JEROEN KEPPENS

SOFTWARE ENGINEERING GROUP PROJECT HANDBOOK 2025/26

DEPARTMENT OF INFORMATICS
KING'S COLLEGE LONDON

PUBLISHED BY DEPARTMENT OF INFORMATICS
KING'S COLLEGE LONDON

Current version: September 2025

Contents

1
Introduction

Congratulations on reaching Year 2 of your course and welcome to the Software Engineering Group Project module.

This document is a handbook on the Software Engineering Group Project (SEG) module. It identifies the objectives of the module, explains the way the module is organised, sets out the most important expectations, provides deadlines and key dates, and specifies how you will be assessed. Formally, SEG is a *core* module, which means that you must pass it (a fail mark cannot be compensated or condoned). It contributes 30 credits and takes places during both Semesters 1 and 2 in Year 2. It is assessed by two pieces of coursework, each based around a group project.

FAQ: What is the software engineering group project module?

1.1 Why a handbook?

The SEG module differs considerably from other modules in that it concerns producing software systems *with others in the context of a substantial project*. Therefore, this module focusses on team-based software engineering, project management, and collaboration with others, and it is assessed almost exclusively via group work. Compared to individual assessment, group work comes with considerable additional uncertainty or risk, because the achievements of your team are dependent on the abilities, attitudes, behaviours, and interactions of others, not just your own. The module is designed around managing that risk, promoting (rewarding) constructive and sustained engagement with team mates, effective collaboration, and sound project and team management. Learning and work practices that may have worked well for you in other modules, may not work for this module.

FAQ: Why does this module have a handbook?

This handbook describes in considerable detail what is expected of you in this module. It describes the organisation of the module, what you need to learn to be successful in this module, the tools we use or can use, and how the assessments are organised and marked.

1.2 How to use this handbook

This document has become rather long, so I do not expect you to sit down and read the whole thing as soon as possible. The handbook

FAQ: How should the module handbook be used?

has been written to provide relevant information at key stages of the year, and to be used as a reference text to enable you to look up information as and when you need it.

Some of the chapters are organised to be relevant to key stages of the module. The Study Guide chapter (Chapter 2 should be read *before we meet in the first lecture*. This chapter tells you what this module is about, what teaching/support is provided, how you will be assessed, and what tools you need to acquire to engage with the module. While the module will be introduced in the first lecture, I will not repeat everything in the Study Guide chapter as that would make for an overly information dense session. If the Study Guide raises any questions, do make a note of them for the Q&A component of the session.

Modern software engineering is heavily reliant to tools. In this module, you will learn how to use a range of software engineering tools and gain experience in using them in team-based projects. The Tools chapter (Chapter 3) presents of overview of the tools you require and what you need to use them for. In order to stay on track with the module, it is crucial that you install certain tools before the second week of teaching.

Next, familiarise yourself gradually with the General Expectations chapter (Chapter 4) in the first weeks of the academic year. This chapter will give you an idea of what is expected of you and your team mates in the remainder of the academic year. This handbook includes chapters on Project Management (Chapter 5) and Software Quality Assurance (Chapter 6). These chapters provide more specific expectations on how you should be working in the group projects and what standard of work is expected of teams. It is worth reviewing these during the first (small) group project you undertake as a guide.

You will be undertaking two group projects in this module: a Small Group Project and a Major Group Project. The chapters by the same names (Chapters 7 and 8 respectively) advise you how these projects are organised, and how you will be assessed. You should review these chapters before the respective group projects assignments start.

This handbook is provided as a searchable PDF document, so that you can look up specific information as and when you need it. In the margins of this text, you will find Frequently Asked Questions (FAQs) that the main text answers alongside it. The text is organised such that each lowest level section, subsection, or paragraph answers a FAQ: a question that at least some students have each year. You can find a list of the FAQs (Chapter **??**) at the end of the handbook, along with the page number where that question and the corresponding answer appears in the handbook. The handbook is also indexed with key terms, so that you can look up important terms and information.

2
Study guide

2.1 Aims and objectives

The SEG aims to meet certain important learning outcomes imposed by the British Computer Society (the UK's professional body of Computer Scientists) that are critical to a professional career in Informatics. This section explains in broad terms what the purpose of this module is.

FAQ: What are the aims and objectives of the Software Engineering Group Project module?

Virtually all Computer Science work involves other people. Software systems are typically developed *by* teams, often multi-disciplinary ones, *for* clients and end-users, *with the support* of people who provide financial support, support in-kind, or access to resources, and *with consideration* of people affected by the change the system will bring. Indeed, often the main challenges in software engineering stem from humans, not technology [DeMarco and Lister, 2013]. Therefore, this module is, first and foremost, about working with other people and the challenges that brings. You are not being assessed on what you can produce individually, but on what you can contribute in collaboration with fellow students!

FURTHER READING: On KEATS, navigate to *Module organisation* → *Learning outcomes* for a comprehensive list of the formal learning outcomes of this module.

Group project work introduces a number of difficulties that you would not experience in an individual project. It requires you to agree objectives and a plan to achieve those objectives with others. A group project rarely proceeds entirely as expected. A team needs to hold itself and its team members to account, and intervene when necessary. Recognising and discussing what is not going well is difficult but essential to success. The module will introduce a range of strategies to overcome such challenges and you have opportunities to put them into practice.

