CPS2004 - Assignment
Object Oriented Programming

Daniel Magro
484497M

January 2018

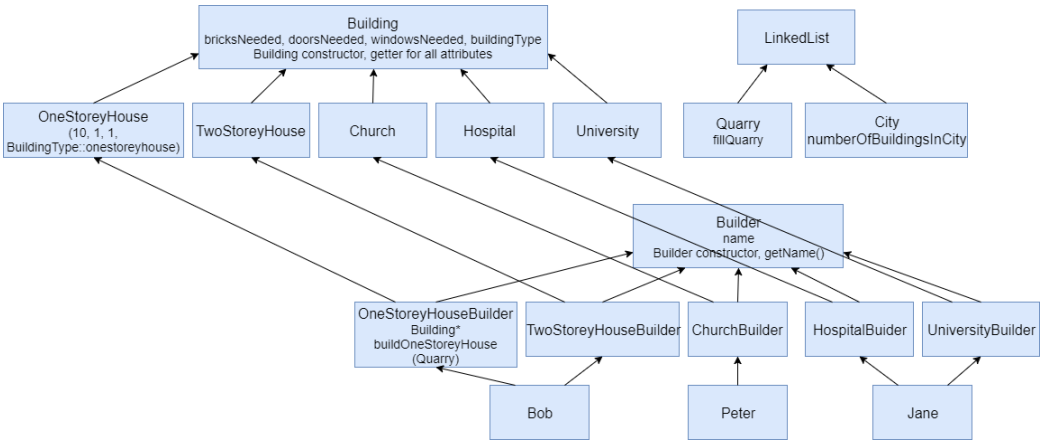
Contents

Task 1

C++ - LegOOPolis

1.1 UML Diagram

A complete UML Diagram for this program is located on the last page of this chapter, on page **10**



1.2 Code Explanation

When the program is launched, it first checks that the input file name was provided as an argument, and if not, immediately exits the program. After that, a Quarry and a City are declared.

The **Quarry** class creates a new **LinkedList** which stores objects of type **LegoPiece**. The Quarry Class also defines the **fillQuarry** method, which can be called to fill the Quarry with Lego Pieces of the type passed as an argument to the method, and in the quantity passed in the argument.
The **City** class creates a new **LinkedList** which stores **Pointers** to objects of type **Building**. The City class also defines the **int numberOfBuildingsInCity** method, which returns how many Buildings are in the city of the type passed as the argument.

After that is done, an instance of the **CSVReader** class is created, and the input file name specified in the arguments to the program is passed as the filename argument to the CSVReader constructor. The CSVReader class defines a method named **bool readLine** which reads the input file line by line, and returns whether the end of the file has been reached or not. If not, it reads the line, checks whether it is a blank line or a comment (eg: #quarry in which case it would move on to the next line), separates the line by delimiters (in this case commas), removes any trailing or leading whitespaces, and puts each token inside the vector.

In the main method, a while loop runs until the readLine method returns false. For each iteration it:
checks whether the vector is of size 2, if it isn't the program warns the user of incorrect input file format an exits.
If it is, it initialises an integer named *amount* with the numeric value of the second element of the vector.
Then, an if-else if tree checks the first element of the vector to determine what word it is to do the related function accordingly.
The words that are recognised by the program are: bricks, doors, windows (relating to the quarry) and 1storeyhouse, 2storeyhouse, church, hospital and university (relating to the buildings). For any other input other than those words, The program notifies the user of incorrect formatting of the input file and exits.
If the current line of the input file reads **bricks**, **doors** or **windows**, the main method does as follows:
First checks if the amount is between 1 and 100, if it is less than 1, then amount is set to 1, and if it is more than 100 it is set to 100. If the inputted amount is changed by the program, then the user is notified. When the amount has been validated, the quarry's **fillQuarry** method is called, and the PieceType is passed depending on what is inside the first element of the vector (bricks vs doors vs windows) along with the amount to be added to the quarry.

The **fillQuarry** method contains a for loop which iterates as many times as the *amount* argument specifies. During each iteration:
A **LegoPiece** object is created on the stack by calling the LegoPiece constructor, and passing the PieceType as passed to the fillQuarry method in the argument.
The newly created object is added to the end of the Linked List (quarry) using the **insertAtEnd** method which is specified in the **LinkedList** generic class. The legoPiece object is passed as the data which is to be stored inside the Linked List.

