Department of Computer Science
Faculty of ICT
University of Malta

Dr Jean-Paul Ebejer
`jean.p.ebejer@um.edu.mt`
Version 2.0 (November 2, 2016)

CPS2004 Object-Oriented Programming

Study-Unit Assignment

Wednesday, 2nd November 2016

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, 13th of January, 2017 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 Notice

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

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

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`.

- **`201617_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.
- **`201617_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 (Ms Vanessa Marie Sammut Borg) 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 three programming tasks. The first task should be implemented in Java and the second task in C++. You may use C++ or Java for the third task – **please justify why you have chosen that specific language in the documentation**.

3.1 Tic-Tac-Toe

Tic-Tac-Toe (sometimes referred to as *Noughts and Crosses*) is a simple game which has been played countless times during lectures by many bored students across the globe. The game is played by two players on a 3x3 grid, and the first player who succeeds in placing three of their *marks* (typically an `X` or an `O`) in a horizontal, vertical, or diagonal row wins the game (see Figure 1). For more details about Tic-Tac-Toe rules and setup refer to [1]. To play for free online refer to [2].

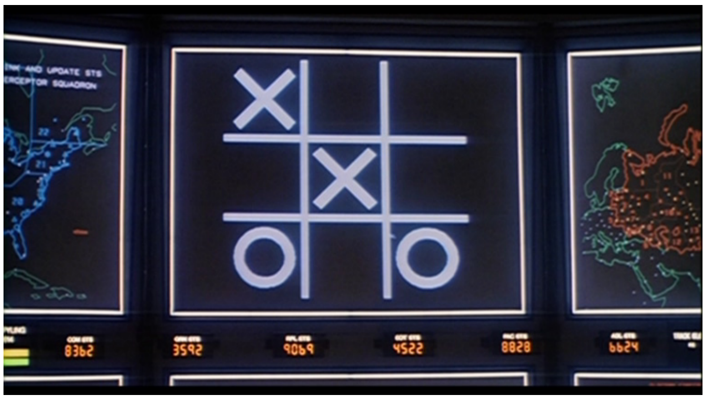


Figure 1: Tic-Tac-Toe in the classic movie *WarGames* (1983).

My (incomplete) Java Tic-Tac-Toe library is available in a github repository at `https://github.com/jp-uom/201617_CPS2004_OOP_Assignment`. You are to use this library to build a working Tic-Tac-Toe game. Some of the code is already implemented for you (e.g. `edu.um.cps2004.task1.engine.GameBoard`) and you need to integrate these classes with your implementation.

The source code of the library is available for your perusal. It is **forbidden** to change any of the code provided. Any modification requests should come via pull requests in github. For this reason, it is suggested you "watch" this repository (in case of updates). Documentation, in form of Javadocs, is available for every class, interface, and enum provided. It is strongly suggested you familiarize yourself with it.

Like real software projects, no guarantee is made on the quality and completeness of the software and documentation provided. Documentation may be outdated in places.

You are to form groups of **three** students (this is a task requirement). Note that this is still an individual assignment and communication in between the team should be kept to a minimum. You can only communicate through code (interfaces *etc.*). You should publish your group name, members, role (defined next), and captain in the assignment's forum in VLE.

Two of you will **each** implement an interface provided in the library, `edu.um.cps2004.task1.player.TTTRobot`, which defines the behaviour of a Tic-Tac-Toe robot. This is where the player's game-playing logic should be located, *e.g.* which is the next best move? Note that these two individuals should build their player's logic separately and should **not** communicate or have access to each other's code. Failure to do so will be considered plagiarism.

The other (third) person in the group must extend the abstract class `edu.um.cps2004.task1.engine.TTTEngine` (also provided) and implement the "engine" of the game using the two players provided by the other members of the team. The engine is where the game logic lies such as deciding on a starting player, giving turns to the players, determining whether a game has ended, *etc.*

It is suggested you use a single **private** github repository between your group (and add my github user `jp-uom` as a collaborator to the repository). Private repositories are available in github if you register for the student pack[2]. Note that if you are implementing the player logic you should only push the compiled `.class` file to the repository, so the other group member (who is also implementing the player logic) will not have access to your source code. **The two player-logic developers should never see and access each other's source code (binaries only). It is your responsibility to protect your source code from plagiarism.** Make sure to then supply your code in the code deliverable described earlier (together with the engine). A reminder: your submission should be a complete and working Tic-Tac-Toe which compiles and runs.