EXPECTATION: To participate meaningfully with your group project, you must engage fully with your team. As a minimum, you must attend all meetings, listen to others, share your views, and present your work.

In this module, you will produce software systems *as a team*. Producing code is an important component activity in that process that everyone must fully engage with. However, software engineering goes beyond mere coding. Other important activities include agreeing *business objectives* for a client or prospective client, defining *requirements and specifications* for a software solution that would enable the objectives to be achieved, producing and continuously refining a *design* for the system, verification and validation including writing *software tests* and reviewing *code quality*, and *deploying* the system.

EXPECTATION: During a group project, you must produce some code each week, share it via GitHub, and be prepared to present it at a team meeting. Other contributions will be welcome and needed as well, but contributing weekly code is essential.

EXPECTATION: As a Computer Scientist, you ought to be able to administer your own workstation without excessive reliance on IT support, so it is important that you develop the skills to do so.

The productivity of a software developer is enhanced substantially by means of tools. These tools include (but are not limited to) a UNIX Command-Line Interface (CLI) to complete a range of system administration tasks, text/code editors, interpreters/compilers (typically called from a CLI), version control (e.g. git and GitHub), and build automation tools. Using such tools requires a level of computer literacy that goes beyond that of a typical user. As part of the course, you will need to learn to employ these tools.

2.2 *Module structure*

FAQ: What is the overall structure of this module?

The SEG module consists of three parts:

1. *Teaching & training*. The taught component aims to ensure that you learn the fundamental knowledge and skills to enable you to participate in the group project. You will learn some important fundamentals of project management, software engineering, and software development tools and practices. You will also be trained in a new programming language and framework that you will use in the small group project (and optionally in the major group project). This component consists of live lectures, video lectures, small group tutorials, and self-guided coding exercises. It takes place in the first eight weeks of the academic year, though most of it will be complete before the Semester 1 reading week.

2. *Small group project*. In the second half of Semester 1, you will participate in a six-week group project. Think of this project as "practice" group project. It is relatively low risk as it contributes only a small portion your overall mark. If anything goes very wrong, you will not lose all that much. However, you are likely to experience a number of potentially significant challenges of group project work. The experience you develop in this project will be valuable as you tackle the final stage of the module. All teams will be working on the same assignment (to deliver a deployed web application) and use the same technologies. This allows us to provide detailed instructions – based on the ideas introduced during the taught component – on how the project should be managed, and what each team and each team member should be doing on a week by week basis.

3. *Major group project*. Semester 2 brings everything you have learned together in the form of the Major Group Project. The project assignments will be substantial, and teams will normally require all of Semester 2 to complete the work to a good standard. You will have an opportunity to form your own teams, choose a project assignment from a short list of topics, and select the technologies you wish to use. Teams will be largely self-guided. However, you should rely on what you have learned in the previous stages of the module, and there will be some advisory meetings with academics.

2.3 *Assessment*

FAQ: What is the assessment structure for the module?

The module's assessment consists of two group projects: your individual small group project mark accounts for 15% of your overall individual mark, and the major group project for 85%. As this is a core module, you must pass it to graduate. To pass the module, you must obtain a combined mark of 40 or more. In other words:

$$0.15 \times \text{individual overall small group project mark} +$$
$$0.85 \times \text{individual overall major group project mark} \geq 40$$

There are *no* other requirements to pass the modules. You can pass the module even if you fail one of the two assessment.

The two group projects are quite different. The small group project takes place in the second half of Semester 1 and aims to prepare students for the major group project. Team are small: 4–5 people are allocated to each team, based on information provided in a registration form. The technology stack is heavily constraint. There is only one assignment that all teams must complete (though there is considerable flexibility in scope, depending on effective team size and ability). The major group project covers all of Semester 2. Students can form their own teams of around 8 people and teams can choose the technology stack. A range of project assignments will be proposed to cater for a wide range of interests.

Both group projects have the same overall assessment structure. Each individual overall group project mark consists of two components: an individual group project mark accounts for 95% of the mark and a peer assessment mark for 5%. The individual group project mark is derived from a team group project mark. The remainder of the section explains the role of the peer assessments and the relationship between individual marks and peer assessments in more detail.

2.3.1 *Peer assessments*

FAQ: How do the peer assessments affect me or my mark?

At the end of each group project, you must participate in a peer assessment exercise. In each peer assessment exercise, you asked to fill out a peer assessment form for each of people you worked with in the team. This form consists of a number of short, usually multiple-choice questions, and feedback field that allows you to write open-ended feedback for your team mate.

Some time after the deadline for submitting the peer assessments, you will receive feedback from the peer assessments your team mates returned about you. The author of each peer assessment will *not* be revealed to you. You will have an opportunity to respond to the peer assessments you received, but you are not required to respond. If a peer assessment contains factual errors or if you wish to provide context for a peer assessment, then it is a good idea to respond so that the concerns you have about a peer assessment are recorded in the system.

The peer assessments are used in two ways.

Firstly, the *peer assessments you write* are assessed and contribute to your overall mark. Your overall mark for each assignment (i.e. the small group project mark and the major group project mark) is the weighted average of your individual group project mark and your peer assessment mark. The peer assessments you write contribute 5% to the assignment mark. The remaining 95% is the individual group project mark:

$$\begin{aligned}\text{assignment mark} = &0.95 \times \text{individual group project mark} + \\ &0.05 \times \text{individual peer assessment mark}\end{aligned}$$

Note that, given the weightings of each assignment, the small group project peer assessments contribute 0.75% ($0.05 \times 15\%$) to the overall mark, and the major group project peer assessments contribute 4.25% ($0.05 \times 85\%$) to the overall mark. Because of this, the assessment criteria for the latter are more elaborate than those of the former.