Else, If the current line of the input file reads **1storeyhouse**, **2storeyhouse**, **church**, **hospital** or **university**, the main method does as follows:
If the amount requested is >0:
First it adds the amount of buildings requested to be built to the *numberOfRequestedBuildings* integer, which keeps track of how many buildings were requested to be built by the user in the input file, this is important for the description of the city which is shown at the end of the program.
Next an instance of the *Contractor* responsible for building the current type of Building is created. Bob is responsible for 1 Storey Houses and 2 Storey Houses, Peter for Churches and Jane for Hospitals and Universities. So if the first element of the vector indicates a Hospital is to be built, an instance of Jane is created and named *contractor*.
Then, a *for loop* iterates as many times as specified in the amount integer of the following:
The contractor's build method is called (eg.3 contractor.buildHospital(&quarry)) with the address to the quarry object as an argument, from which Lego Pieces will be deducted. The return of the build method, say buildHospital, is stored inside the **City** Linked List.
Since the build*** methods will not always return a pointer to a building, since the quarry may not have enough lego pieces for a certain building, the for loop is surrounded by a *try catch block* with catches the **NotEnoughPiecesException**, which allows the program to proceed to build the next building.

At the end of the program, a Description of the City is printed to the user. First the Number of Requested Buildings is printed, Then the Number of Successfully built buildings, and then the percentage of buildings built.
Then the number of each building type built is printed. This is accomplished using the **numberOfBuildingsInCity** method, defined in the *City* class. The numberOfBuildingsInCity method takes the type of building, *BuildingType*, the user would like to know the number of as a parameter. Then it

iterates through the entire linked list, and gets the node at the current index using the LinkedList's **getNodeAtIndex** method, checking the Building-Type of the Node, and comparing that to the BuildingType passed as a parameter, if it is a match the counter is incremented.

1.3 Short Description of Every Class

Main is the launcher for the program, it calls the CSVReader to read the input file, creates the quarry and the city, and runs the relevant code depending on each line of the input file. Finally it displays statistics about the city to the user.

CSVReader takes care of reading input files from Disk. Its constructor takes a file name as an argument. It defines the *readLine* method which returns whether the EOF has been reached, and stores the current line in a vector, after tokenising and removing white space.

LinkedList is a generic Linked List implementation, which for each Node stores data and the address of the next node in the list. The class also defines the following methods: **insertAtEnd(data)**, **int getSize**, **bool NodeExists(data)**, **int NodeIndex(data)**, **deletNodeAtIndex(index)**, **deleteLinkedList** and **Node getNodeAtIndex(index)**.

Quarry is a class which implements the Linked List for objects of type *LegoPiece*, and defines the method **fillQuarry(pieceType, amount)**.

City is a class which implements the Linked List for *Pointers* to objects of tye *Building*, and defines the method **int numberOfBuildingInCity(buildingType)**.

LegoPiece is the class from which legoPiece objects can be instantiated. It also contains an enum named *PieceType* which specifies what type Lego Pieces can be, i.e. bricks, doors and windows. Each LegoPiece has a private variable, of type PieceType, called pieceType which defines what type of piece the lego piece is. It also defines an operator override for the == operator, which is used when comparing two lego pieces together to check if they are of the same type, this overload is used by the LinkedList when searching for a node's existence or index, by the build*** methods.

Building is the superclass from which **OneStoreyHouse**, **TwoStoreyHouse**, **Church**, **Hospital** and **University** all inherit. It also includes an enum called **BuildingType** which includes all the types of buildings that can be constructed. The class defines 4 variables, *int bricksNeeded*, *int doorsNeeded*, *int windowsNeeded* and *BuildingType buildingType*. The class also defines a getter method for each of the private variables.

OneStoreyHouse, **TwoStoreyHouse**, **Church**, **Hospital** and **University** all inherit from **Building**. When an instance of, say Hospital, is created by calling its constructor, which takes no parameters, the Building constructor is called and the constant values for the Hospital are passed - i.e. 60 bricksNeeded, 10 doorsNeeded, 15 windowsNeeded and hospital bulding-Type. The values for the 4 variables are coded as private static constants, such that they cannot be tampered with during runtime.

Builder is the superclass from which **OneStoreyHouseBuilder**, **TwoStoreyHouseBuilder**, **ChurchBuilder**, **HospitalBuilder** and **UniversityBuilder** all inherit. It includes an enum called **ContractorName**, which includes the names of all the contractors. Each instance of builder has a private variable storing the *contractorName* and defines a getter function for the name.

