**City University of Hong Kong**

CS3343 Software Engineering Practice

Analysis and Design Report

Project Title: TankWar

|  |  |
| --- | --- |
| **Name** | **Student Number** |
| Hong Yifan | 57853960 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Table of Contents**

[1. Design Constraints 4](#_Toc437637559)

[1.1. Technical Feasibility 4](#_Toc437637560)

[1.2. Available Operating System 4](#_Toc437637561)

[1.3. Graphical User Interface 4](#_Toc437637562)

[1.4. History of the game 4](#_Toc437637563)

[1.5. Way to Move Piece 4](#_Toc437637564)

[2. Requirement Specification 5](#_Toc437637565)

[3. Use Case Diagram 5](#_Toc437637566)

[4. Use-Case Specification 6](#_Toc437637567)

[5. Class Diagram 8](#_Toc437637568)

[6. Class Description 9](#_Toc437637569)

[6.1. ChessUI Class 9](#_Toc437637576)

[6.2. Chess Classes 10](#_Toc437637577)

[6.3. Main Class 11](#_Toc437637578)

[6.4. Command Handler Class 11](#_Toc437637579)

[7. Design Pattern 12](#_Toc437637580)

[7.1. Model-View-Controller (MVC) Pattern 12](#_Toc437637582)

[7.2. Benefits of Adopting MVC Design Pattern 12](#_Toc437637583)

[Shorten Development Cycle 12](#_Toc437637584)

[Providing Testable Function 12](#_Toc437637585)

[7.3. State Pattern 14](#_Toc437637586)

[7.4. Benefit of Adopting State Pattern 15](#_Toc437637587)

[Increase its Maintainability 15](#_Toc437637588)

[Easy to Test 15](#_Toc437637589)

[8. Design Principle 16](#_Toc437637590)

[8.1. Open-Closed Principle 16](#_Toc437637592)

[8.2. Liskov Substitution Principle 18](#_Toc437637593)

[9. Sequence Diagram - Chess Movement 21](#_Toc437637594)

[9.1. Sequence Diagram (receiveCommand) 21](#_Toc437637596)

[9.2. Sequence Diagram (CrossableChess.validMovement(position Position)) 23](#_Toc437637597)

[9.3. Sequence Diagram (NonCrossableChess.validMovement(position Position)) 25](#_Toc437637598)

[10. Refactoring 27](#_Toc437637599)

[10.1. Cannon class 27](#_Toc437637601)

[Code before Refactoring – Scanning X Axis for Possible Move 27](#_Toc437637602)

[Code after refactoring – Extract Method 29](#_Toc437637603)

[10.2. Horse Class 30](#_Toc437637604)

[Code before Refactoring – Check if Any Foot Kicks 30](#_Toc437637605)

[Code after refactoring – Substitute Algorithm 31](#_Toc437637606)

# Design Constraints

## Technical Feasibility

Our design for the software is technically feasible by Java program. To achieve this, we will gather the information from clients and draw them into a use case diagram. And then, we apply different design principle, pattern, and object-oriented approach into system to create the architecture of the software. In addition, we have researched on the existing software on the market. Therefore, it has proven that the software is technically feasible.

## Available Operating System

The Java application is designed to be run in Mac OSX, and Windows so the system may not be runnable in mobile device like iPhone, HTC smartphone which running Android and iOS system. Therefore, the application needs extra effort in expanding it to other platform.

## Graphical User Interface

The chess software is used for professional training, so the user interface is not required to be fancy. However, a simple graphical user interface is required for user to view the chessboard visually which can help end-user understand the current situation of the game. The GUI must include all the chesses indicating its color and type. Although it is not the top priority, a better GUI can gain more user satisfaction.

## History of the game

The software should log every move of the chess game so as to allow user to track down the step he/she made. The history of the game is one of the important features of the software. The history can help user review every step he/she moved and determine whether such move is a good move. As required by client, the history can be editable and can be stored by user. Therefore, the history should be located into a text field which can let user make amendment and copied it out.

## Way to Move Piece

The method for moving pieces should be simple and clear. For the students, they use the software to review some famous chess games or investigate the variations. Thus, they required to a quick way to move the chess, for example, moving the chess according to the coordinates. Therefore, a text field for them to move pieces is preferable.

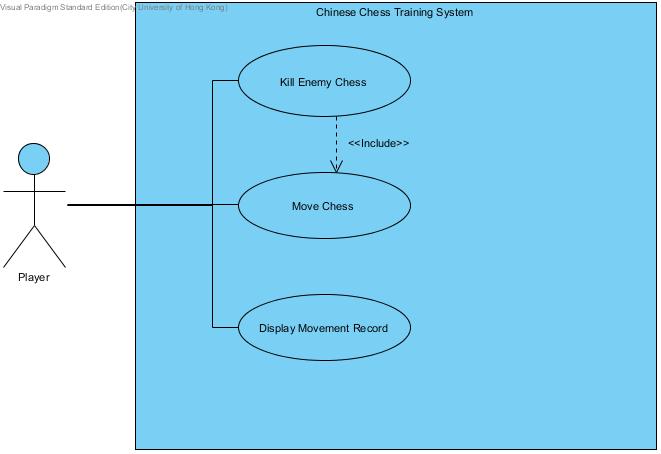
# Requirement Specification

**Player**

Chinese Chess Training Software

* Kill Enemy chess
* Move Chess
* Display Movement Record

# Use Case Diagram



1. Use-case Diagram

# Use-Case Specification

|  |  |  |
| --- | --- | --- |
| **Use Case ID:** | UC 1000 | |
| **Use Case Name:** | Move Chess | |
| **Actor(s):** | Player | |
| **Description:** | Player wishes to move one of the chess must input the command in order to do so, such action will invoke this use case | |
| **Trigger:** | This use case is initiated when the player entered command into the console | |
| **Normal Flows:** | **Actor Action** | **System Response** |
| 1. Player entered a command with proper X Y source and destination coordination | 1. The system received the command and move the selected chess to destination that user already specified. |
| **Post-conditions:** | Chess selected by player has been moved to location specified. | |