Secondly, the *peer assessments you receive* are considered along with attendance and code contribution statistics when determining individual project marks. This is explained next.

2.3.2 *Individual group project marks*

FAQ: How does my individual mark relate to what the team produces?

EXPECTATION: As this module assesses your ability to work in teams, the marking schemes are designed to promote team work rather than individual work. To do well in the module, focus your efforts on improving the team's output.

The group projects are assessed on work delivered by teams. This work is assessed to produce the team's group project mark. Individual group project marks are *based on* the team's group project mark, but it is not necessarily identical to it. An individual group project mark is computed using one of two formula, depending on whether the individual is counted as a participant who met all the expectations of the group project in full.

If an team member did not meet the individual expectations for the group project, the individual mark will be a reduced proportion of the team mark:

$$\text{project mark} = \text{team mark} \times (1 - \text{major correction})$$

where the major correction is in the range 10% to 100%, depending on the extent expectations were not met. An individual mark scheme describes the circumstances in which a major correction is applied and how large it is.

If an individual is counted as a full member of the team, their mark will be very close to the team mark (i.e. within 5 percentage points of it:

$$\text{project mark} = \text{team mark} \pm \text{minor adjustment}$$

where minor adjustment ranges from -5 to +5. The minor adjustments are effectively a small redistribution of marks (within a narrow range), where marks of some team members are moved to other team members. The sum of all minor adjustments within a

team is constrained to zero, so that for some people to gain marks, others need to lose marks. The minor adjustment range equals -5 to +5 to ensure that (only in extreme cases) the difference between the lowest and highest marked fully contributing team member is no more than a grade boundary.

Both group projects come with minor mark redistribution schemes to decide minor adjustments. Because nobody involved in the group projects is both impartial and omniscient (the team members are not impartial, and the lecturers not omniscient), these redistribution schemes are inevitably crude. In essence, they seek to reward people with an adjustment of +3, 4, or 5 where the peer assessments and other contributions statistics are significantly greater than those of others, or penalise with an adjustment of -3, 4, or 5 where those statistics are significantly smaller than others (even though these people are still recognised as fully contributing team members). A mark adjustment between -2 or +2 normally means that either someone else has been rewarded or penalised, but you are not deemed to have contributed more or less than other fully contributing members.

REALITY CHECK: It is impossible for a marker to conduct an objective comparative assessment of the contributions of individual members. To achieve that, a marker would have to be intimately familiar with the inner workings of each team. We simply do not and should not have that kind of access to a reasonably independent team. Moreover, there often is no objective truth. In a team of five people, there will be up to five different opinions on the merits of every members work. Where teams work well, individual contributions are not easily separated from one another. As markers cannot prefer one student's views over those of another, individual marking will be based on the evidence available to us – participation statistics, code statistics, and peer assessments. Subsequent pleas cannot change individual marking. However, reports of errors in the evidence or falsification of evidence will be considered.

2.3.3 *Effective team size*

A common concern among students is the effect of team size on team output. Initial team sizes vary as your class size is unlikely to be completely divisible by our desired team size. Moreover, each year, a number of students fail to engage with their group project team, disengage from the group project, or need to interrupt their studies. Some small group project teams will have 5 people. Some teams may end up with just three people, or even two in extreme cases.

FAQ: The number of participating people in my team is smaller than in other teams. How will that affect us?

Many assume that smaller teams have a disadvantage compared to larger teams. After all, larger teams have more people to produce work during the same period of time. Therefore, one might expect their combined output will be larger. In practice, that is not universally true. In fact, Brook's law suggests that "*adding [people] to a late software project makes it later*"[Brooks, 1975]. Based on observations made during his professional experience, Brooks suggests that as teams grow, they have greater communication and coordination costs, and the increase in these costs can be greater than the benefit of the extra person.

Nevertheless, the detailed team marking criteria for the small and major group project[1] consider the number of people actively engaged with the group project. This will be referred to as the team's effective size. Effective team size will be calculated at the end of the project, based on the applied major corrections, as follows:

[1] You can find the team marking criteria for each project in the chapter on that project.

$$\text{effective team size} = \sum_{i=1}^{n} 1 - |\text{major correction for student}$$

This number will be rounded down to the nearest 0.5. For example,

if no member of a team of 4 people has a major correction, their effective team size equals 4. If, in a team of 5, two member receive a major correction of -70%, the effective size of that team is deemed to be 3.5 ($5 - 0.7 - 0.7 = 3.6$, rounded down to the nearest 0.5.

The marking schemes mention effective team size exclusively in relation to functionality. Teams with fewer people should build applications that are smaller in scope. All other criteria are assessed irrespective of team size.

EXPECTATION: As a general rule, try to complete tasks as soon as possible, and delay decisions as late as possible.

It is not possible to determine a team's effective size until the project is over. Some teams wonder how they can possible decide the scope of the project at the start when they do not know what the engagement of team members will be. The answer to that question is that you don't decide the scope at the start. By building software incrementally and iteratively, always ensuring that what you do adds value, the impact of the risk of unpredictable team size can be reduced considerably.

