Department of Computer Science
Faculty of ICT
University of Malta

Mr Aldrin Seychell
aldrin.seychell@um.edu.mt
Version 3.1 (November 8, 2017)

CPS2004 Object-Oriented Programming

Study-Unit Assignment

Wednesday, 8th November 2017

Necessary Preamble: This document describes the assignment for study-unit *CPS2004: Object Oriented Programming*. This assignment is worth **100%** of the total, final mark for this unit. You are expected to allocate approximately 70 hours to complete the assignment. You will be required to demonstrate your solutions. The deadline for this assignment is **Friday, 12th of January, 2018 at noon**. Late submissions will **not** be accepted. Questions regarding the assignment should **only** be posted in the Assignment VLE forum (and not via personal correspondence with the lecturer or tutors of this study-unit).

Unless otherwise stated in the task description, this is an individual assignment. Under **no** circumstances are you allowed to share the design and/or code of your implementation. You may **not** copy code from internet sources, you will be heavily penalized if you do so! The Department of Computer Science takes a very serious view on plagiarism. For more details refer to plagiarism section of the Faculty of ICT website[1].

Important Note

The main objective of this assignment is to demonstrate you understood the OO concepts presented in class. Submitting a perfectly working solution, implemented in a non-OO manner will result in a failing grade.

1 Deliverables

You are to upload all of your code and documentation on the VLE website (submission via email will not be accepted). The following deliverables are expected by the specified deadline. Failure to submit any of these artefacts in the required format will result in your assignment not being graded. Only two files are required for electronic submission. Replace `NAME` and `SURNAME` with your name and surname respectively (doh!). Replace `IDCARD` with your national id. card number (without brackets), e.g. `123401G`.

[1] `https://www.um.edu.mt/ict/Plagiarism`

- `201718_CPS2004_SURNAME_NAME_IDCARD_assignment_code.zip` - A zip file containing your assignment code. This needs to be uploaded to VLE. Each task should be located in a top level directory in the `.zip` file named `task1`, `task2`, and `task3`. It is your responsibility to make sure that this archive file has uploaded to VLE correctly (by downloading and testing it). Failure in opening the zip file will result in your assignment not being graded.
- `201718_CPS2004_SURNAME_NAME_IDCARD_assignment_doc.pdf` - The assignment documentation in `.pdf` format. The documentation has to be uploaded to VLE, together with your code. A hard-copy should be submitted to the secretary's office (Mr Kevin Cortis) at the Computer Science department by the stipulated deadline.

  The report should **not** be longer than 20 pages (including figures and references) and, for each task, should contain the following:
  - UML Diagram showing the classes' makeup and interactions
  - A textual description of the approach (highlighting any extras you implemented)
  - Test cases considered (and test results)
  - Critical evaluation and limitations of your solution
  - Answers to any questions asked in the task's description number

  This report should **not** include any code listings.
- **Signed copy of the plagiarism form -** This should be submitted to the secretary's office at the Department of Computer Science.

2 Technical Specification

Your code will be compiled and run on Linux (distribution: Ubuntu 16.04.1 LTS). C++ code will be compiled using the `g++` compiler (version 5.4.0) from the GNU Compiler Collection. This will be run using the command line `g++ -Wall -std=c++11 *.cpp`. Your Java submissions will be compiled and run using the Oracle Java Development Kit (version 8). Please make use of standard libraries only. While you are free to use any IDE, make sure that your code can be compiled (and run) from the command line. Failure to do so will have a severe impact on your grade. As a requirement, add a bash script (`compile.sh`) in each task directory to compile your task. Also make sure to add a bash script (`run.sh`) in each task directory to execute your task. In these bash scripts, include any command line arguments and test data files – if applicable. Each task should have a `Launcher` file (either `.java` or `.cpp`) which contains a `main(...)` method used to run your solutions. **Note that you should not have any absolute paths hardcoded in your programs/bash scripts.**

3 Tasks

This assignment consists of two programming tasks. The first task should be implemented in C++ and the second task in Java.