|  |  |  |
| --- | --- | --- |
| **Use Case ID:** | UC2000 | |
| **Use Case Name:** | Kill Enemy Chess | |
| **Actor(s):** | Player | |
| **Description:** | Player attempts to eliminate one of the enemy chesses | |
| **Trigger:** | This use case is initiated when the player entered command into the console | |
| **Normal Flows:** | **Actor Action** | **System Response** |
| 1. Player entered a command with proper X Y source and destination coordination | 1. The system received the command and move the selected chess to destination that user already specified. 2. The chess belongs to enemy will be eliminated from the board |
| **Post-conditions:** | Chess selected by player has been moved to location specified and the enemy chess stationed at that location will be eliminated. | |

|  |  |  |
| --- | --- | --- |
| **Use Case ID:** | UC3000 | |
| **Use Case Name:** | Display Movement Record | |
| **Actor(s):** | Player | |
| **Description:** | Player reviews the movement log which contains all movement made in the past | |
| **Trigger:** | This use case is initiated when the player entered command into the console | |
| **Normal Flows:** | **Actor Action** | **System Response** |
| 1. Player entered a command randomly. | 1. Movement record will be displayed to user. |
| **Post-conditions:** | Movement Record will be displayed to user. | |

# Class Diagram3.pngClass Diagram

Fig. 2 Class Diagram

# Class Description



## ChessUI Class

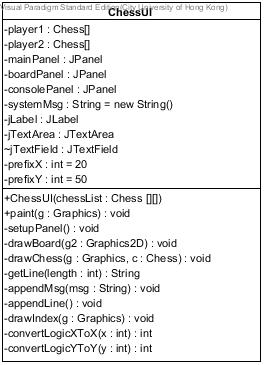


Fig. 3 ChessUI Class Diagram

This classes is mainly used to render the graphical user interface (GUI) including the chess board, and it will render the chess board based on the data stored in the Chess [ ][ ] array. This class will be invoked after the controller finishes the updated game state. Therefore, user can see the chess board is updated after he/she entered a command.

## Class Diagram3.pngChess Classes

Fig. 4 Chess Class Diagram

There are different kinds of Chinese chess classes; each chess object stores its position associated with position class. In addition, they will have a common method called “getPossibleMove” to get all the next possible move of that chess. With the method, we will be able to check if the chess is moving to a valid position, when the user enters a command to move the chess.

## Main Class

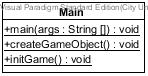


Fig. 5 Main Class Diagram

Main class is the class to be called at the very beginning. Therefore, main class is used to establish connection between the chess list and chess UI class so that chess board can be rendered and the game can be initialized by setting all chesses into original position.

## Command Handler Class

Command handler class is the core component in this system, although it is transparent to our user. Command handler will update the chess position, if the command, which user entered, is valid. Command handler will also be responsible for returning an error message to chess UI, if the command is invalid.

# Design Pattern



## Model-View-Controller (MVC) Pattern

In the class diagram, there are totally 16 classes in this system. They are conceptually separated into three kinds of classes. The first kind of class is responsible for storing the game data, and the game state (Model). The second kind of class is responsible for rendering the user interface and interacting with the user (View). The third kind of class is responsible for handling the user input and updating the game state (Controller). The classes are categorized as followed:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Storing the game data and its state (Model)** | | | | |
| Chess | CrossableChess | NonChrossableChess | Cannon | Chariot |
| Elephany | General | Horse | Scholar | Solider |
| Position | GameData | Constants |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Handling user input and updating game state (Controller)** | | | | |
| CommandHandler | Main |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rending user interface and interacting with user (View)** | | | | |
| ChessUI |  |  |  |  |

## Benefits of Adopting MVC Design Pattern

There are two advantages for game development with MVC design pattern

### Shorten Development Cycle

With MVC design pattern, we can develop the UI part and logic part concurrently. In the traditional development process, we have to develop either the logic or the UI first, and then develop the remaining one. However, with this design pattern, we can define some methods to link logic with UI classes before developing. Once the method is defined, we can develop logic and UI independently. And at the final stage - integration, we can simply combine two of them by calling the method defined.

### Providing Testable Function

MVC design pattern can be integrated with the test-driven development. The MVC ensure the inter-independence of Model and View, while Controller acts as the intermediary between them. Therefore, testing team can easily write test cases for the logic of the program without the hindrance of user interface. For instance, tester can write the test case for getPossibleMove() of each chess type with the absence of UI so the percentage of test coverage can be improved.

## Class Diagram3.pngState Pattern

Fig. 6 State Pattern Diagram

In Chinese chess game, different kinds of chess have different rules. For example, for “Chariot”, it moves and captures any distance orthogonally, but it cannot jump over the intervening pieces. For “Cannons”, it can move any distance orthogonally without jumping, but can only capture by jumping over a single piece, friend or foe, along with the path of attack.

Therefore, we create different classes for different kind of chess. Every chess can be moved and having its own position. Therefore, we create an abstract class named - “Chess” for it. All the different kind of chess is inherited from it.

## Benefit of Adopting State Pattern

### Increase its Maintainability

Using the traditional method to implement this function, we have to write an if-else statement to separate the way to handle different kind of chess. And it will turn out a lengthy method which is uneasy to modify. However, using state pattern, we can implement different kind of rule within the different kind of concrete chess classes. With the dynamic binding mechanism introduced by Java, the system will be able to choose the corresponding block of code to execute so as to reduce the if-else statement.

### Easy to Test

Using state pattern, tester will be able to test a small unit of code instead of a batch of if-else statement. The obvious benefit is to reduce the branches which can reduce the number of test cases. Another benefit is to enable tester to test the specific block of codes with writing test stub to reduce the function dependency.