2.4 *Mitigating circumstances*

FAQ: I have difficulties engaging with the project due to circumstances outside my control. What can I do?

The group projects require sustained engagement with the project throughout the project period (six weeks for the small group project and ten weeks for the major group project). If you are unable to engage sufficiently with the group projects due to circumstances beyond your control, you can request mitigation by submitting a Mitigating Circumstances Form (MCF). The module lecturers cannot offer any form of mitigation (including deadline extensions), so if you send them a request for mitigation, they can only signpost the MCF processes.

To submit a MCF for this module, you should follow the same process as for any other module:

- Use the form on Student Records to create a MCF to make your case.
- Clearly indicate what assessments are affected by your mitigating circumstances. Beware that each group project comes with two submissions: the group project submission and the subsequent peer assessment. Clearly indicate all assessments/deadlines that are affected.
- Submit the MCF on time.
- Submit evidence within the timeframe specified on Student Records.

After your MCF and the associated evidence are submitted, your case will be considered by the Department's Mitigating Circumstances board. If you require help submitting a MCF, contact your programmes officer, personal tutor, or the KCL Student Union Advice Service.

If your MCF is approved, the Mitigating Circumstances board will decide what mitigation they will offer. Although you can request a particular mitigation, you will not necessarily be offered what you asked for, even if your MCF is accepted.

For the group project deliverables, *only* the following mitigations are possible:

- *An extension of the group project deadline by two weeks*. In other words, if you receive this mitigation in response to your MCF, the team's deadline is extended by 14 days from the original deadline. This mitigation is *not* cumulative. Even if several members of the team submit MCFs and all receive the standard two week deadline extension, the new deadline is only two weeks from the normal deadline. The extensions apply to submissions associated with the coursework, including peer assessments.
- *An individual deferral of the affected group project*. If you receive this mitigation, the current attempt at the group project is deemed void and moved to the next available opportunity to take this group project. In this case, the peer assessment associated with the group project will be deferred as well.

No other mitigations are possible. Specifically, we cannot adjust your marks (including your team mark or any individual corrections) based on mitigating circumstances. The College Regulations do not allow this.

2.5 *Reassessment: resits and replacements*

FAQ: What happens if I need to redo a group project?

A student may be allowed to take two attempts at a module. The first attempt is marked in the normal way. The second attempt is capped at the pass mark (i.e. 40%).

You may require reassessment of one or both group projects under two circumstances.

- If you are granted a deferral of a group project if your MCF is approved or following a successful appeal, then you will take a *replacement attempt* at the group projects. A replacement attempt does not count as a new attempt. Thus, a replacement first attempt counts as a first attempt and is, therefore, uncapped.
- If you fail the module at the first attempt, you may be allowed to do the module again. This is called *resit*. Resits are, by definition, capped at the pass mark (40%).

The department's replacement policy is as follows:

- If you are granted a deferral for the small group project, *and* you subsequently pass the major group project, then you will be offered a replacement attempt during the summer (normally in July). This replacement attempt will take the form of an individual project, offering you an opportunity to pass the module

at the first attempt before the end of the academic year in which you started the module.

- *In all other circumstances involving a reassessment, you must retake the entire module in the next academic year*. If you require a resit, or you defer either group project until the next academic year, you must retake both group projects in the next academic year. The department does not offer partial reassessments of modules.

Note that if you have been granted a deferral for the small group project but pass the module overall based on the major group project alone, you can request to cancel the deferral to allow you to complete the module without taking a new attempt at the small group project. Normally, we would advise you to take a replacement attempt if one has been offered. However, because deferring the small group project to the next academic year would require you to take the major group project again, you may wish to avoid the associated time commitment as that can prolong your studies unnecessarily.

We only offer a reassessment of the major group project once a year, in the Semester 2. There are good reasons for this. The major group project assesses how students work in teams. Therefore, it can only be assessed through a larger/longer group project in a substantial team. It is challenging to organise this reliably. Outside Semester 2, we simply do not have sufficient students available who are likely to engage with the module.

2.6 Policy on generative AI

FAQ: Can we use generative AI in this project?

As in many other fields, generative AI is increasingly becoming an important tool for software engineers. In this module, you will have opportunities to gain experience with using generative AI. Beware that other modules have their own policies and may prohibit generative AI altogether. Read this section carefully to familiarise yourself with the constraints that apply.

In this module, the following rules apply:

- For the peer assessment component – i.e. writing the peer assessments and/or responding to peer assessments – use of generative AI is strictly prohibited. In other words, you are not allowed to use generative AI for writing peer assessments or responses.
- For all other work, including coding and report writing, we employ Model 3: "Authorised use of Generative AI tools to generate low-level output". The remainder of this section briefly explains what this means

This means you can use generative AI and similar tools, such as GitHub Copilot to author simple routines that are small in scale and carry out a specific functionality that is likely to have already been implemented by a library. You can use generative AI as an

alternative to Python and Django documentation to find the functions, classes, and methods that enable you to reuse existing code. Generative AI tools are also useful as an alternative to searching/browsing tutorials, forums (e.g. Stack Overflow), and examples to find out how to complete common tasks. In software development, this type of code is normally reused but generative AI makes it easier to find and use as it can generate more readily usable examples from natural language prompts. This can make you considerably more productive as a software engineer. However, it also introduces new challenges as it allows you to build larger systems more quickly.