OneStoreyHouseBuilder, **TwoStoreyHouseBuilder**, **ChurchBuilder**, **HospitalBuilder** and **UniversityBuilder** all inherit from **Builder**. The inheritance done here is **virtual**, since a Contractor can inherit from multiple Builder classes (e.g. Jane inherits from HospitalBuilder and University-Builder) both of which inherit from *Builder*. Because of this, the **Diamond Problem** occurs, hence why the **virtual** keyword is required. Each class defines a build method for that type of building, for example *HospitalBuilder* defines the **Building* buildHospital(Quarry*)** method, which takes a pointer to the quarry as an argument, and returns a **Pointer** to the **Building**.
The *buildHospital* (and similarly, every other build*** method) does the following:
A new building object is created on the heap, whose pointer is stored.
Three LegoPieces, a brick, a door and a window are created on the stack.
Three consecutive for loops remove the required bricks, doors and windows for the building, respectively.
The first for loop iterates for as many times as the building object's *getBricksNeeded* method returns. In each iteration:
It is checked whether the quarry contains a brick. Then the index of that brick is searched for. Finally the node storing that brick (node with the found index) is deleted from the LinkedList/quarry. If no more bricks are found in the quarry, the user is notified, the building is deleted from memory, and the **NotEnoughPiecesException** exception is thrown, and caught by the main method, from where the build method is called.
The same process occurs for doors and windows. If the end of the method is successfully reached, the pointer to the Building is returned by the method.

NotEnoughPiecesException is a custom exception which is thrown when not enough pieces are found in the quarry to finalise the construction of a particular building by a build*** Method.

Bob, **Peter** and **Jane** are *Contractors*. Each Contractor is capable of building certain types of buildings. For Example, Jane inherits from *HospitalBuilder* and *UniversityBuilder*. When an instance of Jane is created, the class's constructor calls the constructors of *Builder*, *HospitalBuilder* and *UniversityBuilder*. Thus Jane can access the *buildHospital* and *buildUniversity* methods.

1.4 Testing

The program was tried with a wide range of inputs.
First, the program was run with a few sets of inputs which conformed to the input specification, for example providing the exact number of bricks in the quarry for the requested building, this always worked correctly, and the right buildings were built.
When commands which fill the quarry were given with amounts outside of the 1-100 range, the program caps the amount (1 if ¡1, 100 if ¿100)
Next, inputs where not enough bricks were provided for the requested buildings were tested, and each time, the correct result was obtained, where the buildings for which sufficient bricks existed were built, and exceptions were thrown and displayed to the user for the rest which did not have enough lego pieces.
When completely irrelevant or unspecified inputs were given to the input file, the program either carries on to the next line of the input file, or displays an error to the user and gracefully exits.
When the order of the commands or statements in the input file is not observed as per the specification, the program still runs, however will execute each command in the order it is inputted, thus if the buildings are listed first, and the quarry last, then the program will not find enough lego pieces, however will still fill the quarry when those commands are reached.