# Design Principle



## Open-Closed Principle

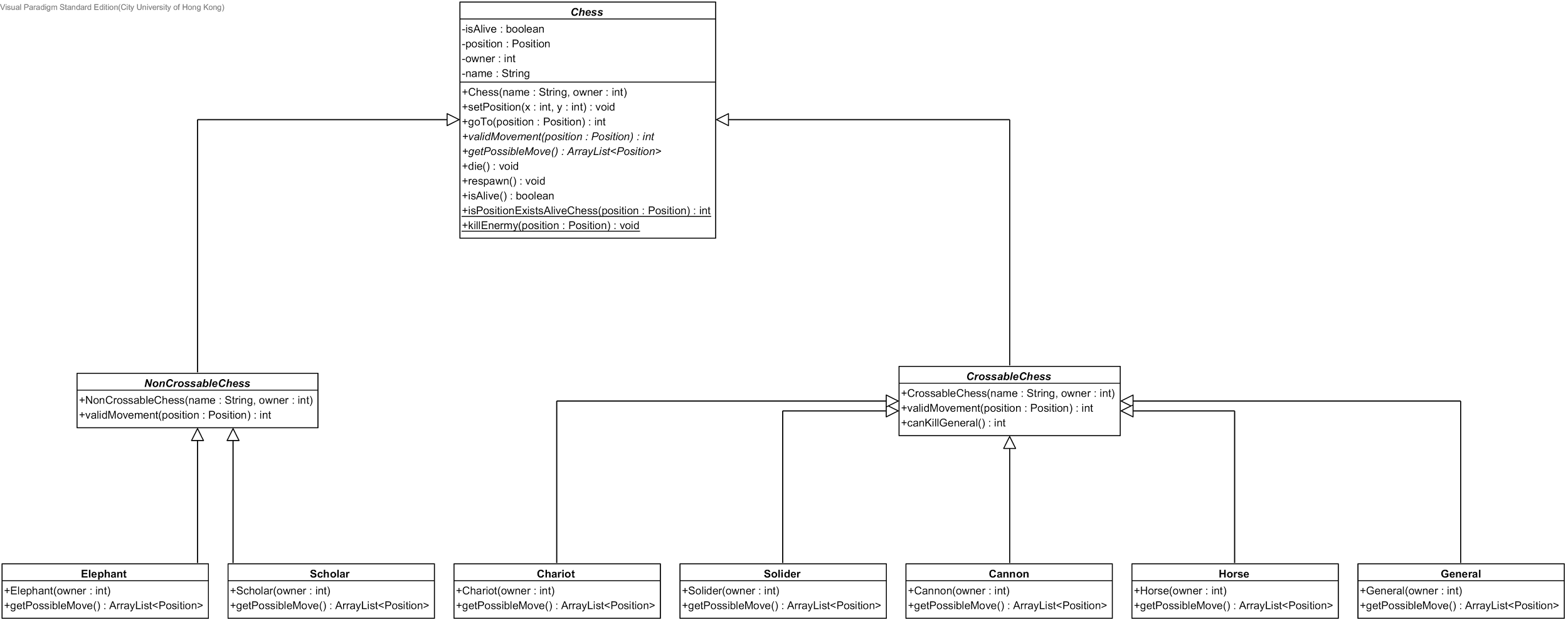


Fig. 7 Open-Closed Principle Diagram 1

The definition of open-closed principle is that open for extension and closed for modification. And for a well-known game like “Chinese Chess” game, it is not possible to add any other new rule or new kind of chess. However, bugs maybe found out in the future, for a specific kind of chess.

If that case is happened, we can simply write a new chess class in replace of the buggy one in the system. Therefore, our programmer can easily maintain and debug the game without modifying the existing code.

