School of Computing Sciences

Module: CMP-6048A
Advanced Programming
Assignment: Project

Set by: Rudy J. Lapeer – r.lapeer@uea.ac.uk
Checked by: Stephen D. Laycock
Date set: 27 September 2021
Value: 60%

Date due: Report: Week 12, 15/12/2021
Demos: Week 12
Returned by: January 2022
Submission: Blackboard (report), Github/Bitbucket link (code)

Learning outcomes

To understand different programming language paradigms.
To be able to program in established general-purpose programming languages such as Java, C++ and C#, as well as learning new languages such as F#.
To understand and be capable of applying the principles of modern software engineering to develop an industry-grade desktop application.
To be able to design and implement algorithms that solve project-related problems

Specification

Overview

To develop a software product following the principles of modern day software engineering and using modern day development platforms and programming languages.

Description

- To develop an industry grade desktop software solution using one or more GPLs (e.g. C#, Java, C++, Python) on one or more platforms with (an) IDE(s) of your choice.
- The UI can be based on higher level platforms or engines such as wxWidgets, WPF, Unity, Unreal, SDL, etc. but these should **not be the integral part** of the project and should be considered as a supporting framework.
- No standalone web apps or mobile apps or database apps or projects of which these three are central to the development. However, they can be used as an add-on to the project of choice.

Relationship to formative assessment

Formative assessment is based on weekly laboratory (follow-up) meetings with the Module Organiser (MO) and an AT (Associate Tutor).

Deliverables

- Live report of no more than 4,000 words but no limit on number of pages. The use of UML diagrams, pseudo code, explanatory diagrams and figures are strongly encouraged. A template is provided.
- Source code and executable.

Projects –

Four choices:

1. Compiler/Interpreter project

a. Maths software

b. Esoteric programming language

2. Graphics simulation

a. Ecology

b. Crowd simulation

See Appendix A for details

Group work: Groups of minimum 2 and maximum 4 members.
See Appendix B for weekly tasks guideline.

Marking scheme

1. Minimal (non-functional) requirements to obtain a good honours mark (60%+):

- A working software product.
- Evidence of adequately using IID, e.g. AP (Agile Programming).
- Well-tested and well-documented programming code which does not contain major bugs e.g. freeze or crash, behaves unpredictable or yields incorrect output, has unintentional or inoperative functionality, etc.
- Implementation of an appropriate (numerical) algorithm.

2. Additional marks (*guideline!) for:

- Broad and/or deep functionality
- Excellent UI/UX
- Multi-threaded or parallel implementation
- Use of plugins or DLLs
- Generic, flexible and/or scalable architecture
- Code optimisation for speed and/or memory usage.
- Use of design patterns, templates/generics, etc.
- Good project management

- Use of multiple languages or multiple language paradigms.
- Other

*Do not “force” the above non-functional requirements/features into your project. A stable and basic deliverable will get higher marks than an unstable one with many additional features!

3. Marks are deducted for:

- Flawed architecture, e.g. not flexible, hard-coded, etc.
- Specifications are not met or are incorrectly implemented.
- Major Bug in a main component of the software (with or without crash).
- Poor project management including time management, task distribution and communication.

4. Guidelines for marks:

< 40 A software product that does not have the essential features as outlined in Section 1 and/or which has substantial flaws as outlined in Section 3.

40-60 A software product that just about meets the essential specs (Section 1) but with little or no additional features as outlined in Section 2 but has no flaws as outlined in Section 3.

60-70 A software product that fully meets the essential specs (Section 1) and may have well-implemented lower-weighted additional features as outlined in Section 2 and has no flaws as outlined in Section 3.

70-85 A solid software product that meets the essential specs (Section 1) and has well-implemented higher weighted additional features as outlined in Section 2 and has no flaws as outlined in Section 3.

85+ A stellar software product with all essential features (Section 1) no flaws (Section 3) and a substantial number of additional high weighted features as outlined in Section 2 and preferably additional well-implemented features not specifically mentioned in Section 2.

APPENDIX A - projects

Project 1a – Interpreter for maths visualisation software

Objective: To create a desktop solution that can be used to evaluate expressions, define variables and functions and visualise functions of two variables. The software should allow the user to enter expressions, statements and commands (via function calls) using a custom syntax which is then processed by a lexer, parser and executional pass. Intuitive and potentially interactive visualisation of functions should be provided. Additional functionality could include zero crossings, function differentiation and integration. Examples of professional maths software that do similar things are Matlab, Mathematica and Maple. An acceptable variation on this theme would be a statistics software. Commercial examples are SPSS, R and SAS.

Project 1b – Compiler/interpreter project for esoteric programming language

Objective: To develop a compiler or interpreter for an esoteric programming language or "esolang" that makes the project fun but without losing the rigour required to develop a syntactically correct language that closely follows the principles of common GPLs or DSLs. A list of popular esolang examples can be found at https://en.wikipedia.org/wiki/Esoteric_programming_language
However, the idea is to come up with your own quirky language!