Generative AI tools tend to do well on small, common, and highly specific coding tasks. However, if you set it tasks that are larger, less common, or framed in vaguer terms, these tools become increasingly less reliable. Generated code may fail to meet the requirements of the task, software quality standards, or the design assumptions of other parts of the code. You are responsible for any code that you share with the team. In particular, you must ensure that the code is fit for purpose, meets your team's quality standards, and permits integration with the remainder of the system. As mentioned at the outset of this section, generative AI is a powerful tool. It is not a hack: you will need to keep your brain switched and use these tools responsibly.

The College requires that all contributions by generative AI systems are acknowledged, including code. I appreciate this is cumbersome, but it is a College requirement that I am not allowed to overturn. For both group projects, you should acknowledge all sources in the README.md file, but not through comments in the code (as that undermines code cleanliness). For each use of generative AI, state the path to the file, the affected function, method, or class, the number of lines of code, and a qualitative assessment of the proportion of generated code of the function, method, or class ("whole unit", "more than 50% of the unit", "about 50% of the unit", "less than 50% of the unit", "less than 10% of the unit").

2.7 *Workload*

FAQ: What is the expected workload for this module?

As a rule of thumb, the College suggests that 1 credit in a course should correspond to 10 hours work. In other words, this module should require approximately 300 hours of work. This guideline ignores some very important factors, such as your prior earning, innate ability, productivity, and your ambitions. Nevertheless, this section will use that guideline to advise you how you should expect to allocate your time to this module. Please adjust the 300 hour headline figure to your individual requirements.

Normally, you can distribute your work evenly between Semesters 1 and 2. In other words, you will need approximately 150 hours per Semester. If programming (e.g. as taught in 4CCS1PPA), data structures, and database systems were modules you struggled with in

Year 1, or if you struggle with the computer literacy skills the module requires, then you should dedicate more time on this module in Semester 1. If you choose to join a very ambitious team for the major group project, you should dedicate more time to the module in Semester 2. 150 hours per Semester is a considerable amount of time. You should aim to distribute that time evenly across approximately 10 to 11 weeks of each Semester (i.e. 13.6 to 15 hours per week).

EXPECTATION: Aim to dedicate about 14 to 15 hours per week to this module during term time. The group projects in particular will require require consistent engagement throughout the project (not short bursts of activity).

Table 2.1 lists the major activities of this module and identifies an approximate workload for each. Details for the small and major group projects have been omitted as these are discussed in more depth below.

Table 2.1: SEG workload distribution (times may vary depending on individual circumstances).

Activity	Total workload
Live lectures	16 hours
Software installation and setup	2 hours
Studying assignments and handbooks (KEATS)	4 hours
Small group tutorials	8 hours
Independent study (Python/Django, devops, project management & software engineering)	60 hours
Small group project	60 hours
Major group project	148 hours
Peer assessments	2 hours

2.8 *Schedule*

FAQ: What are the deadlines for this module?

A schedule of deadlines (i.e. dates by which you submit something) and publication dates (i.e. dates by which we will release information, such as feedback), has been specified and published on KEATS. At the time of writing, these dates have not been confirmed yet by the Department, so they may change. To ensure you have access to a single source, the schedule for this module is published on KEATS under Module organisation > Schedule.

3
Tools

As in any engineering discipline, software engineering is heavily reliant on tools. The general trend in software engineering is one of increasing availability of increasingly sophisticated tools. The most recent development is the emergence of generative AI tools (which you will be encouraged to employ, responsibly).

FAQ: What is the role of software tools in this module?

In this module, we make extensive use of a wide range of software engineering tools. All the tools we use are tools that regular, professional develops use: no "training wheels" Integrated Development Environment (IDE) (such as BlueJ) will be provided to shield you from some of the complexity of software development. There are certain tools that you must use and others where you are allowed to select from a range of tools. Some tools are essential to enable collaborative software development in teams. It is also important that you ensure your individual development environment is compatible with that of your team mates. In this section, we will identify what tools/type of tools you need to use, and make some recommendations for specific tool categories where you get a choice.

As the module starts (from the very start of the academic year), you must ensure that you have regular access to a workstation that has the software you need installed. This can be a Faculty lab PC, a Faculty student VM (a virtual machine you access from a web browser), or your own laptop or desktop PC. It is critical that you do not procrastinate organising your workstation and access to the necessary tools: doing so would delay your learning and ability to engage with the group projects.

3.1 Overiew

Table 3.1 summarises the essential tools you need to engage with this module. The table also provides suggestions for accessing/installing these tools on Linux, MacOS, or Windows. Although there will be other packages to install, this should be straightforward once the tools in Table 3.1 are installed correctly. In the Semester 2 major group project, your team is allowed to work with the technology stack they choose, so more software installation may be required then.

FAQ: What software tools do I need in this module?

Table 3.1: Essential software requirements in Semester 1, with suggestions for the three most popular operating systems.
(*) These applications are IDEs rather than lightweight code editors, but they can be used as a code editor only.