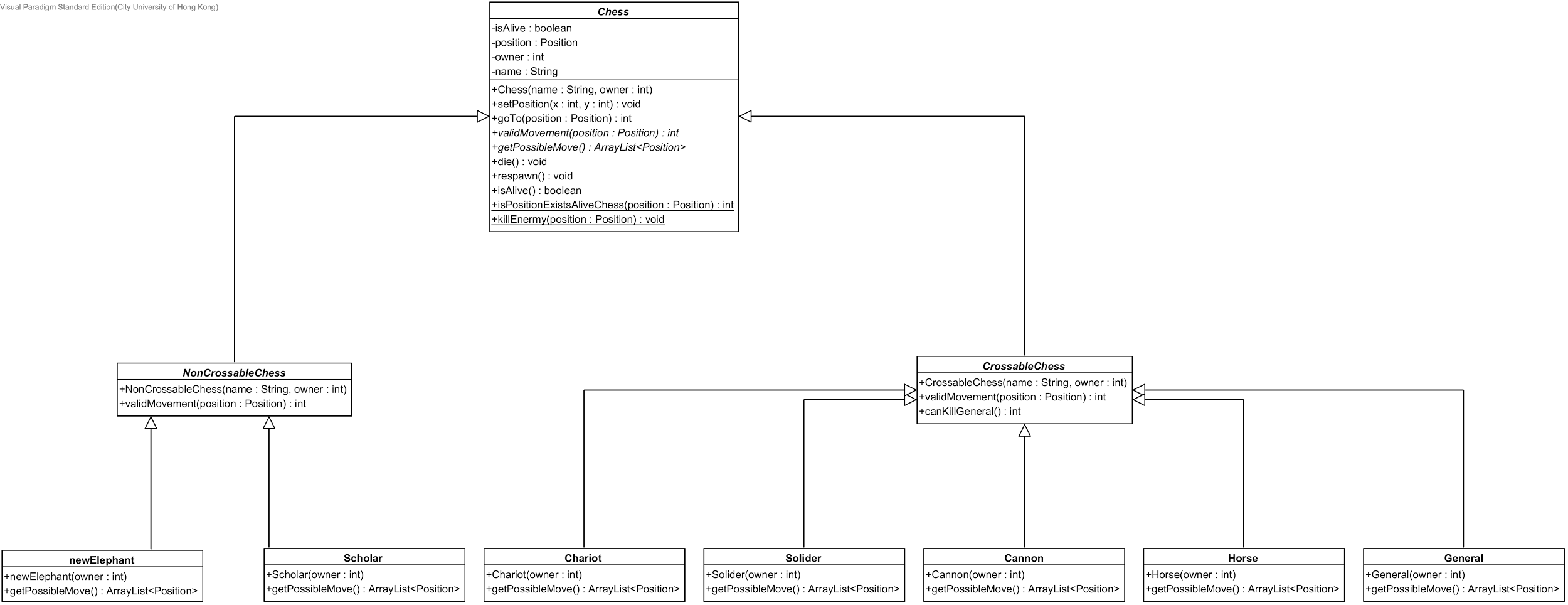


Fig. 8 Open-Closed Principle Diagram 2

For example, if we want to find a bug in Elephant class, then, we can simply add a new Elephant class and use it to replace the “Elephant” class. Therefore, it will be closed for modification.

## Liskov Substitution Principle

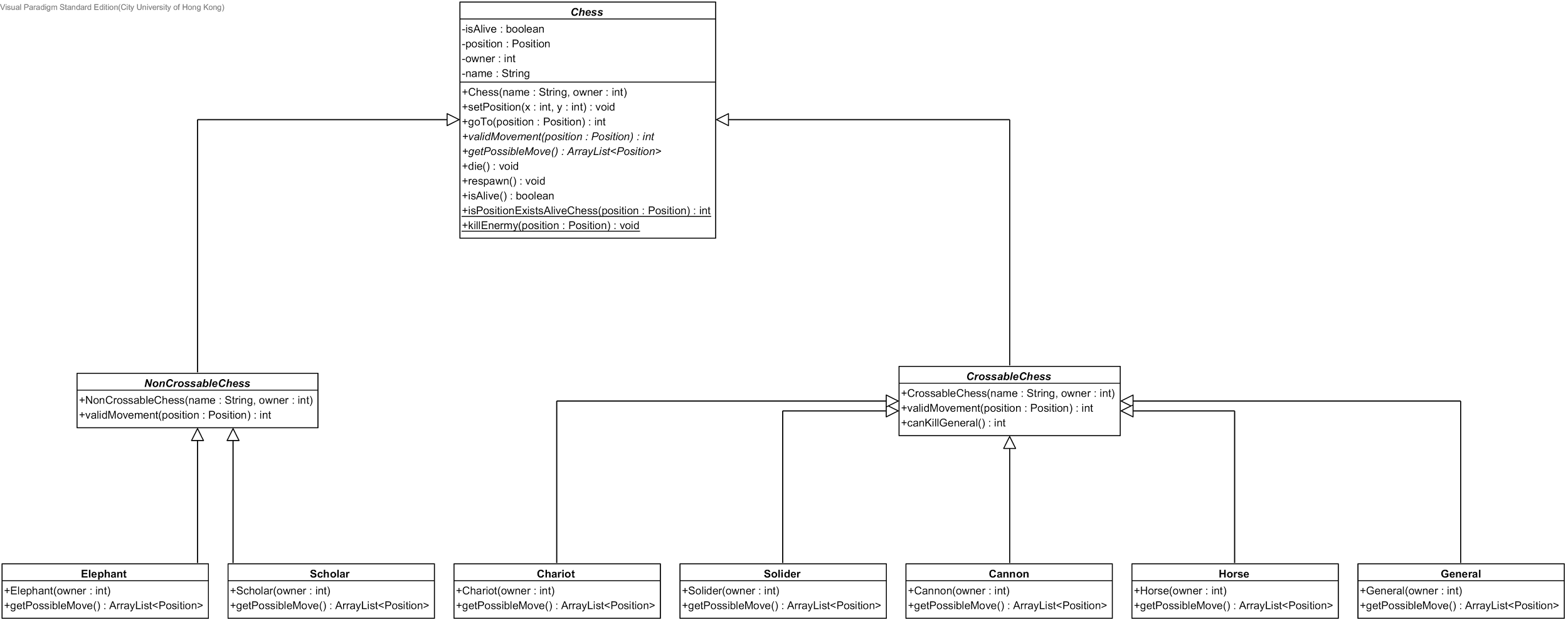


Fig. 9 Liskov Substitution Principle Diagram 1

The definition of Liskov substitution principle is that derived types must be completely substitutable for their base types.

In Chinese chess, despite of the fact that all the chess can be moved, some of them cannot cross the river, but some of them will be able to do that. Therefore, we designed our chess model in a three level architecture, in order to follow the Liskov substitution principle.

