Háskólinn í Reykjavík

Assignment 2

T-528-HLUT, Hlutbundin forritun í C++

Gylfi Andrésson

Ívar Markússon

Teacher:

Yngvi Björnsson

14. maí 2017

1 Introduction

We were assigned an assignment in two parts, the first part was to design and implement an engine for generic two-player board games, for example chess-like games. The engine had to have a well-defined interface and default behaviors as appropriate. Then another team would then use our engine to write different board games. Now for the twist, we are also the other team. We then head out to the second part of the assignment and write tree different games, Breakthrough, Fox and Hounds and Mega-Breakthrough.

2 Design

Our design philosophy was simplicity, we wanted to write code that was readable, understandable and worked well in a simple way without too much complexity. Therefore we started the assignment by sitting down in front of a piece of paper rather then a computer, many ideas came onto that paper and most were then crossed off. This process went for around 3 pieces of papers, after brainstorming many different ideas we finally thought we were ready to start coding. Even though we carefully designed the project on paper before we started, a lot of the ideas were not really code friendly or as simple as initially thought, so then we started designing again, the design process was rather iterative, both when designing part I from our initial plans and designing part II from part I. Part I changed quite a bit after we started on part II since then we realized that part I wasn't really working as intended.

3 Part I

As described in the design section, our main goal with this part was simplicity, and functionality. We decided on 4 classes: Board, Game, Piece and Player, later on we realized we didn't want the Player class and therefore we removed it. Game is the main class and it is an abstract class that is utilized to play the games in part II.

3.1 Piece

Piece is the building block of the engine. It is the only class that does not use any other class, but a class that every other class uses. A struct called Position is defined within this class and is built to keep track of generic x and y coordinates.

Private variables:

char symbol_ : stores the symbol for the Piece in play.

int owner_ : stores the owner of the Piece.

Position position_ : stores the position of the Piece.

std::vector<std::pair<int,int» posible_moves_ : stores all available moves for the Piece.

3.2 Board

Board is a relatively simple class. All it does is that is has a double dynamic array of Piece to keep track of the board that is in play. Every square on the board is an object, it has a position, owner and a symbol, if neither of the players own a Piece on a square its symbol is a dot (.) and its owner is no one.

Private variables:
int rows_ : stores the number of rows.
int columns_ : stores the number of columns.
Piece **board_ : keeps track of the board.

if columns_ is equal to 0 then it is set as the number of rows so we get a square board.

3.3 Game

Game is the fundamental class for playing games. It is an abstract class and therefore there can not be an instance of the class unless it is a pointer. Because no one wants to be able to play a non determined game and frankly that wouldn't make any sense. Game is the brain of everything and most of the calculations happen in this class. Games constructor initialized an empty Board for the game to play on.

Protected variables:
Board *board_ : a pointer to the Board class. int turn_ : turn counter. std::vector<Board*> timeline_ : a vector to store previous game states. char level_ : represents the current AI difficulty level.

4 Part II

Now begins the fun part. In this section we set out to write three different games and use part I as the base for those games. After the games were complete and playable we wrote 3 difficulty levels of AI to play the games and also a random AI. All of the games inherited the Game class which we discussed in the section about Part I. There wasn't the need of extreme amounts of code for each game due to the fact that Game does a lot of the heavy lifting. Everything that Game can do he does. That means that Game does everything that isn't game specific. The derived classes do everything that concerns rules for that specific game.

4.1 Breakthrough

We called our breakthrough class BT for simplicities sake. The constructor in this class calls the constructor for Game to make an empty Board then it initializes the board for breakthrough by adding pawns to the top two and bottom two rows and give every pawn a set of moves it is allowed to make on the board. Then a function called legal_moves returns a vector of legal moves to play.

4.2 Fox and Hounds

Our FaH class was implemented almost identically to BT but with a different set of rules and Piece types. The constructor first creates an empty board and then initializes the bottom row with a fox and the top row with hounds. We hardcoded the board size in Fox and Hounds to 8x8 even though it would work to use all kinds of board sizes. Its just that Fox and Hounds is meant to be played on a 8x8 board.

4.3 Mega-Breakthrough

Another game implemented in a class called MBT and was designed the same way as BT with the exception of the bottom row pawns and the top row pawns have a different rule set and can move 2 squares forward.

4.4 Main

main is where the magic happens. The main function found in main.cpp was built to play all of the games and follow simple commands. Available commands are.
List: lists available game, which are 3 in this particular assignment.
Game: used to select a game to play, must be done at the start.
Start: starts a new game of the game that is selected.
Legal: show all possible legal moves for the player whos turn it is.
Move: does the move that is called. e.g move a2 a3
Retract: undoes the last move moved, can be used to get back to the start of the game.
Display: displays the current game state.
Evaluate: Displays the evaluation value of the current game state.
Go: the AI moves for the player whos turn it is with the current difficulty level
Level: sets the difficulty level. e.g level easy. initially set to random
Debug: used only for debugging purposes
Size: sets the size of the board e.g size 5 5 sets a board to 5x5, do this before you select a game to get your preferred size of a board, initial board size is 8x8.

When you first run the program you must select a game otherwise the program crashes. This is due to the variable used to play the games is a pointer to an abstract class and therefore can't be used. The html team will find a way to never let the player play a game unless having first picked one from the list.

5 Results

In the end everything worked as intended, due to time constraints the quality of the program was not as high as initially aimed for. But functionality wise everything worked. A few more days would have boosted the program to a whole new level of awesomeness. But you don't get everything you wish for and the final product is still something we are proud of.