Tool	Linux	macOS	Windows
UNIX shell	Terminal	Terminal	Windows Subsystem for Linux (WSL)
Lightweight code editor	Sublime, Zed, GNU Emacs, Vim, Bluefish, Visual Studio Code (*), Pycharm (*)	Sublime, Zed, GNU Emacs, Vim, Textmate, Bluefish, Visual Studio Code (*), Pycharm (*)	Sublime, Notepad++, Visual Studio Code, GNU Emacs, Vim, Bluefish, Visual Studio Code (*), Pycharm (*)
Python 3 ($\geq$ 3.10) with PIP	Install with package manager (apt, yum, dnf, ...) or from official tarball	Install with Python's official installer or via Homebrew	Install with Python's official installer
Git	Install with package manager (apt, yum, dnf, ...) if necessary	Install with XCode command-line tools	Download and install from the Git website
Web browser	Use any up-to-date version of a current web browser.		
Calendar	Office 365 or a calendar of your choice.		
GitHub	Use your webbrowser.		
PythonAnywhere	Use your webbrowser and UNIX CLI skills.		
Team Feedback	Use your webbrowser.		

3.2 *UNIX CLI*

FAQ: Do we have to use/know how to use a UNIX command-line interface (CLI) or Terminal?

EXPECTATION: Some proficiency in using a CLI is a valuable skill for any Computer Scientist. It is best learned by using it routinely for file management, version control, and accessing dev ops tools. If this a new skill, make sure you always have some type of cheat sheet to hand. There is no need to memorise commands.

A UNIX shell provides a CLI to your operating system. It allows you to access system administration commands that come with the operating system, and software tools using a text-interface. At first, this may seem like an old-fashioned way of interacting with your computer. However, it is by far the most flexible and efficient way to work with many system administration and software development tools. UNIX' CLI enables instructions with a wide range of arguments. Different commands can be combined with one another, for example by feeding the output of one command as the input of another. Moreover, when access certain systems, such as servers, remotely, a CLI may be the only interface available to you. Therefore, familiarity with the UNIX CLI is an essential skill for software

engineers. Linux and macOS come with UNIX shells through the Terminal application. On Windows, you need to install the Windows Subsystem for Linux (WSL) to have access to a UNIX shell.

Most of the time, a UNIX CLI is an incredibly convenient tools for a proficient power user, but not essential. However, there will be certain occasions where the team must use a UNIX CLI. One such occasion is deploying a web application in production. Every member of the team should be able to do this, so every student in this module is expected to develop some basic familiarity with the UNIX CLI

3.3 *Code editor*

A *lightweight code editor* focusses primarily on supporting code editing. This distinguishes it from an IDE, which combines all the software development tools a developer needs into a single application. Software engineers continuously develop tools that aim to improve productivity in software development through analysis, visualisation, and automation. An IDE encourages developers to use the set of tools supported through the IDE's interface. In the face of prolific growth of available software development tools, that can sometimes be constraining. More importantly, software produced with certain IDEs may be difficult to edit and review without access to the IDE, which may affect collaboration with teammates and assessment. A lightweight code editor does not come with such baggage and it pairs well with a UNIX shell.

FAQ: What code editor or integrated development environment (IDE) should we use? Is using an integrated development environment (IDE) allowed?

My advice to you is that you should use a lightweight code editor (in combination with a UNIX shell). You can use any code editor you like. You are *not* prohibited from using an IDE. However, if you *choose* to use an IDE, *you are responsible for the consequences of that choice*. In particular:

- You must ensure that the IDE plays nice with the version control tools you are using. An IDE may generate a number of files that should not be version controlled. Failing to exclude such files from a team's repository may cause your code contributions not to be recognised. Consider yourself forewarned that this is not something that can or should normally be corrected.
- If multiple team members are using IDEs, failing to exclude IDE related data from a shared repository can cause spurious merge conflicts.
- *The examiners will not be using any IDEs.* They must be able to install, run, and test your team's software without an IDE. If they are unable to do so, because your source code is dependent on the IDE, your team may fail the coursework.
- It may be challenging to use the IDE of your choice in combination with the other tools you need to use. Time spent towards setting up an IDE is especially productive.

Therefore, you should only use an IDE if you know what you are doing. In general, it is only worth doing if you are confident that using the IDE ensures you are more productive.

Table 3.1 suggests some code editors and some IDEs that can be used as code editors.[1]

[1] Personally, I prefer Sublime. Although it is not free, you can evaluate it without time limit. In the videos, I use Sublime or Atom. However, Atom is no longer recommended as it is not being developed anymore.

3.4 Python

FAQ: What version of Python and Django do I need

In the small group project, you are required to use Python and Django. To develop *Python* applications and *Django* web applications, you must install a recent version of Python 3. This comes with Python's package manager pip. At the time of writing, I recommend that you use Python 3.12. While there are more recent versions, there are still a substantial number of packages that are not yet compatible with Python 3.13. As you start building Django applications, additional packages need to be installed. We will install these with pip as and when needed. The most default versions pip chooses will do.

3.5 Version control

FAQ: What version control tools do we use in this module?

Version control software is an essential tool to manage your code base. It is especially important when multiple people are working on the same codebase (ideally on different parts of it) simultaneously. In this module, we will be using *Git* for version control along with GitHub to maintain a shared code repository for the entire team.

Git needs to be installed on your workstation. It may already be installed. On a UNIX machine (including macOS), you can check whether git is installed by opening a UNIX shell/Terminal window and executing the command:

```
$ which git
```

If Git is installed, this will return the directory where the executable resides.