In the documentation include how you would extend the game engine to create a class tournament to find the champion between all the robots developed (of all the CPS2004 students).

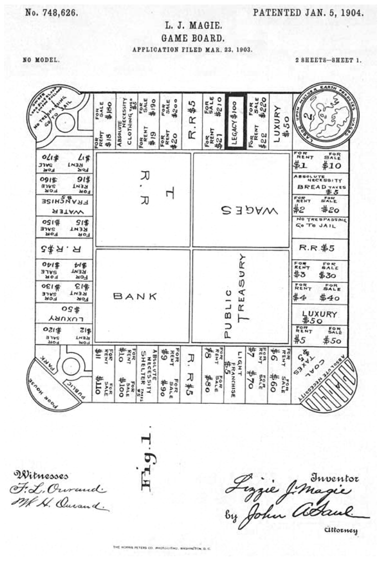
Design and deliver an interface for a player of a (named) game you are interested in. This is the equivalent of `edu.um.cps2004.task1.player.TTTRobot` for your chosen game.

[2]`https://education.github.com/pack/join`

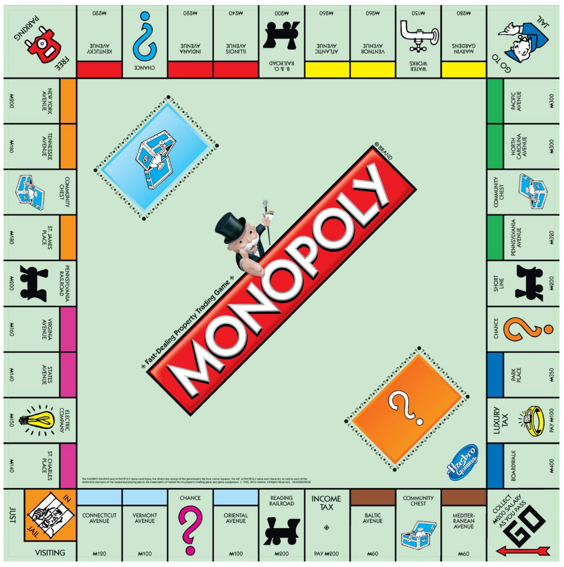
The scope of this task is to familiarize yourself with real-world OO and software development practices, *e.g.* using github, making use and learning to read other developers' code, and using OO principles to extend an existing library. These are all critical skills in today's software development industry.

3.2 MonOOPoly

Using C++, implement the game of *MonOOPoly* (a simplified version of *Monopoly*). For more details about *Monopoly* rules and setup refer to [3]. To play online for free refer to [4]. This is an individual (not group) task.



(a) The original board of *The Landlord's Game* (the *Monopoly* precursor).



(b) The classical *Monopoly* board.

Figure 2: Monopoly Boards

In *MonOOPoly*, the following simplifications apply:

1. Properties of the same colour have the same price, building costs, and mortgage. Properties can have the values of the 'middle' property in the real game.
2. There is only one pile of *Chance* and *Community Chest* cards called *Xorti*; the possible outcomes are:
    a) It is your birthday, collect a random amount from each other player (in the range €10 to €100)
    b) You parked your car in a lecturer's spot, and are therefore fined a random amount (in the range €50 to €150)
    c) MEPA has fined you for **each** of your buildings (a random amount in the range €40 to €180)

   d) Advance to a random position on the board

3. There are no hotels (only houses).
4. No need for a visualization (although that would be spectacular!) – a round may be described in text (*e.g.* Dog lands on Paceville).
5. No *Go to Jail* play (*Go to Jail* and *Jail* locations are just empty – upper-right and lower-left corners in Figure 2b).afterpage
6. No need for money bills, just keep a balance attribute for each player.

State any more assumptions you take during the game's development. Note that, even though desirable, the main aim of this exercise is not to have a fully-functional Monopoly-like game, but rather to apply OOP concepts to dissect the system in smaller working components which work and interact together.

