithm may not be correct at once, with version control, we can roll back to previous versions which have a better code maintaince.

8.3. Testing

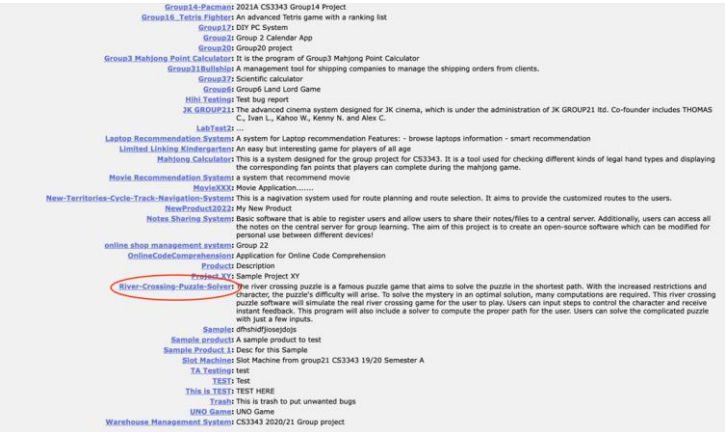
8.3.1. Junit5

```
@Test
void testGetNextState1() {
    ArrayList<String> TlandA = new ArrayList<String>(Arrays.asList("farmer", "sheep"));
    ArrayList<String> TlandB = new ArrayList<String>(Arrays.asList("tiger", "grass"));
    int initialOrgin = 0;
    PuzzleState ps = new PuzzleState(TlandA,TlandB,initialOrgin);
    Move m = new Move("farmer","sheep",0,1);
    PuzzleState newState = ps.getNextState(m);

    ArrayList<String>ExplandA = new ArrayList<String>(Arrays.asList());
    ArrayList<String>ExplandB = new ArrayList<String>(Arrays.asList("farmer", "sheep","tiger", "grass"));
    int Exppos = 1;
    PuzzleState expState = new PuzzleState(ExplandA,ExplandB,Exppos);

    boolean result_state = newState.isIdentical(expState);
    int result_pos = newState.getBoatPos();

    String msg;
    msg = "Testing two travellers successfully travel from landA to landB";
    assertEquals(true,result_state,msg);
    //only test boat position if there are travelers
    if(!(TlandA.containsAll(ExplandA) && TlandB.containsAll(ExplandB))) {

    msg = "Testing boat on landB(1)";
    assertEquals(1,result_pos,msg);
    }
}
```

Commented [LH2]: later need label

For testing part, we use the Junit5 library which is the latest version. It supports Java language and is suitable for our development process. With different unit test and integration test. We can ensure the program bug is minimised and function as expected.

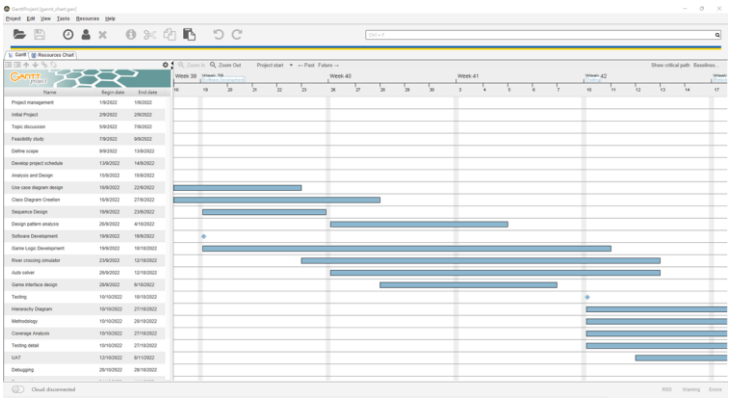
8.3.2. Bugzilla



As show in above photo, the river crossing puzzle solver has include in Bugzilla which is a bug tracking system. When the program tester discovers bug, he will report to the system. After that a bug report will be generated, include the step to recreate the bug and the developer assigned to fix the bug. Using this tools, the programmer is easier to follow and debug.

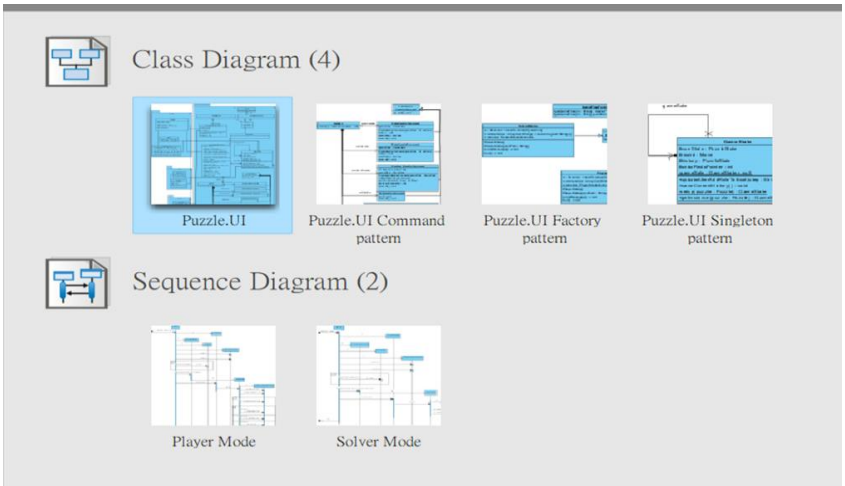
8.4. Documentation

8.4.1. Gantt project



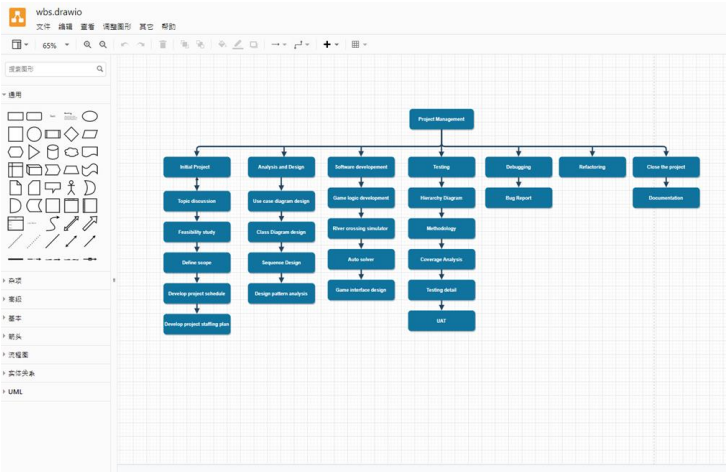
To ensure better project management, we use Gantt Project to create Gantt chart. Each task has its own duration and expected finished day. Team members can refer to it to organise the time. By following this project timeline, the project can delivery on time, which highly decrease the chance of project failure.

8.4.2. Visual Paradigm



To perform analysis and design, we use visual paradigm to create diagram. By using this tools, we have created use case diagrams, class diagrams and sequence diagrams to show our program structure and characteristics. With this kind of diagram, the complicated logic and flow can be present easily.

8.4.3. Draw.io



To create different diagram in the project plan, we also used draw.io to draw diagrams. It allows multi user edit and can present in a clear structure. We have use this tool to create work break down structures and people management diagrams.