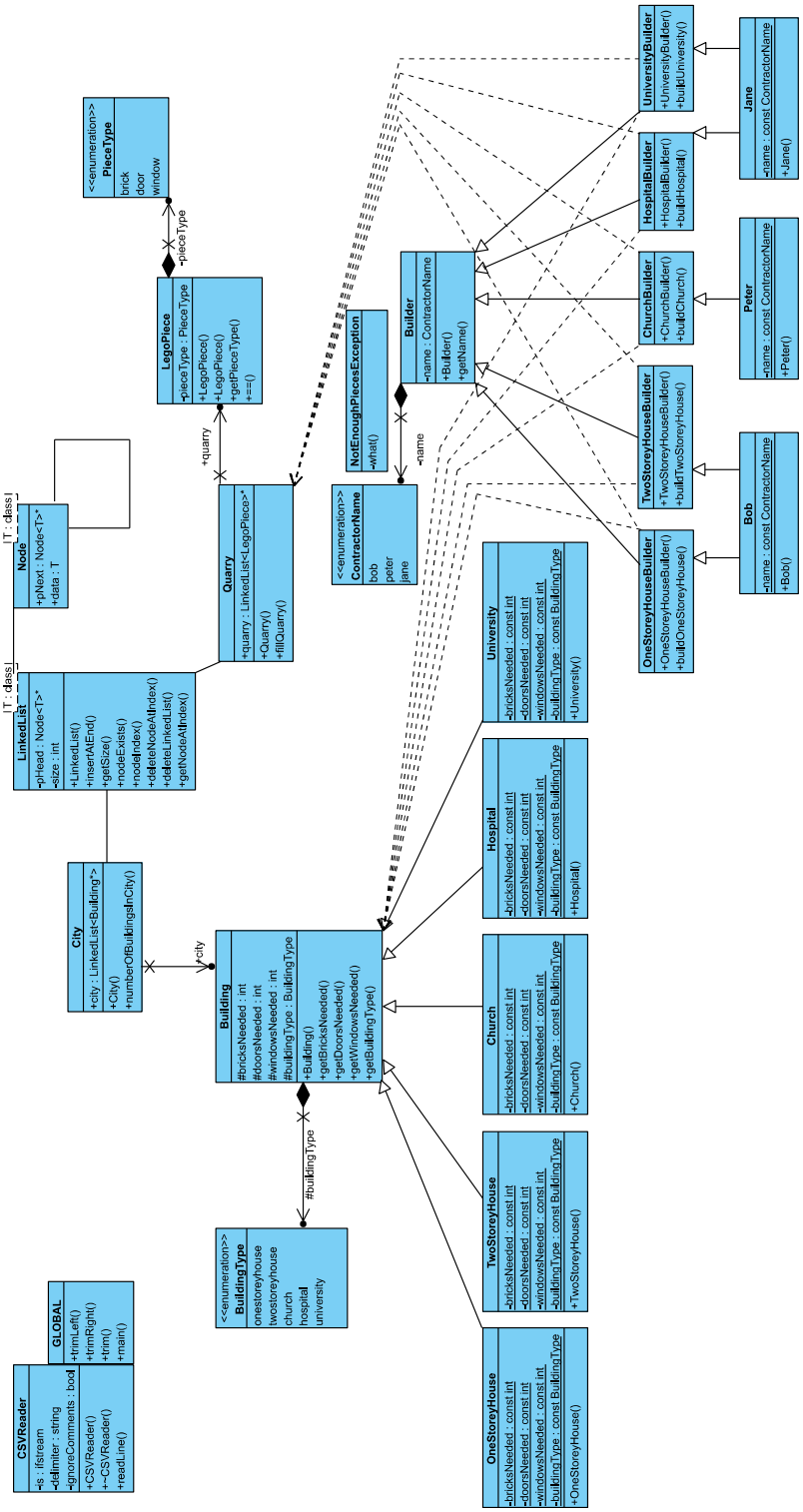
1.5 Evaluation of Solution

The code for this solution compile successfully without any errors or warnings. Every command which can throw an exception should be wrapped inside a try catch block, such that it is difficult to crash the program. Even on forced exit due to invalid input or whatever reason, the *garbageCollection* method is called which frees any allocated memory, i.e. the Linked Lists and every node inside the List.
The output of the program always matches the intended output, from the test cases considered, or for any input that I could imagine.

1.6 Discussion

a) Discuss how multiple inheritance can reduce code duplication.
Multiple Inheritance can help reduce code duplication as, if for example 3 methods are defined in 3 separate classes, another class can inherit from these 3 classes and thus have access to these 3 methods, without the need to copy-/rewrite the code inside the subclass. For example, in *Bob*, *buildOneStorey-House* and *buildTwoStoreyHouse* are inherited from *OneStoreyHouseBuilder* and *TwoStoreyHouseBuilder* respectively, avoiding the need to have a copy of these methods inside the *Bob* class.

b) Discuss how memory is being handled and how you avoided memory issues in your program.
In this solution, when objects are created, they were declared on the stack as frequently as possible (i.e. the *new* keyword was not used in instantiation). This way, when the end of scope is reached the memory is automatically freed.
In other cases, where objects were created on the heap, such as for nodes in the Linked List, or new buildings, whose pointers were stored in nodes of the Linked List, it was made sure that in every possible exit point of the program, be it at an intentional exit (at the end of execution) or in a non-recoverable error (such as inside a catch block), the *garbageCollector* method is called. The garbageCollector method first deletes the data inside each node, then deletes each node inside the linked list, then the pointer to the linked lists.
Alternatively, *smart pointers* could have been used to make sure that the allocated memory is simply freed automatically.