3.3 Suffix Trie and Tree

A trie (pronounced *try*) is an ordered rooted tree where each edge is labelled with a letter from an alphabet (denoted Σ). A node in a trie has at most one outgoing edge labelled c, where $c \in \Sigma$. Keys (or collection of words) are spelled out along some path starting from the root and ending at leaves. A suffix of a string S is a substring that occurs at the end of S. A suffix trie is a trie which contains all suffixes of some text T, *i.e.* each path from root to leaf represents a suffix of T. For more information refer to [5], [6], and [7]. This is an individual (not group) task.

Figure 3 shows text *hahaa* as a suffix trie. Note the addition of a terminal character (#), a character not found in the text, to the end of each suffix. This is required so that no suffix is a prefix of another suffix. The nodes of a trie do **not** have any labels, but you can think of them as the string constructed from the root to the current node.

Using either C++ or Java, you are to build a suffix trie, that for any T, implements following operations:

- **`substring(`*S*`) -`** checks if a string S is a substring of T. Since every substring of T is the prefix of some suffix, start at the root and follow edges labelled with characters of S. A substring exists if the characters of S are exhausted and a labelled path exists which represents S. Returns a boolean.
- **`suffix(`*S*`) -`** checks if a string S is a suffix of T. Like `substring(`*S*`)` but make sure last edge is a terminal. Returns a boolean.
- **`count(`*S*`) -`** count how many times a substring S appears in T. Follow the characters of S (like in substring) in T. If S exists and ends at node n, count the number of leaves in the subtrie rooted at n. Returns an integer.

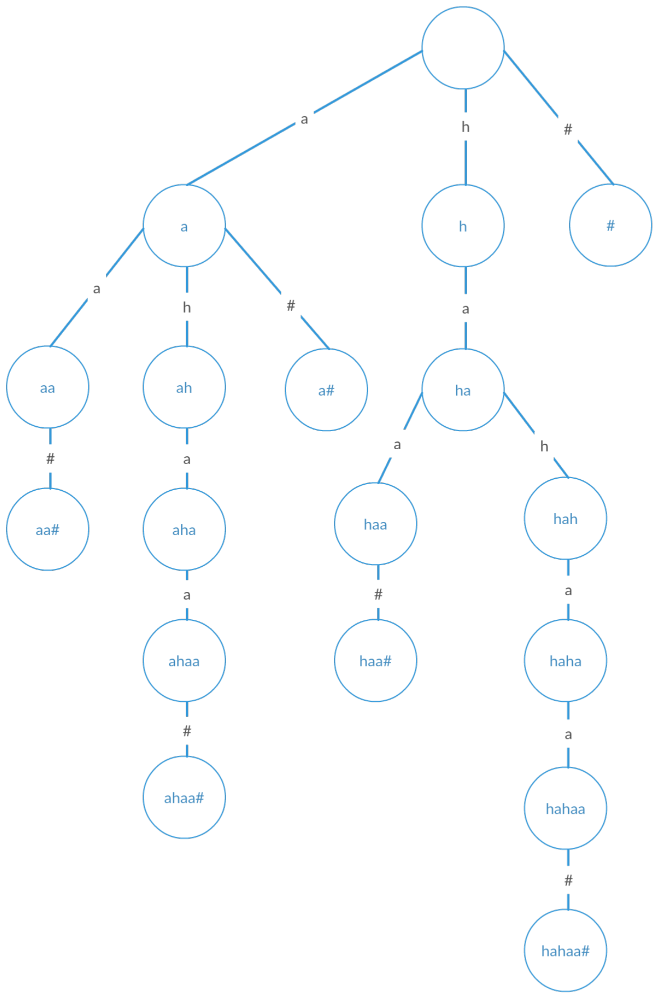


Figure 3: A suffix trie for text *hahaa*. Node labels are shown in blue for illustration purposes only (they are not part of the suffix trie).

- **`longestRepeat()`** - finds the longest repeated substring in T. Find the deepest node with more than one child. The repeated sequence is the string made up from the root to that node. Returns a string.
- **`longestSubstring(`S`)`** - finds the longest common substring between S and T. Returns an string.
- **`show()`** - Displays the trie.