Project 2a – Graphics simulation – Ecology

Objective: To create a simulation and visualisation of an ecology that has a (finite and rather small) number of species in it, for example, a carnivore, a herbivore and vegetation. Examples are [fox, rabbit, grass], [lizard, bug, plants], etc. Alternatively, a more societal type of simulation could be developed with for example humans, bees, ants, etc. Irrespective of the specific type of simulation that is adopted, it will have a strong Artificial Intelligence (AI) aspect to it where the most basic underpinning computational method would be the finite state machine (FSM). The use of games engines such as Unity and Unreal is permitted and even encouraged.

Project 2b – Graphics crowd simulation

Objective: These simulations visualise certain human behaviour in various situations, e.g. aircraft boarding, traffic or building evacuation. Besides the visualisation aspect, there will be underpinning algorithms that drive the crowd mechanics depending on the specific scenario.

Example scenarios:

- Aircraft boarding: Simulate the boarding of an aircraft using various strategies, for example, see http://www.vox.com/2014/4/25/5647696/the-way-we-board-airplanes-makes-absolutely-no-sense

- Traffic Simulation: Simulate the traffic flow in a city at certain locations and (algorithmically) suggest improvements of the infrastructure (road layout, traffic lights, etc.). It is best to start with a simple T junction and then iteratively increase complexity to create an entire city map. Motorist behaviour can also be gradually made more complex as the project progresses.
- Building Evacuation: E.g. a fire in a building. The software should run different scenarios in existing buildings and potentially suggest improvements to buildings to speed up evacuation. The algorithmic complexity of the crowd dynamics can range from basic to advanced depending on the number of characteristic features a crowd member has. It is best to start with a single room at one level and then add more rooms and levels (floors) as you go along to increase complexity.

APPENDIX B – Weekly task guide

The idea is to adopt a 'living' or 'live' report approach, where your report – which is an important final deliverable – is updated on a weekly basis. This avoids you having to do a full report write-up last minute.
Source code – which is the other deliverable – should also be archived using version control. We advise you to use git in conjunction with either Bitbucket or Github.

Next, a (loose) guideline is given for each week as to what you should work on during, and have achieved by the end of, a particular week. You should keep track of your weekly updates and what tasks having been accomplished (you can use for example Trello). We aim to adopt a scrum-styled AP approach with two-weekly sprints.

Week 1 – Formation of groups. Decide on a project, i.e. 1a,1b, 2a or 2b. Discuss specific objectives with your group. Discuss the non-functional aspects, i.e. what language, what IDE, platform, etc. These tasks will be aided by the teaching team (TT) during the live lab sessions. Communicate your choice and specs to the TT. If you cannot find a group as yet, decide on your project of choice and the TT will place you in a group by the following week.

Week 2 – Do a background review on similar systems or research (literature review). Write this up in your live report in the Introduction chapter. Create a MoSCoW.
Start your first sprint. Decide what functional aspects will be covered and who will be doing what. Make sure this is small and manageable and can be completed in two weeks.

Week 3 – Finalise your first sprint. Perform final tests (some tests, like unit testing, should be done as soon as you completed a small part of the code). Add used methodologies to the relevant chapter in your live report and summarise the results of your first sprint in the implementation chapter. Add tests results in the testing chapter.

Week 4 – Based on your successes (and failures) of your first prototype following the first sprint, start to specify the requirements for your next sprint and who will do what. Again, the upgrade should be manageable so requirements can be completed within two weeks.

Week 5 – Finalise your second sprint. As before, perform final tests and update relevant sections in the live report.

Week 6 – Before starting the next sprint, reflect on your original objectives, adapt the MoSCoW if needed to ensure that you are "en route" to meet your final objective. You may wish to consult the marking scheme guidelines in this document to make sure you meet the minimal essential (non-functional) requirements stated there. Any additional broad features as outlined in section 2 of the marking scheme should be decided on now. Once you are satisfied you can start your third sprint.

Week 7 – Finalise the third sprint in the same vein as before.

Week 8,9 – Same scenarios for the fourth sprint.

Week 10 – You are coming close now to the submission deadline (Week 12). So now is the time to reflect again on what you have achieved so far and what is still remaining. This is similar to what we did in Week 6 but now you need to make an informed decision of what functional aspects of your project can still make it to the final deliverable and what will need to be dropped. By now you should meet the essential and additional (non-functional) requirements as set in sections 1 and 2 of the marking scheme.

Week 11 – Round up. Finish off your final deliverable, fix any remaining bugs, loose ends etc. so your code meets all your 'M' functional requirements and meets the minimal non-functional requirements of the marking scheme and does not have any flaws as outlined in section 3 of the marking scheme. Finalise and 'clean up' your live report by cutting out sections that are no longer relevant and make sure it is complete, coherent and easy to read. Make sure appendices are complete and are referred to from the main text body.

Week 12 – Demos and submission of report.