GitHub is a web service to maintain a remote repository. Figure 3.1 explains the relation between Git and GitHub, from the perspective of a software developer working in a team. Each workstation maintains a *local repository* that is managed via Git. In addition, a GitHub server stores a *remote repository*. Every time you "commit" code to their local repository, you should also "push" that commit to the remote repository on GitHub. Other team members can then see that you have done work, and review your work on GitHub. If they want the most up-to-date versions of the code on their workstation, they can "pull" the remote repository.

3.5.1 GitHub.com and GitHub.kcl.ac.uk

FAQ: Which version of GitHub should I use?

In this module, you can use either GitHub's own GitHub service at github.com or King's College London's GitHub Enterprise service.

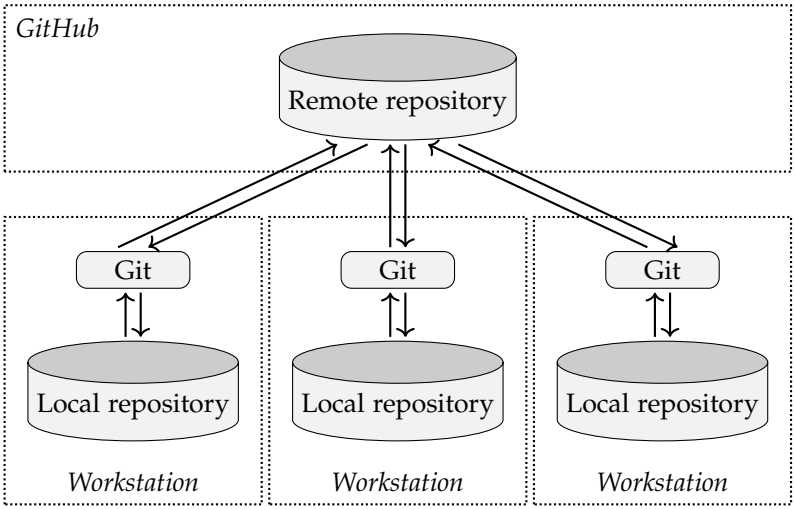

Figure 3.1: GitHub and git

As each team will only maintain a single remote repository, you need to choose as a team which GitHub service to use.

GitHub.com tends to provide a more up-to-date range of features, compared to the Enterprise version. However, to access all of GitHub.com's features, you need a pro account. You can get this for free as a student, but there is a verification process to follow. King's College London's GitHub Enterprise service can be accessed using your College credentials. No signup or registration is required. At the time of writing, both services are GDPR compliant, though GitHub.com data is stored on US servers. Both services are very reliable though GitHub.com's uptime has been slightly better in recent years.

3.5.2 Git/GitHub configuration

When using Git and GitHub in a team, it is important to do some additional configuration of Git on your workstation to ensure that your commits are attribute to you. Your code contribution data will be collected from GitHub with a view to assess whether you contributed sufficiently regularly and substantially to your team's code base. It is your responsibility to ensure that your workstation is configured correctly!

FAQ: How do I configure git to assign the correct email address to my commits?

When you *make* a commit, the name and email address of the "author" of the commit is attached to that commit. This name and email address are retrieved from the workstation's git configuration, and it is fixed at the time the commit command is executed. The name does not matter, but the email address is important. When the commit is pushed to GitHub, GitHub attributes a GitHub account as the author of the commit using that email address. For that attribution to be possible and correct, you must ensure that your workstation is configured to assign the primary email address of your GitHub account to all commits made on that workstation.

EXPECTATION: You must make sure that your work is attributed to you on Team Feedback before the deadline of the project. If code is not attributed to you by the time individual marking is carried out, your mark may be affected.

To find the primary email address of your GitHub account, open GitHub in your webbrowser. The click on your avatar (the image

in the top righthand corner of your screen. A menu opens: choose "Settings". This leads to a new screen with a large menu on the lefthand side. Open "Emails" (under Access). This will show your primary email address. Your workstation must be configured to assign that email address to all the commits you make.

To assign a particular name and email address to each commit, we recommend two approaches. The first is a global configuration that assigns the same name and email to all commits in all git repositories on this workstation. To set a global configuration, open a UNIX shell and use the following commands (replacing "Charlie Doe" and "charliedoe@example.com" by your name and email address:

```
$ git config --global user.name "Charlie Doe"
$ git config --global user.email charliedoe@example.com
```

The second approach is a local configuration that assigns a given name and email to each commit within a single repository. To do this, use the following commands:

```
$ git config --local user.name "Charlie Doe"
$ git config --local user.email charliedoe@example.com
```

To verify your configuration, use the following command:

```
$ git config --list
```

If you only have a single GitHub account or all your GitHub accounts (on GitHub.com and GitHub Enterprise) use the same primary email address, you can use a global configuration. If you have GitHub accounts with different email addresses, you will need to use local configurations and must not forget to set your configuration at the start of the project. The latter approach is obviously a little less foolproof. Therefore, I recommend that you do the following at the start of the academic year:

1. Log into github.kcl.ac.uk and retrieve the primary email address of that account. Look it up, do not assume you know what it is.
2. If you do not already have a github.com account with the same primary email address, create a new github.com account using the same primary email address.

This will enable you to use a single configuration for your work, irrespective of which GitHub service your team chooses to use.

3.5.3 Rules of engagement

FAQ: How should the team be using version control tools? What is allowed and what is prohibited?