Note that you will need a depth first search implementation to search the trie. Your program should take three command-line arguments. The first is a text file from where to read T, the second argument is one of the operations defined above and the third (optional) is S (not needed for `longestRepeat()` and `show()`).

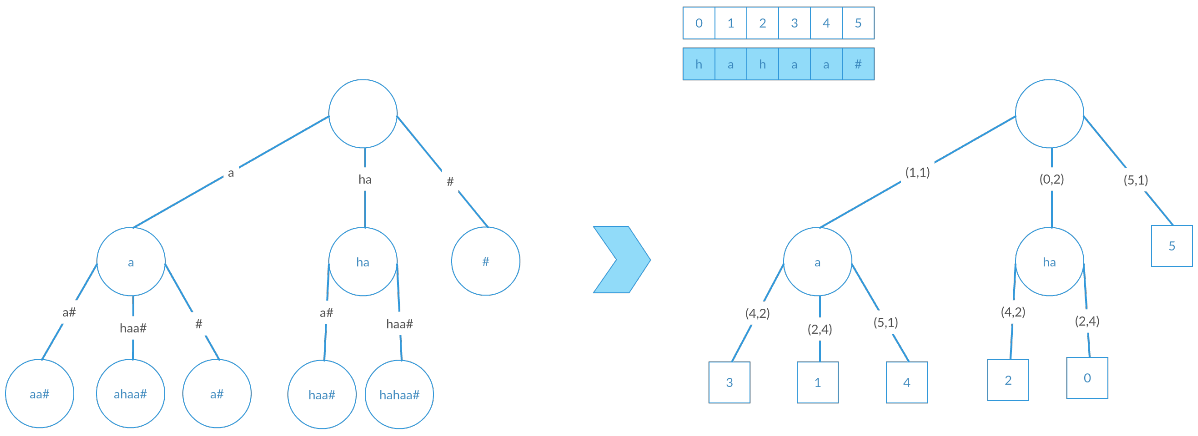


Figure 4: Optimizing suffix tree for text *hahaa*.

We can optimize this trie structure with some simple modifications. Build an optimized (second) version of the data structure, which coalesces non-branching paths into a single edge with a string label (see Figure 4 for an example of this). This has the effect of reducing nodes and edges, and effectively transforms the trie into a tree. A second optimization is to store the full text (T) alongside the tree and use the *(offset, length)* tuple to refer to the suffix of interest. Since each leaf node is a suffix, the leaf nodes only need to store an offset (see right-hand side of Figure 4 for an example).

Highlight the steps required to build a system that makes use of the suffix trie (or tree) functionality, but does not tie itself to a particular implementation (and which can make use of either the optimized or unoptimized versions you implemented).

In your documentation make sure to give a **technical** justification for the choice of this task's programming language.

4 Grading Criteria

The following criteria, described in Table 1, will be taken into consideration when grading your assignment.

Table 1: 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] Tic-Tac-Toe wikipedia article, `https://en.wikipedia.org/wiki/Tic-tac-toe`, Last Accessed: October 14th, 2016.

[2] Tic-Tac-Toe - Play retro Tic-Tac-Toe online for free, `http://playtictactoe.org/`, Last Accessed: October 14th, 2016.

[3] Monopoly wikipedia article, `https://en.wikipedia.org/wiki/Monopoly_(game)`, Last Accessed: October 20th, 2016.

[4] Monopoly | Pogo.com® Free Online Games, `http://www.pogo.com/games/monopoly`, Last Accessed: October 21st, 2016.

[5] Carl Kingsford, Suffix Trees Lecture Notes, Carnegie Mellon University, `https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/suffixtrees.pdf`, Last Accessed: October 27th, 2016.

[6] Ben Langmead, Tries and Suffix Tries Lecture Notes, Johns Hopkins University, `http://www.cs.jhu.edu/~langmea/resources/lecture_notes/tries_and_suffix_tries.pdf`, Last Accessed: October 27th, 2016.

[7] Esko Ukkonen, Online construction of suffix trees, `https://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf`, Last Accessed: October 28th, 2016.

The End