Capstone Project—Software Development Project Assessment Guidelines*

26 June 2018

1 Scope

These guidelines are intended to apply to 25 point "software development" projects, as available in the MIT and the older MEDC and MSSE. It is not intended that a given project code be designated as a software development project, but rather that a supervisor and student can agree that the project will include a software deliverable and therefore will be assessed as a software development project.

Information about research projects (as opposed to software development projects) is provided in the separate document *2018 Guide to Research Projects*.

2 Rationale for Software Development Projects

These marking guidelines are specific to Capstone Projects involving significant software development – called Software Development Projects. Such projects are distinct to Research Projects, as Software Development Projects do not necessarily involve research – marking guidelines specific to Research Projects, such as contribution and publishability, are not entirely sensible for Software Development Projects. Having said this, Software Development Projects will almost certainly address research needs, for example in the form of a reusable tool or utility that analyzes research data, a simulator, or an experimental software system used as a proof of concept. The emphasis of the Software Development Project is therefore as much as demonstrating a clear understanding of the objectives of the software to be developed and establishing that such objectives were met, as it is in the quality of the developed software itself.

As further specific distinction from a Research Project, the Software Development Project produces software as one of the assessable outcomes. If software was not expected to be delivered as part of the project then the assessment could be largely focused on the written report. For example, the student in a Software Development Project does not simply analyze data or run a simulation as they may in a Research Project, but must develop and provide the software that does the analysis or that is the simulator itself. The software must be provided in a way that it is generally reusable by other staff and students. In some cases, for example when a simulator exists but which may be complex in nature, the student may produce the

*This document was created by Aaron Harwood. Updates by James Bailey (February 2018) and Harald Søndergaard (June 2018).

configuration files and environment as the outcome of the Software Development Project that enables others to readily undertake an otherwise complex simulation process in a reusable way. In this way the project does not necessarily result in software par se but in environments, testbeds and other moderately complex system infrastructure that demonstrates equivalent software development abilities. For example the collation of various complex data sets, such as spatial data or social network data, into appropriate databases and the establishment of procedures for data access and processing may well be an acceptable Software Development Project.

3 Assessment Components

In addition to the usual assessment items considered in a Research Project, the Software Development Project includes one more item, and thereby the following assessment items are what might typically be included in the assessment of a Software Development Project:

- end-of-semester oral presentation (10%). Due date is the same as per the oral presentation for research projects.
- end-of-semester report (45%). The end-of-semester report could be a final report (in a final semester) or a progress report (if the project extends to another semester). Due date is the same as per the end-of-semester report for research projects.
- software (45%). Due date is the same as for the end-of-semester report.

This way, a 25 point Software Development Project, that might otherwise involve a thesis of 6,000 to 8,000 words without software being produced, now involves a thesis of 3,000 to 4,000 words with a 45% software assessment item.

4 Thesis Structure

Depending on the exact goals of the project, two possible thesis structures can be considered:

- a conventional Research Project thesis
- a software engineering thesis

The structure, and the assessment guidelines, for a conventional Research Project thesis is explained in the separate document *2018 Guide to Research Projects*.

In this case the thesis treats the project from the perspective of addressing a research question whereby the delivery of a software product is warranted as a key contribution. For example a proof of concept might be undertaken from this perspective. For a software engineering thesis the structure is derived from an applicable software engineering methodology, usually one including use cases, requirements, design, implementation and testing. For example a research tool or utility might be undertaken from this perspective.

5 Software Assessment Criteria

The assessment of software is a rocky road, requiring considerable time (especially if a demonstration is included) and reasonable practical experience. It is understandable that assessment of software is often avoided in favour of a written report that details the software. Build environments, third party libraries, languages and complex system support such as databases, web systems and Internet services can quickly overwhelm an unprepared examiner.

To aid the software examiner, one or more of the following criteria may be applied to arrive at a final assessment mark. Note that some of these some assessment criteria (e.g., building the software) would be treated as PASS/FAIL hurdles in deriving an overall mark.

- coverage of objectives
  - low (0% to 64%): very few if any of the objectives appear to be met in any clear way
  - medium (65% to 74%): the objectives have been met, at least in part but clearly not in entirety and/or there are some clear deficiencies or aspects that fall short
  - high (75% to 100%): all of the objectives have been addressed, with few cases that may be questionable and/or additional aspects explored
- usability
  - low (0% to 64%): very little to any of the software is intuitive to use, there are little or no instructions, excessive effort is required to make use of the software
  - medium (65% to 74%): can be used with some effort, outcomes are as expected in most cases but there are some aspects that are confusing and/or need more effort than warranted
  - high (75% to 100%): generally easy to use, very few unexpected outcomes
- technology choices
  - low (0% to 64%): no or little ability to make use of appropriate third party libraries or systems, inappropriate/ill-suited choices of technology,
  - medium (65% to 74%): reasonable choices of technology but lacking in the capacity to integrate/make use of them, perhaps some choices could have been better in hindsight
  - high (75% to 100%): excellent choices of technology and clearly established ability to use them with few if any caveats
- application of computing knowledge
  - low (0% to 64%): no or little demonstration of anything other than basic programming skills, frequent use of superfluous and inappropriate programming approaches
  - medium (65% to 74%): a reasonable demonstration of computer science knowledge in terms of data structures, algorithms and other applicable theory, but largely piecemeal without sophistication