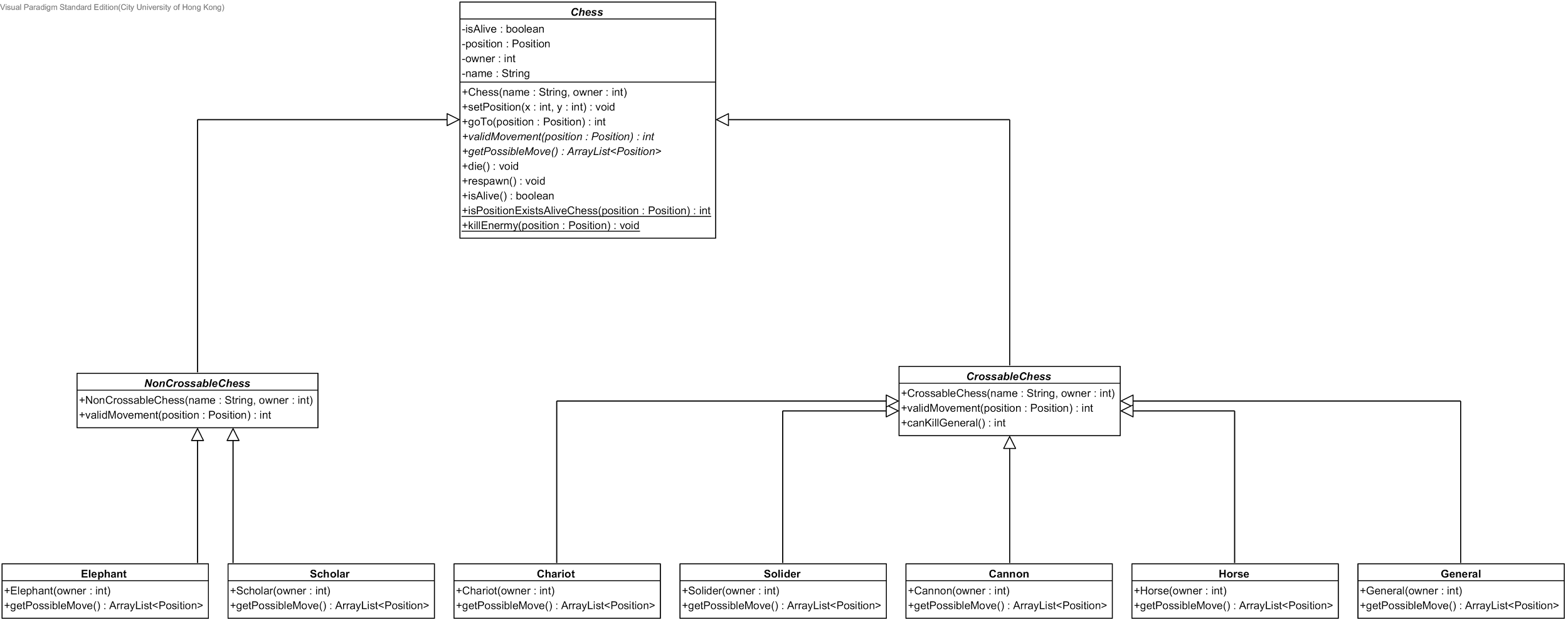


Fig. 10 Liskov Substitution Principle Diagram 2

The abstract class – “Chess”, which is the top level, is used to store and describes all the common actions and attributes of each type of chess.

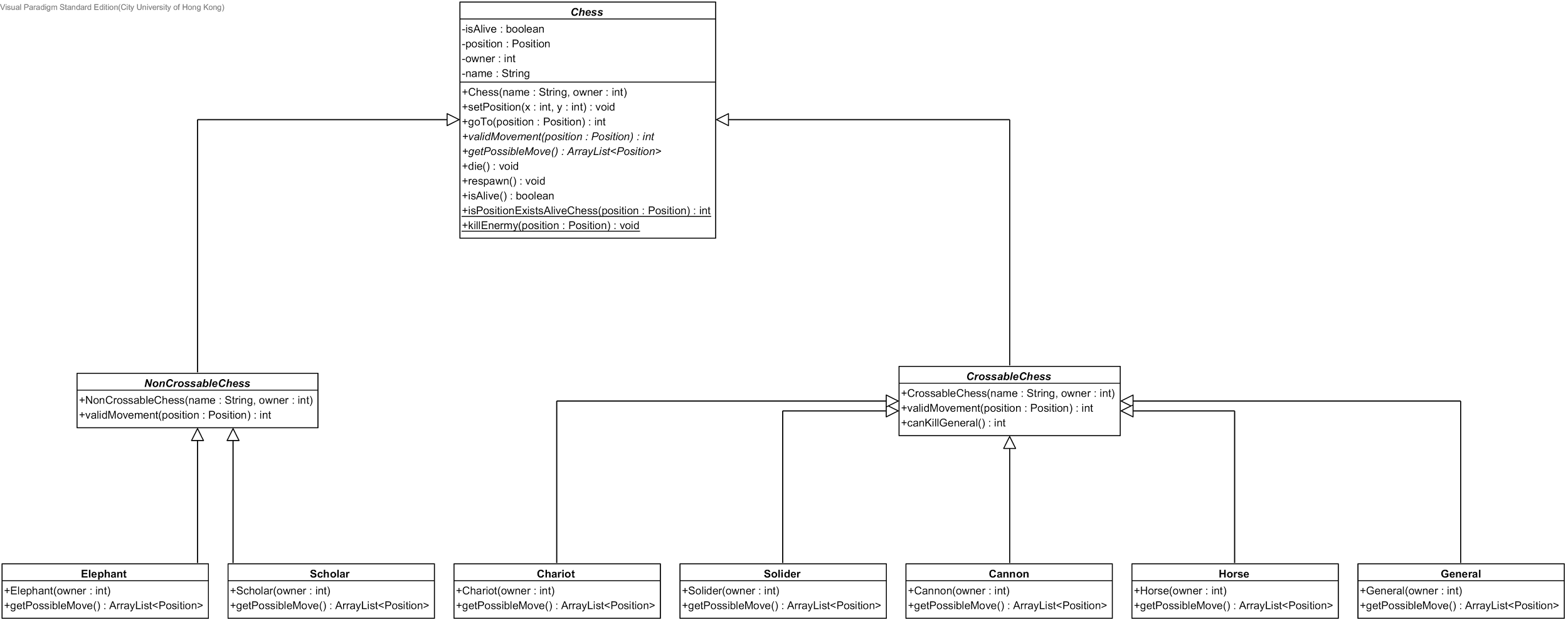


Fig. 11 Liskov Substitution Principle Diagram 3

The intermediate level is the level added, in order to satisfy the condition of Liskov substitution principle. In this level, there are two abstract classes. The first one is “CrossableChess”, another one is “NonCrossableChess”. Crossable Chess is the parent class of all the chess that can cross the river; the other one is the parent class of all the chess that cannot cross the river.