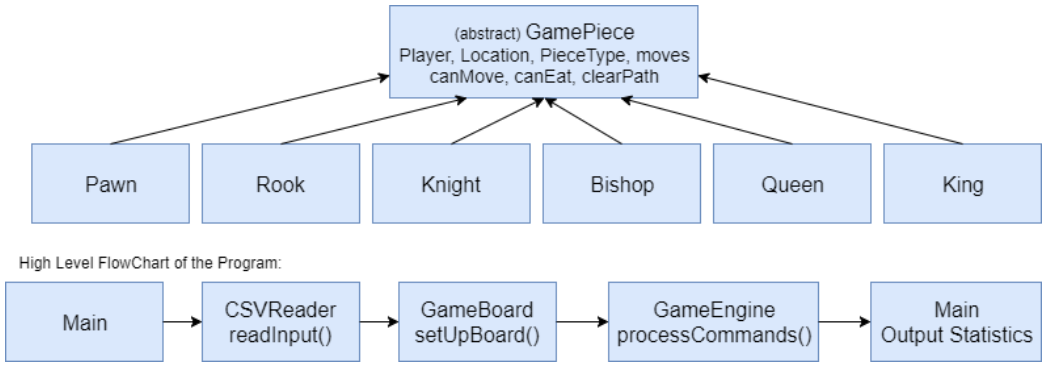

CSVReader
-is : ifstream
-delimiter : string
-ignoreComments : bool
+CSVReader()
+~CSVReader()
+readLine()
GLOBAL
+trimLeft()
+trimRight()
+trim()
+main()
<<enumeration>>
BuildingType
onestoreyhouse
twostoreyhouse
church
hospital
university
#buildingType
Building
#bricksNeeded : int
#doorsNeeded : int
#windowsNeeded : int
#buildingType : BuildingType
+Building()
+getBricksNeeded()
+getDoorsNeeded()
+getWindowsNeeded()
+getBuildingType()
*city
City
-city : LinkedList<Building*>
+City()
+numberOfBuildingsInCity()
OneStoreyHouse
TwoStoreyHouse
Church
Hospital
University
-bricksNeeded : const int
-doorsNeeded : const int
-windowsNeeded : const int
-buildingType : const BuildingType
+OneStoreyHouse()
+TwoStoreyHouse()
+Church()
+Hospital()
+University()
LinkedList
-pHead : Node<T>*
-size : int
+LinkedList()
+insertAtEnd()
+getSize()
+nodeExists()
+nodeIndex()
+deleteNodeAtIndex()
+deleteLinkedList()
+getNodeAtIndex()
T : class
Node
+pNext : Node<T>*
+data : T
Quarry
+quarry : LinkedList<LegoPiece>*
+Quarry()
+fillQuarry()
*quarry
LegoPiece
-pieceType : PieceType
+LegoPiece()
+LegoPiece()
+getPieceType()
+==()
*pieceType
<<enumeration>>
PieceType
brick
door
window
NotEnoughPiecesException
-what()
<<enumeration>>
ContractorName
bob
peter
jane
-name
Builder
-name : ContractorName
+Builder()
+getName()
UniversityBuilder
+UniversityBuilder()
+buildUniversity()
HospitalBuilder
+HospitalBuilder()
+buildHospital()
ChurchBuilder
+ChurchBuilder()
+buildChurch()
TwoStoreyHouseBuilder
+TwoStoreyHouseBuilder()
+buildTwoStoreyHouse()
OneStoreyHouseBuilder
+OneStoreyHouseBuilder()
+buildOneStoreyHouse()
Jane
-name : const ContractorName
+Jane()
Peter
-name : const ContractorName
+Peter()
Bob
-name : const ContractorName
+Bob()

Task 2

Java - KasparOOP Chess

2.1 UML Diagram

A complete UML Diagram for this program is located on the last page of this chapter, on page **17**



2.2 Code Explanation

The code for this solution is commented in-line in detail, thus not explained again in this document.

2.3 Short Description of Every Class

Main is the Launcher for the program. When run, it creates a new List of **Command**s, and initialises it using the return of the **CSVReader**'s *readInput* method, passing the argument of the program (name of input file) with it.
Then the Game Board is declared as an 8x8 2D array of **GamePiece**s and initialised using **GameBoard**'s *setUpBoard* method.
Then **GameEngine's** *processCommands* method is called, passing the list of Commands and the 2D array of GamePieces - board, as parameters.
Finally, the main method outputs the End of Game Statistics. The number of applied commands is printed, which is returned by the *processCommands* method. The number of Invalid/Skipped commands is stored inside the **InvalidCommandException** and printed using its getter method (*getInvalidCommands*). Each Player's score is found using **GameEngine**'s *calculateScore* method, and the player with the higher score is printed as the winner. Then each Player's most used piece is printed using **GameEngine**'s *findMostUsedPiece* method. Lastly, The Last Chess Board State is printed using **GameBoard**'s *displayBoard* method.