  - high (75% to 100%): can apply computer science knowledge when and where appropriate showing sophistication, a high level of appreciation for programming intricacies

- quality of source code
  - low (0% to 64%): extremely hard to read, little or no attention to code formatting, lack of logical organization, little or no appreciation of coding and commenting practices
  - medium (65% to 74%): reasonable clarity and code can be followed with some effort, comments help to understand the code, logical organization may be lacking in some ways and there may be a number of questionable expressions
  - high (75% to 100%): a high standard of coding and commenting is evident, the code is easy to read and appears clear to maintain, ready to be open sourced

- building and installation
  - low (0% to 64%): very difficult or impossible to build without modifications, frequent build problems, little or no instructions and more or less unable to install in any reasonable way
  - medium (65% to 74%): can be built and installed with some effort, problems occurred but can be overcome with minor changes or clarifications to instructions
  - high (75% to 100%): straight forward and easy to build and install, little or no effort required (works out of the box)

6 Selection of Criteria and Weighting

To provide the greatest scope for the application of these guidelines, it is recognized that the assessment criteria are not always all applicable to the project under consideration. Projects that involve user interface design may not have an need to assess the application of computing knowledge. There are two ways to approach this:

- The examiner makes a case by case argument/statement for each criteria that is judged relevant, based on the examiner's expert opinion.

- The supervisor indicates to the examiner which of the criteria are to be applied; and these criteria may be negotiated with the student at the beginning of the project. If source code of any kind is to be delivered then it would be expected that the majority of the criteria, including quality of source code and building and installation, will be included.

Some criteria such as building and installation may be given less weight than others. For example the building and installation is sometimes a mere case of succeeding, for example in the case of a Java program that compiles without bother, without any real effort. In these cases a mark of 100% for this criteria is clearly less important. The criteria can be considered as a Pass/Fail hurdle, in that the software will not be assessed unless it can be built and installed.

From time to time it may be useful to update the list of criteria.

7 Independent Examination

In this case the examiner sits alone to examine the software. This is suitable for software such as tools and utilities where test cases can be applied. It may or may not be suitable for software that requires a web server or a database to be installed; it will depend on the examiner's resolve. Independent testing is the most reliable form of testing since the examiner's confidence that the software is genuinely usable is evident from a successful examination process.

7.1 Assisted Examination

In this case the examiner asks for assistance from the student during the examination of the software, perhaps including questions directed at clarifying aspects of the assessment. This breaks from the tradition of students not knowing their examiners. The examination proceeds more or less as an independent examination however the student is available to expedite the examination in cases which may otherwise be lengthy. For example where a building or installation procedure is tedious, or requires specific build tools, or where login credentials may be needed.

7.2 Demonstration

In this case the examiner requires the student to demonstrate everything required to undertake the assessment. The examiner asks the student to proceed through the various aspects of the examination. Again, this breaks from the tradition of students not knowing their examiners. For complicated systems this may be the only real option, for example if a system is run on external resources such as a supercomputer in another organization.

7.3 Software Considerations

As well as considering the structure of the thesis, it is prudent to outline general expectations and/or scope of the structure of the software. If the software is to be genuinely usable by other staff and students then there is certainly something to be said. It is not the intention here to mandate specific practices, technologies or systems, since doing so can readily eliminate possibilities and quickly become outdated. It is also unwise to prescribe specifics concerning management of code and other secondary processes and practices, as these things can easily consume copious amounts of student time to address; in sometimes what is an otherwise time limited single semester. For example it may be nice to consider that all software be developed using the 'autoconf' methodology, which is a standard practice for UNIX environments and portability across OSes, but this is not currently taught in the School and it would require too much time to address on a case by case basis. It is also not the best choice for all kinds of software development. The conclusion is that these matters of management are best left to the discretion of the supervisor and the student.

Having said this, expectations and scope are especially important with respect to examining software. If the examiners cannot independently examine the software on their own office desktop then there is very likely a large barrier to be overcome. The testing of some software may require login credentials to, for example, a specific server or cloud. It might require administration privileges or special networking support. In these cases the examination can quickly escalate to requiring a demonstration.