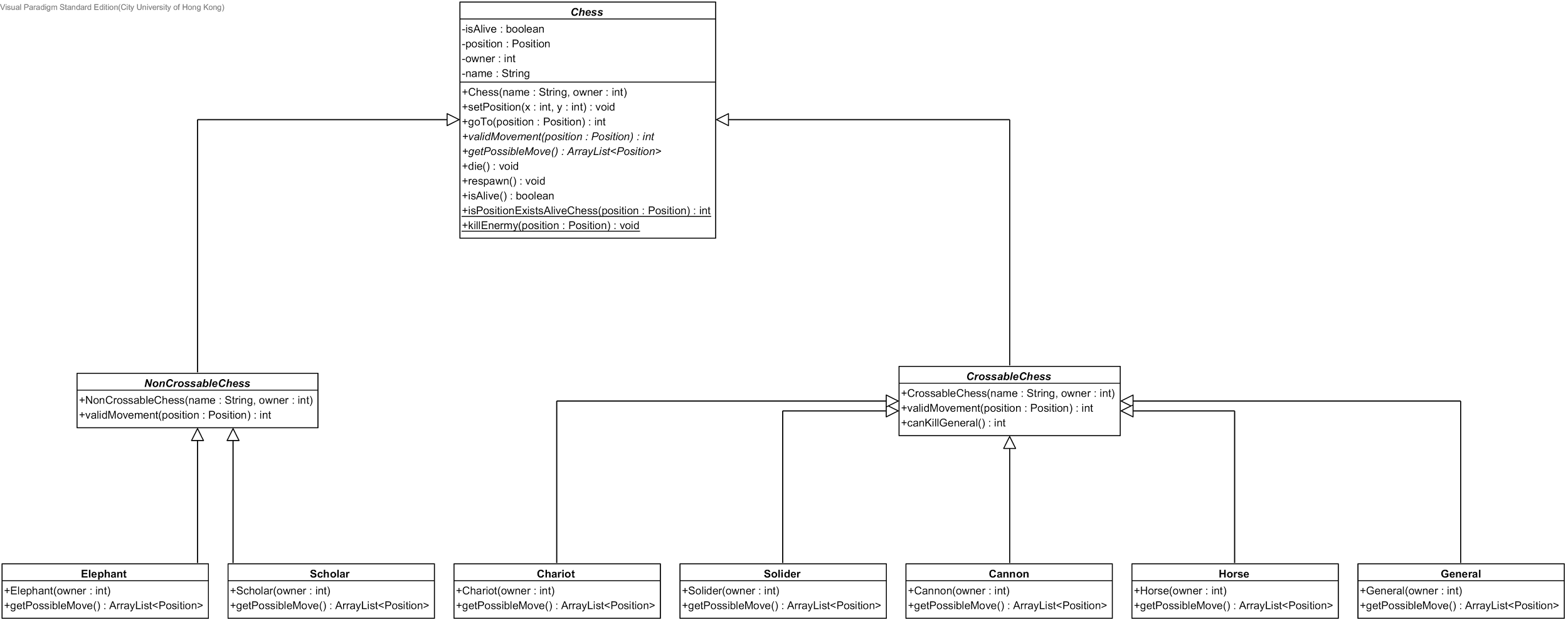


Fig. 12 Liskov Substitution Principle Diagram 4

The third level is the concrete class level, and it will be responsible for computing the possible move of the chess and return the possible move array for controller, so that controller will be able to validate the user command.

# Sequence Diagram - Chess Movement

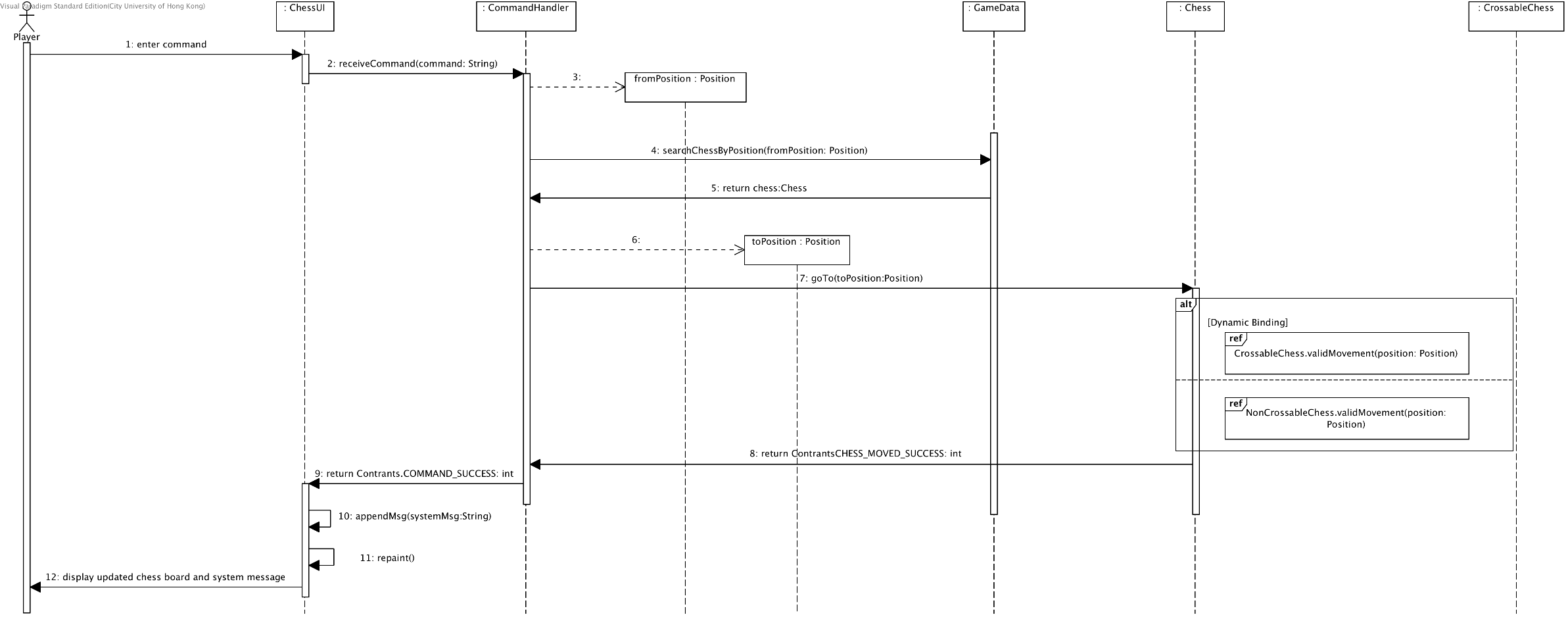
The sequence of moving chess will invoke the user case - "Move Chess". Before the user can move a chess, he must write the positions in the text field, and then the system will receive the command and proceed to input validation. If the input is valid, the chess position will be updated. The message regarding successful movement will also print on the history log. Additionally, the chessboard will be updated so that user can visually understand the chess has been moved to desired position.