Command defines what attributes each Command should have, and also defines getters and overrides some functions. Each Command should have a **Player**, **CommandType**, **Location source** and **Location destination**. A constructor exists for these 4 attributes, as well as a getter for each attribute. A *toString* method override is also defined, such that a Command can be printed in the same format that it is entered into the input file.

Player is an enum which defines that the Player can either be **B** or **W**.

CommandType is an enum which defines that the CommandType can either be **MOVE** or **EAT**.

Location defines what attributes each Location should have. Each Location stores an **int x** and **int y**. A getter for each of these co ordinates exists. A *toString* method override is also defined for Location, such that the Location can be printed in the same format that it is entered in the input file.

CSVReader defines a collection of methods which accomplish the parsing of the input text file into a List of Commands. The *readInput* method takes the name of a file as a parameter, and returns a List of Commands. If

any line in the input text file is of the incorrect format or has invalid values, the **InvalidCommandException** is thrown. This exception is caught and the invalid command is printed to the user. The program carries on to the next line in the input file.

InvalidCommandException is an Exception which is thrown every time a command cannot be parsed or when a command cannot be executed. This exception has a static int, *invalidCommands* which counts how many times it was thrown. This can be retrieved using *getInvalidCommands*, and is used to determine how many commands were invalid or skipped.

GamePiece is an **abstract** class which defines all the attributes and methods all GamePiece subclasses must have. Each subclass object must have a **Player**, **Location**, **PieceType** and **moves** (number of moves) attribute. Furthermore, each subclass must define the *canMove*, *canEat* and *clearPath* methods. A final method is implemented in GamePiece, named *incrementMoves*, this method was made final since its implementation is the same for all subclasses. A getter was implemented for all the attributes since they are private variables. The *Location* attribute is set when a Piece is initialised, however is not updated when a piece moves, this is because that data is never used, and can be removed, however was left in for completeness sake.

PieceType is an enum which defines that the PieceType can either be **PAWN**, **ROOK**, **KNIGHT**, **BISHOP**, **QUEEN** or **KING**.

PAWN, **ROOK**, **KNIGHT**, **BISHOP**, **QUEEN** and **KING** are all subclasses of **GamePiece**, i.e. they inherit from/extend **GamePiece**. Each of these subclasses contain an implementation of the methods specified in **GamePiece**. This way, when **GameEngine** wants to check if a game piece at, say, b2 can move to b4, it will use the canMove method of the piece currently at b2 to determine whether it can legally move to b4.

GameBoard defines the *setUpBoard* and *displayBoard* methods. The *setUpBoard* method takes the 2D array of *GamePiece*s which will act as the Game Board, as a parameter, along with two ArrayLists of *GamePiece*s, which will store all the Game Pieces of each player, such that the statistics of the game piece are not lost when a piece is removed from play (i.e. is eaten). The method initialises certain locations on the chess board (2D array) with the chess piece that belongs there. This is done using **Dynamic Binding**, for example **GamePiece p = new Pawn(Player.W, new Location(i,1), PieceType.PAWN)**. (This is explained further in Section 2.6b).

The *displayBoard* method takes the 2D array of **GamePiece**s as a parameter, and traverses it to display the state of the game board. The game pieces are represented by their unicode symbols.

GameEngine defines the *processCommands*, *findMostUsedPiece* and *calculateScore* methods. The *processCommands* method takes the List of Commands and the Game Board as parameters. It loops through every command in the list, and for each command, and first checks whether it is a MOVE or an EAT commands. Then a set of conditions check whether the current command is legal or not, based upon a set of universal conditions, and some conditions specific to the game piece being referred to (eg. canMove and canEat of the GamePiece being moved). If it is found that the command is invalid, the **InvalidCommandException** is thrown, and the counter of invalidCommands is incremented. If a command is successfully applied, then the counter of appliedCommands is incremented, and returned at the end of the method.
The *calculateScore* method traverses the Game Board, and depending on a piece's type, and which player owns it, adds the respective score to that Player's point tally. The current implementation calculates both scores and the winner is chosen depending on who obtained a higher score. A more efficient implementation would be to add the white pieces' scores, and deduct the black pieces' scores, and if the result is positive white is the winner, whereas if it is negative the black player wins.