The following aspects are important to keep in mind and discuss at the beginning of a software development project:

- source code – The software should be deliverable as source code. It is recommended that this be in the form of a self contained directory structure that is self descriptive, with appropriate named directories and concise textual information files (for example 'README.txt', 'COMPILING.txt' and 'INSTALL.txt') at the root directory, and delivered as a single archive file that can be unarchived by the examiner and inspected.

- build environment – Source code is often intimately attached to a build environment. The requirements and procedures for building the software should be contained along with the source code in an obvious place for the examiner to find. In many cases the source code directory structure is itself a kind of project defined by a build environment, for example an Eclipse Project or an iOS Application. If the software cannot be built independently by the examiner then there is significant doubt surrounding its genuine re-usability. At the very least the build process should be examined by demonstration.

- execution – Tools and utilities generally have a single executable that may well be supplied along with the source code. More complex systems, for example real-time analysis of streaming data, may require a number of processes to be started. Nevertheless these things should be clear for a third party (the examiner) to execute. Generally speaking the instructions given with the source code should be a step-by-step, "run this command", "edit this configuration file", "double click this", style.

- operating system – The operating system requirements should not be prohibitive. Generally speaking the operating system should be available to staff and students; i.e., UNIX, Windows or Mac. At the very least the supervisor should be in a position to be able to examine the software on systems available to them. In some cases the software may be developed for an embedded system such as a mobile device or sensor and the hardware should be available to the School.

- third party libraries and systems – Similar to cited publications, all third party libraries and systems should be readily accessible and either without charge or as part of a package that the School has access to. In some cases, for example where a Google or Twitter subscription is required to use a third party service, this can be problematic and may require examination by demonstration.

8 Roles and Responsibilities for Student and Supervisor

A software development project is a significant undertaking and the interaction of student and supervisor is an important component of the research process. That interaction can be greatly facilitated if both the student and the supervisor have a clear understanding of their respective roles and responsibilities. It is good to agree on those expectations up-front at the commencement of the supervisory relationship. Some things to discuss include frequency and format of meetings, preparation required before meetings, degree of direction to be given by the supervisor, frequency of submission of intermediate writing, timing of turn around on feedback. There are no right or wrong answers to these issues but it is vital that all concerned are clear of what to expect at the start of the process.

8.1 The Integrity of the Project Report

The integrity of the report rests on it being the students work; therefore supervision should be supportive but also at arms length. It is neither ethical nor reasonable for the supervisor to conduct the project or write the report. Therefore the student should have no expectation that the supervisor will overly assist with the work. In particular:

- The supervisor must not collect or analyse the data.
- The supervisor must not write or heavily edit the report. The University of Melbourne PhD Handbook defines editing as the detailed and extensive correction of problems in writing style and of mechanical inaccuracy (as opposed to giving general guidelines about problems). Just as with a PhD, this is not acceptable supervision practice during Masters. Although the supervisor should provide commentary on writing style and presentation, it is not the supervisors task to write the report, or any part thereof.

8.2 Roles and Responsibilities of the Student

The student should initiate many of the activities in the supervision of a software development project, in particular the administration activities. Responsibilities of the student include:

- Selecting a software development project topic
- Informing the course Coordinator of the topic selected
- Preparing a software development project proposal
- Maintaining progress as documented in the software development project proposal
- Negotiating alterations to the proposal with the supervisor
- Meeting regularly with the supervisor (it is advisable to maintain a diary entry that summarises each meeting email this to your supervisor after each meeting)
- Raising any issues or problems with the supervisor at an early date
- Ensuring that the writing style and presentation of the report is appropriate
- Copying, binding and submission of the report as outlined above.
- Reporting on progress to supervisor at the end of each semester
- Demonstrating the software to supervisor and possibly examiner(s), as required.

8.3 Roles and Responsibilities of the Supervisor

The supervisor should meet regularly with the student and provide assistance and monitor progress. Responsibilities of the supervisor include:

- Negotiating a suitable software development project topic with the student
- Assisting the student to prepare the software development project proposal
- Guiding the student to appropriate reference material
- Checking that the work contained in the proposal looks feasible and appropriate
- Meeting regularly with the student
- Informing the course Coordinator if the student fails to attend scheduled meetings without reason

- Checking for writing style and presentation problems
- Acting as an examiner for Masters reports and projects
- Where appropriate, encouraging the student to publish their research or make their software open source
- Monitor progress of student for each semester

8.4 Difficulties in the Student/Supervisor Relationship

The majority of student/supervisor relationships are supportive and rewarding, for both parties. Difficulties may arise from time to time, and these difficulties can interfere with progress. It is important that any issues are resolved respectfully and quickly - there is no time to waste in a relatively short project! If either the student or the supervisor has concerns that they are unable to resolve within supervision, they should discuss these with the Course Coordinator who will aim to assist in moving the situation forward. If this proves unsatisfactory, then the matter should be discussed with the Head of School.