## Sequence Diagram (receiveCommand)

In this sequence diagram, user needs to input command (with instruction: old position X, old position Y, new position X, new position Y) in user interface. The command handler will receive and handle the command to search the chess by its old position. If the chess is found, system will check if the move is valid. Once system confirm it is a valid move, it will then send successful message and update the chess board

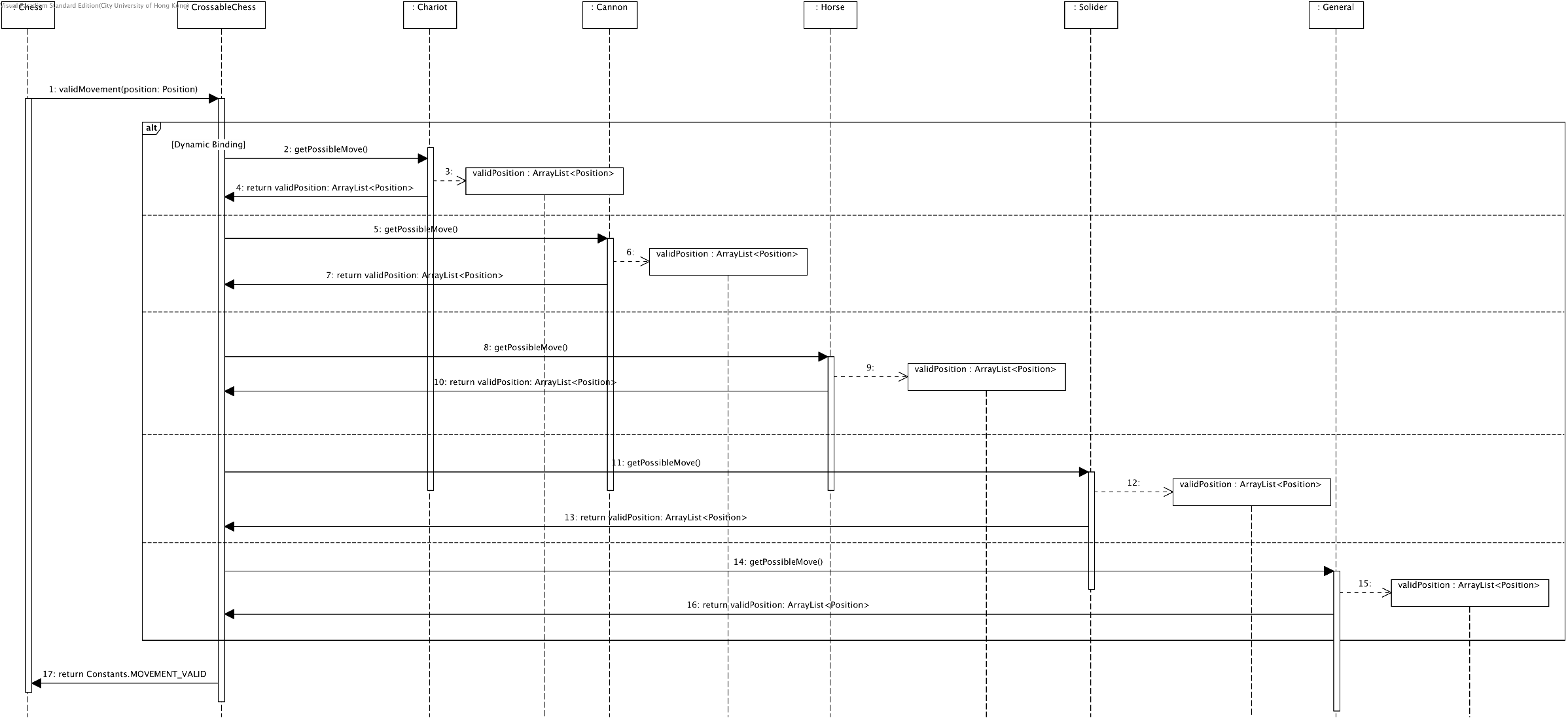
1. User enters command in user interface
2. After receiving command, system will search chess by position
3. System will check if the move is valid if the chess is found
4. System will then print successful message and update the chessboard for valid movement

 Fig. 13 Sequence Diagram (receiveCommand)

## Sequence Diagram (CrossableChess.validMovement(position Position))

Position is given to check the validation of such move according to different chess type belonging to crossable chess. If the move is valid, system will return a successful message