The **Observer Design Pattern** could have been implemented in the **GameEngine**'s *processCommands* method so that an observer is updated each time a command is successfully executed, and thus keeps the count of how many commands were executed.
Another application of this Design Pattern could have been with a piece's move counter, where each time a piece is moved its observer is updated.

2.4 Testing

The CSVReader for this program was tested with multiple different commands, of all sorts, however all of the invalid commands were always caught. Because of the way the CSVReader is implemented, with checking for enum values for the player and command type, as well as using the reg-ex matcher for the location, and checking the length of the command and number of values, it should be impossible to pass an invalid input without an exception

being thrown.
The Game Engine of the program was tested extensively, all the pieces were tried with multiple different commands: Moving pieces of both players back and forth, in places they shouldn't be able to move to, in wrong directions, onto occupied squares, eating pieces of the same colour, however all the invalid commands were always caught.
The scoring system was also tested extensively, with each test input file, the score was manually calculated, and for every test case the output was as expected.

2.5 Evaluation of Solution

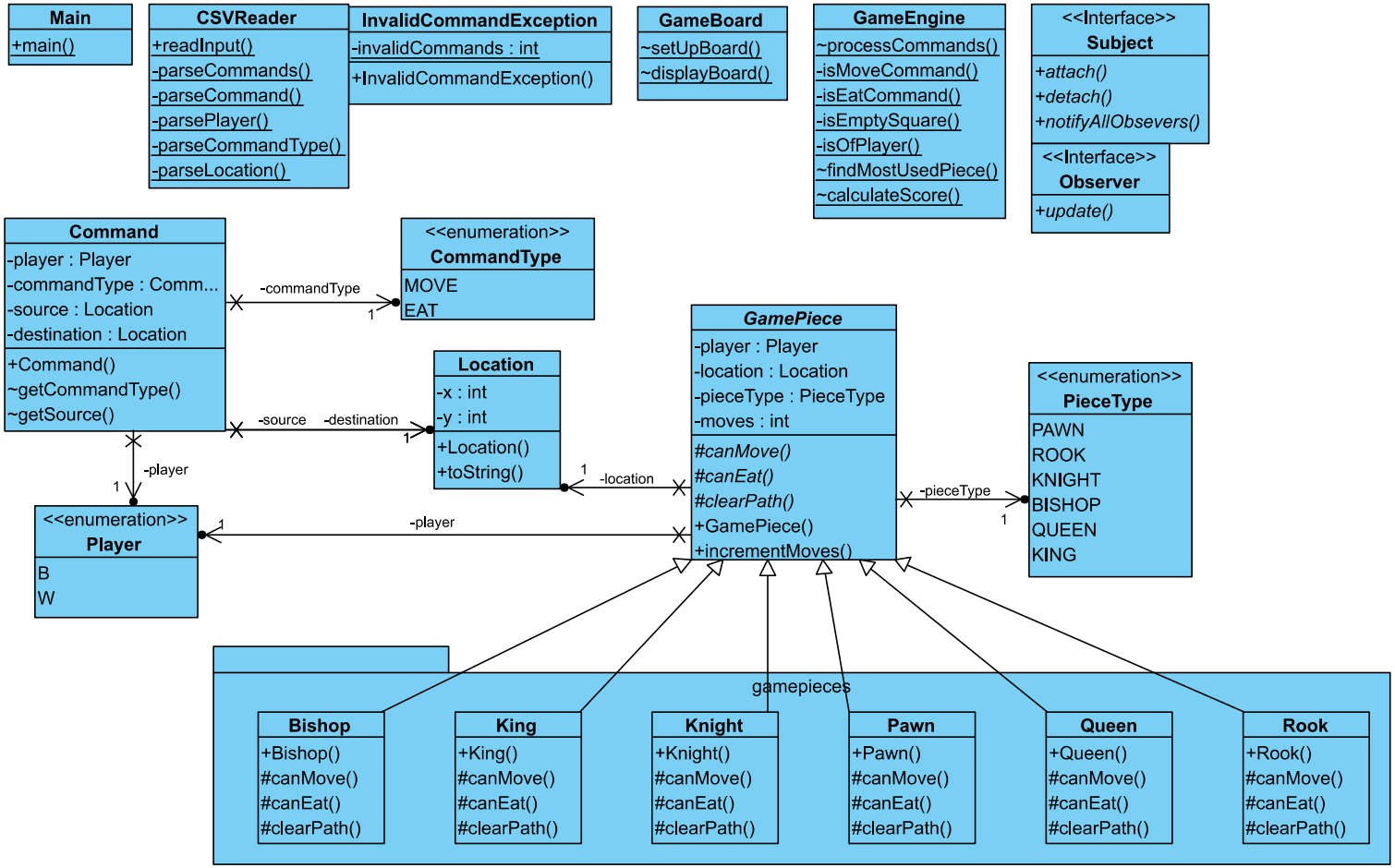
The solution for this task was tested extensively, and no bugs were found during testing, which suggests that the solution is a very good one. The solution can also be improved/adapted since the code is very 'modular'. The displaying of the chess board was made to look more graphical by using unicode symbols to represent the chess pieces instead of Letters.
Special moves such as *en passant*, *castling* and *promotion* were not implemented due to them not being in the specification, and due to time constraints. Another Limitation of this solution is that certain moves of the King which would be illegal in standard Chess are not checked for, i.e. the King not being allowed to move into the line of sight of an enemy's piece.