3.1 LegOOPolis

Lego [1] is a popular creative puzzle for young (and not so young!) people where small Lego pieces are used to construct more complex structures. Objects that can be built using Lego range from houses and cottages to cars and buses to churches and cathedrals, football stadiums and theaters. Building all these different types of objects requires various skills.

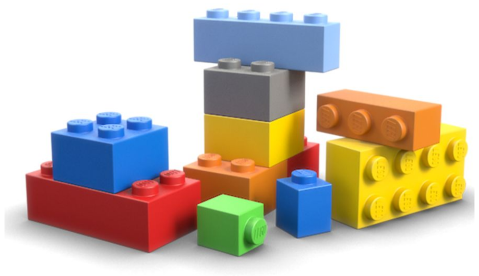

Figure 1: Lego pieces that can be combined to build more complex objects

3.1.1 Task Description

Build an Object-Oriented system that simulates a Lego construction site for a Lego village; LegOOPolis. The system should take as input a simulation specification file (see Figure 2) that describes:

1. A number of Lego constructions to be built to construct LegOOPolis
2. The number of lego pieces available; referred to as the Quarry (**Note:** lego pieces available can be either of: bricks, doors or windows).

The system should then attempt to build the specified buildings where possible and finally output to the console a description of the built city of LegOOPolis. This output should conform to the following format:

```
Number of requested Buildings: x
Number of Completed Buildings: y
Percentage completed: z%

List of Buildings completed:
  Houses: a
  Churches: b
```

3.1.2 The Implementation

In the implementation, you are to represent the following concepts:

1. LegoPiece

```
#quarry
bricks,quantity
doors,quantity
windows,quantity
#buildings
1storeyhouse,quantity
2storeyhouse,quantity
church,quantity
hospital,quantity
university,quantity
```

Figure 2: Format of the Input specification file. ***Note:*** `Quantity` *is a positive integer which can range between 1 and 100 (both inclusive).*

2. `Quarry` *(**Note:** Use the linked-list implementation from the tutorial sheet to store the LegoPieces in the* `Quarry`*)*
3. `Builder`
4. `Building`
5. `City`

The type of buildings that can be built in `LegOOPolis` and the required `LegoPieces` to build them are as follows:

1. `1-Storey House`(10 bricks, 1 door, 1 window)
2. `2-Storey House` (20 bricks, 1 door, 3 windows)
3. `Church` (40 bricks, 3 doors, 10 windows)
4. `Hospital` (60 bricks, 10 doors, 15 windows)
5. `University` (80 bricks, 10 doors, 20 windows)

Implement a *builder* class for each type of building. Each builder needs to have a *private* field for his/her name which will be passed as a parameter to the constructor and may be retrieved through a getter method. Each builder class should also have a method with signature similar to

```
Building *buildXXX(Quarry *quarry)
```

Such method will take the required pieces (if possible) from the given Quarry and construct a new building object of the respective type. If the Quarry does not contain some of the required pieces, the method should throw an exception (and consequently not build the requested building). For simplicity's sake, on such exceptions you are not required to return (to the Quarry) any Lego pieces already taken out of the Quarry. *It is a good exercise to think about possible solutions to this if it was a requirement.*

Lastly, you are to define contractors. Contractors are entities that specialize in certain type of building, and may have more than one specialization. Implement contractors, *Bob*, *Peter* and *Jane* as specified in Table 1. ***Hint:*** *these should be subclasses of the* `Builder` *class you previously created but you should use* C++ multiple inheritance *to avoid code duplication in these subclasses.*

	Bob	Peter	Jane
1 Storey House Builder	Yes	No	No
2 Storey House Builder	Yes	No	No
Church Builder	No	Yes	No
Hospital Builder	No	No	Yes
University Builder	No	No	Yes