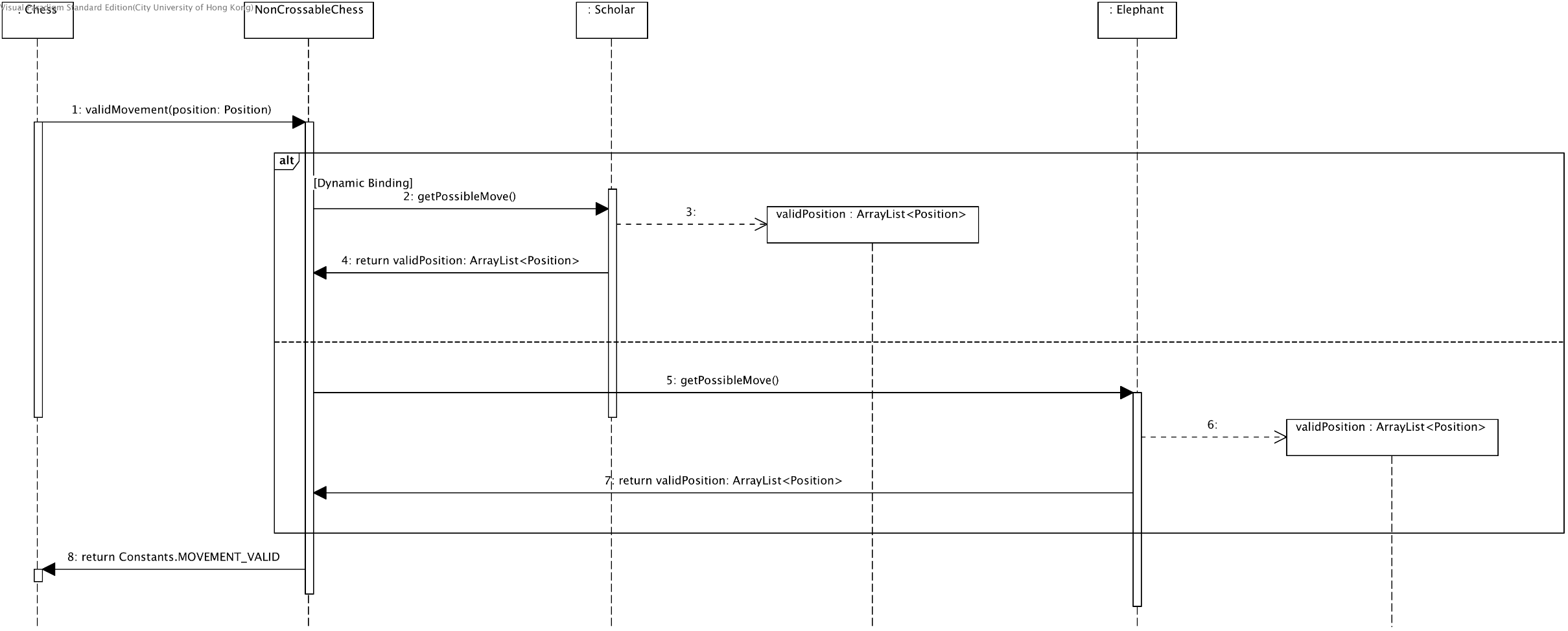
1. The position input by user will be sent to system for validation
2. The system will get the possible moves according to its chess type
3. System will then check if the new position will match with the old position
4. If it matches, it will return valid message.

Fig. 14 Sequence Diagram (CrossableChess.validMovement(position\_Position))

## Sequence Diagram (NonCrossableChess.validMovement(position Position))

Position is given to check the validation of such move according to different chess type belonging to non-crossable chess. If the move is valid, system will return a successful message.

1. The position input by user will be sent to system for validation
2. The system will get the possible moves according to its chess type
3. System will then check if the new position will match with the old position

Fig. 15 Sequence Diagram (NonCrossableChess.validMovement(position\_Position))

# Refactoring

After testing and debugging, the code should be refined to gain a better performance and become more testable. In the testing stage, extra test cases are added to fully test the functions due to excessive nested If-statement. Additionally, the codes developed initially contain redundancy, and are too complicated. Therefore, simplification is made, and some examples will be discussed below.



## Separate render Explosion method into multiple methods with clear responsibilities in explosion.java

Code before Refactoring

文本

描述已自动生成

Motivation for Improvement:

1. The old version violates Single Responsibility Principle by mixing multiple functionalities (state checking, drawing, and animation updates) in one method
2. No preservation and restoration of the original graphics color state
3. Lack of proper boundary checking
4. The method is too large and complex, making it difficult to maintain and test

Improvement Strategy:

1. Split the method into three independent methods:

* checkExplosionState(): Checks explosion status
* drawExplosionFrame(): Renders explosion frame
* updateAnimation(): Updates animation state

2. Add appropriate documentation and comments

3. Add color state preservation and restoration

4. Implement proper boundary checking logic

Code after refactoring 文本

描述已自动生成

文本

描述已自动生成

Benefits after Improvement:

1. Each method has a single responsibility, making it easier to maintain and test
2. Clear code structure with improved readability
3. Proper documentation for each method's functionality
4. Correct handling of graphics color state
5. Adherence to Single Responsibility Principle
6. Easier to extend in the future (e.g., adding new explosion effects)

## Refactoring Collision Handling Logic in WallManager.java

Code before Refactoring

文本

描述已自动生成

1. Motivation
   1. Both methods contain similar code structure
   2. Null checking logic is scattered
   3. Collision handling logic lacks uniformity
   4. Method names could be more descriptive
2. Improvement
   1. Create a generic collision handling method
   2. Unify null checking logic
   3. Use generics for different collision types
   4. Provide clearer method naming

Code after refactoring

文本

描述已自动生成

Benefits after improvement:

1. Reduced code duplication
2. Unified null checking
3. Easier to extend for new collision types
4. Clearer code structure
5. Lower maintenance cost

## Refactoring Distance Calculation Methods in TankAI.java

Code before Refactoring

文本

描述已自动生成

Motivation

* + 1. Three methods for distance calculation with duplicated code
    2. Different method names cause confusion in usage
    3. Lack of unified interface for distance calculation
    4. High maintenance cost

Improvement

* 1. Create a unified distance calculation utility
  2. Use method overloading instead of different method names
  3. Provide clear documentation
  4. Ensure calculation consistency

Code after Refactoring

文本

描述已自动生成

电脑屏幕的截图

描述已自动生成

Benefits after improvement

* + 1. Code is more concise and unified
    2. Eliminated code duplication
    3. Easier to maintain and test
    4. Provides clear method overloading
    5. Improved code readability