2.6 Discussion

a) Discuss the design choices you made and why.
The Main class is what ties all the classes of this program together. The Command class, along with the Player enum, CommandType enum and Location class define a set of attributes each command should have, as well as a set of methods applicable for all command. The same applies for the Location class. This allows for a small change to any of these classes to be made easily, without disrupting other classes, for example a new CommandType can be simply added to the CommandType enum. The implementation of the new command can then be added to the CSVReader and the GameEngine using the same interface the MOVE and the EAT commands are using. The CSVReader class defines a set of methods, each of which specifically deals with the parsing of a particular value of the commands in the CSV input file.

The GameBoard class defines methods which have to do exclusively with setting up the board and displaying the board. The GameEngine class processes and executes the inputted commands, which are retrieved by the CSVReader class, if they are valid they are executed, if not an exception is thrown. The conditions which determine if a command is valid or not are all methods which return booleans, such that they can be modified or optimised as required.
All the different types of GamePieces inherit from the abstract GamePiece class, such that they can all have their implementation of the canMove and canEat method (and clearPath which is employed by those 2 methods). This is discussed in further detail in Section 2.6b.

b) Discuss how polymorphism was used as an abstraction tool.
GamePiece is an abstract class which defines what attributes and methods must be present in all sub-classes. All the Game Pieces (i.e. Pawn, Rook, Knight, Bishop, Queen and King) inherit from the GamePiece class. When the Chess Board is being set up, Dynamic Binding is used, such that a GamePiece is declared but is of type, say, Pawn. A very big advantage of instantiating all the GamePieces as such, is that the GameEngine can call a piece's *canMove* or *canEat* method, without having to determine what type of Game Piece is being moved.


Main
+main()
CSVReader
+readInput()
-parseCommands()
-parseCommand()
-parsePlayer()
-parseCommandType()
-parseLocation()
InvalidCommandException
-invalidCommands : int
+InvalidCommandException()
GameBoard
~setUpBoard()
~displayBoard()
GameEngine
~processCommands()
-isMoveCommand()
-isEatCommand()
-isEmptySquare()
-isOfPlayer()
~findMostUsedPiece()
~calculateScore()
<<Interface>>
Subject
+attach()
+detach()
+notifyAllObsevers()
<<Interface>>
Observer
+update()
Command
-player : Player
-commandType : Comm...
-source : Location
-destination : Location
+Command()
~getCommandType()
~getSource()
-commandType
1
<<enumeration>>
CommandType
MOVE
EAT
-source
-destination
1
Location
-x : int
-y : int
+Location()
+toString()
-player
1
<<enumeration>>
Player
B
W
1
-player
GamePiece
-player : Player
-location : Location
-pieceType : PieceType
-moves : int
#canMove()
#canEat()
#clearPath()
+GamePiece()
+incrementMoves()
1
-location
-pieceType
1
<<enumeration>>
PieceType
PAWN
ROOK
KNIGHT
BISHOP
QUEEN
KING
gamepieces
Bishop
+Bishop()
#canMove()
#canEat()
#clearPath()
King
+King()
#canMove()
#canEat()
#clearPath()
Knight
+Knight()
#canMove()
#canEat()
#clearPath()
Pawn
+Pawn()
#canMove()
#canEat()
#clearPath()
Queen
+Queen()
#canMove()
#canEat()
#clearPath()
Rook
+Rook()
#canMove()
#canEat()
#clearPath()