Table 1: Contractors

3.1.3 `LegOOPolis` Task Summary

Implementation Summary:

1. Implement a class hierarchy as explained in Section 3.1.2.
2. Create a main file which
   a) reads input file (from a location which is passed as an argument to the executable).
   b) creates an instance for each of the contractors from Table 1.
   c) based on the input file uses contractors to build requested buildings, and add any successfully completed buildings to the city. ***Note:*** *the system should only halt after attempting to build all requested buildings. Should a building attempt fail due to insufficient* `LegoPieces` *in the* `Quarry`*, the System must gracefully proceed to the next building.*
   d) outputs a summary (as described in Section 3.1.2) of what was built in LegOOPolis to the console.

Final Notes:

1. Write well documented and clear code!
2. You are responsible of handling the memory efficiently. ***Hint:*** *avoid memory leaks by deleting objects that are not needed anymore such as used* `LegoPieces`*.*
3. In your report include:
   a) a discussion on how multiple inheritance can reduce code duplication.
   b) a discussion on how memory is being handled and how you avoided memory issues in your program.

3.2 KasparOOP Chess

Chess[2] is a strategy board game that was invented in the 6th century. Chess is a two player game, played on an 8x8 black and white chequered board. A player owns a set of chess pieces (black or white) which are initially set up as illustrated in Figure 3. Different chess pieces can move in different patterns and can eat pieces of the opponent when certain conditions are met. For a full set of basic rules, please refer to `http://www.instructables.com/id/Playing-Chess/`. To get familiar with the rules, nothing beats a few rounds of practice which can be done online[3]. For the scope of this assignment, do **not** consider special moves like castling and en passant.

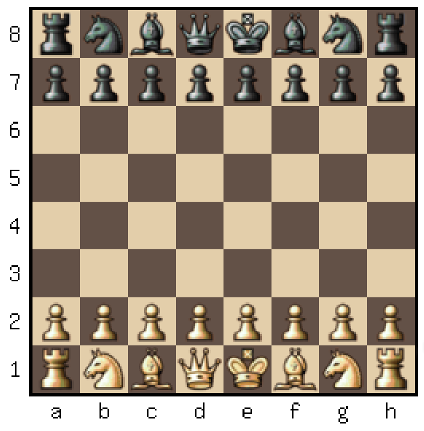

Figure 3: Chess board and the start positions

3.2.1 Task Description

You are requested to write a Java program; `KasparOOP` named after chess grandmaster Garry Kasparov[2], which can analyse an arbitrary set of chess movements.

On initialisation `KasparOOP` should have a chess board with pieces positioned in the chess start up arrangement (see Figure 3). The program should then read and parse a CSV file (from the location which is passed as an argument to the executable) containing a list of *move/eat commands* of the following format:

```
<PLAYER>,<COMMAND_TYPE>,<SOURCE>,<DESTINATION>
```

where

- `PLAYER` can be either `B` or `W`.
- `COMMAND_TYPE` can be either `MOVE` or `EAT`.
- `SOURCE` and `DESTINATION` are coordinates on the board, e.g. $c1$ or $d5$. Refer to Figure 4.

[2]`https://en.wikipedia.org/wiki/Garry_Kasparov`

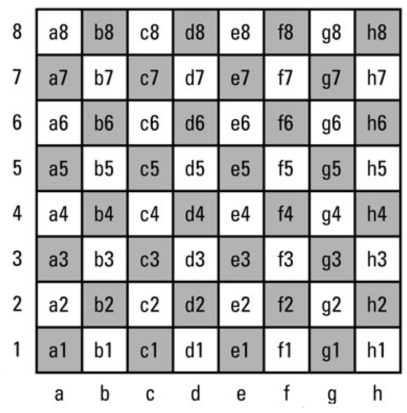

Figure 4: Chess board coordinates

The parsed commands are then applied only if they abide by the following constraints:

1. The command must have a valid format (as defined above).
2. A chess piece must exist at the source coordinate.
3. The chess piece at the source coordinate belongs to the player referred to in the command.
4. The source and destination coordinates in the command are within the board's boundary e.g. *a*9 is an invalid coordinate since it lies outside of the board.
5. An `EAT` command requires an opponent's chess piece at the destination coordinate.
6. The chess piece at source coordinate must be able to reach the destination coordinate by following the particular chess piece movement rules. For example, a `Bishop` can only move in diagonals.
7. A chess piece (unless it is a `Knight`) can move to its destination if the path from source to destination is empty. The `Knight` is allowed to *jump over* any other chess piece to reach its destination.

Note: *If a command is invalid, the system should gracefully proceed to the next command.*

After applying all valid commands, `KasparOOP` should determine the winning player by comparing the players' final scores using the scoring system from Table 2.

Finally, `KasparOOP` should output to the console the following information:

```
Number of applied commands: x
Number of invalid/skipped commands: y
Winning player: BLACK|WHITE
Most used chess piece: ROOK, 42

Last chess board state:
```

Piece	Symbol	Qty	Value	Total
Pawn	`P`	8	1	8
Bishop	`B`	2	3	6
Knight	`N`	2	3	6
Rook	`R`	2	5	10
Queen	`Q`	1	9	9
King	`K`	1	20	20

Table 2: Chess pieces, quantity per player and value based on the standard valuation

```
_ _ _ K _ _ _ _
_ P P P P P _ _
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
_ N _ _ _ _ _ _
_ _ _ _ _ _ _ _
_ _ P P _ _ _ _
R _ _ K _ _ _ _
```

Note: *Although the output of the chess board will not influence greatly the final mark, try to be creative in the way you output the chess board, e.g. lines and borders, ANSI colors, etc*

3.2.2 Final Notes

1. Write well documented and clear code!
2. You are expected to make use of *encapsulation*, *abstract classes*, *inheritance*, *polymorphism* and *exception handling*.
3. Employing the *observer* design pattern to track the players' scores and statistics will be rewarded with bonus marks.
4. Although it is important that the program outputs correct statistics; the design and effective usage of object-oriented principles (to obtain such output) is of higher importance!
5. In your report include:
   a) a discussion on the design choices you made and why.
   b) a discussion on how *polymorphism* was used as an abstraction tool.

4 Grading Criteria

The criteria described in Table 3, will be taken into consideration when grading your assignment.

Table 3: CPS2004 Assignment grading criteria.

Overall Considerations	
OO Concepts	Demonstrable and thorough understanding and application of OO concepts. You should make use of most of the concepts presented during lectures.
Documentation	Complete documentation of solutions including: design (in UML), technical approach, testing, critical evaluation and limitations. Must be properly presented (*e.g.* no loose papers, page numbers, table of contents, captions, references, *etc.*)
Coding Practices	Adherence to coding standards, consistency, readability, comments, (no) compiler warnings, code organization using packages/namespaces, *etc.*
Functionality	Completeness and adherence to tasks' specification. Also, correctness of solutions provided.
Quality and Robustness	Use of exceptions and proper exception handling. Proper (and demonstrable) testing of solutions. No unexpected program crashes.
Environment	Use of Linux/Unix setup for compiling and running programs.

Any documented extra (cool) functionality will result in bonus points. Note that **not** submitting one (or more) of the tasks, will severely affect your overall mark.

References

[1] Lego wikipedia article, `https://en.wikipedia.org/wiki/lego`, Last Accessed: November 4th, 2017.

[2] Chess wikipedia article, `https://en.wikipedia.org/wiki/chess`, Last Accessed: November 4th, 2017.

[3] Play Chess Against the Computer - Chess.com, `https://www.chess.com/play/computer`, Last Accessed: November 4th, 2017.

THE END