In this module, version control tools serve a dual purpose. First and foremost, these tools are used for collaborative software development. In this capacity, they allow team members to share their work, review other people's work, and integrate developments into the teams emerging system as and when they are produced and

quality controlled. Teams and team members must adhere to certain rules of engagement when using version tools. These include, but are not limited to:

- As team members work on a task, they must commit and push their work regularly with clear commit messages.
- When working on a task, team members should commit and push at least once a day so that team members can observe your activity.
- All commits should occur in branches. Create one branch per task.
- The team must agree on standards that must be met before a branch can be merged with the main. The team must also agree a process to follow to ensure these standards are met before the branch is integrated into the main. All team members must follow this process.

A secondary purpose of the version control tools is to maintain an accurate record of the development effort. In this capacity, the tools are used to collate statistics on the nature and regularity of code contributions, and to identify to what extent team members produce usable work. Once made, all commits must be preserved as they are. Branches must never be deleted, even if the work in it cannot be merged with the main. The commit history must never be rewritten or changed in any way. Problems should be corrected with new commits.

If you have serious concerns about a team member's version control behaviour, contact the module organiser. Serious concerns may include fake coding activity to create a pretence of engagement when there was none, or rewriting the commit history. When reporting a concern, always include full details of the sha, date, and message of each commit that concerns you. Concerns must be raised as soon as possible, normally before the submission deadline. The longer you wait to raise an issue, the less likely it can be resolved.

3.6 Team Feedback

FAQ: What is Team Feedback?

The only tool used in this module that professional developers do not require is *Team Feedback*. Team Feedback is a system developed to manage SEG. It collates all information lecturers need to track teams and team members in one place. You will use it to form teams (or receive your team allocation in case of the small group project), record meetings (including attendance and minutes), track code contribution statistics, submit, receive and respond to peer assessments, record collaborative coding sessions, and receive your team and individual marks. This section explains the role of these various features in more detail.

3.6.1 *Account data*

FAQ: How can I get access to Team Feedback?

Registration If you are registered to take the Software Engineering Group Project at the start of the current academic year, then a Team Feedback account will be created for you (assuming you do not already have one). You can access the service at https://apps.nms.kcl.ac.uk/stf, and log in via your regular College credentials.

If you have used Team Feedback before, you will normally be directed to the previous module you participated in. Set a new module by navigating to "Module" > "Select module". A screen with the modules you are registered to will appear. Select this year's SEG module and click the "Select module" button.

If you do not have an account or you are not registered for the module in this academic year, you must contact the module organiser as soon as possible. Unfortunately, it takes several weeks from the start of the academic year before class lists are accurate and complete, so it is possible you are not registered from the module at the start of the academic year.

FAQ: Do I need to register my gender identity, ethnicity, and disability?

Sensitive data Team Feedback will encourage you to provide your gender identity, ethnicity, and disability. You can provide this data under "Account" > "Profile", but you are not required to do this. If you provide your gender identity and ethnicity, it is used *exclusively* to produce statistics that compare marks across different gender identities, ethnicities, and people with/without disabilities. This allows the module organisers to identify any attainment gaps, should they arise. We aim to ensure that all students have equal opportunities to excel in this module, irrespective of their gender identity, ethnicity, or disability. By proactively collecting data that aims to reveal attainment gaps, we can assess to what extent our efforts are successful.

3.6.2 *Team formation*

FAQ: Where can I find the team I am allocated to or register the team I want to be part of?

Team formation Team formation is handled via largely via Team Feedback.

Before the small group project, you must register to be allocated to a small group project team using a form available via KEATS. This form will become available several weeks into Semester 1. Team allocations will be released via Team Feedback, where you will find the names and contact details of your team mates.

Before the major group projects, all students are strongly encouraged to form their own teams. Once your team starts to take shape, one team member should register the team and *invite* the other members to this team. The invitees *must accept the invitation* via Team Feedback if they wish to be a member of the team! Aim to form a team of approximately 8 people (teams of 6–10 students will normally be allowed to proceed without changes to the allocation). Teams with 5 or fewer members will have additional people

allocated. Teams with 11 or more members will be split. If you are unable to join a student-formed team before the team registration deadline for the major group project, you must register to be allocated to a team using a form available via KEATS. Final team allocations will be released via Team Feedback.

3.6.3 *Meeting scheduling*

FAQ: Do we have to schedule our meetings via Team Feedback?

Team are *not* required to schedule team meetings via Team Feedback. However, you can use Team Feedback to find available meetings slots (this was a useful feature before similar functionality was incorporated into Office 365) and create a meeting invite to all team members. Feel free to use this feature if you find it useful.

3.6.4 *Meeting records*

FAQ: How can I record a team meeting on Team Feedback?

How? Open the submenu of the relevant group project in the left-hand menu, and navigate to "Team meetings". It is *not* necessary for the meeting to have been scheduled in order to record the meeting. To start recording a new meeting, click the "+ New meeting" button. At the start, you must record attendance for every member of the team. You cannot proceed to create the meeting and edit the minutes until attendance has been taken.

Once a meeting is created, a record of the meeting will be listed in a table, along with a deadline to complete editing. The meeting will only remain editable until the time shown there. This ensures that edits are timely corrections (not a rewrite of the team's history). While this is possible, click the edit button to edit the meeting or the delete button to delete the meeting record if it was created by accident.

FAQ: Who must record the team's meeting on Team Feedback?

Who? The purpose of the meeting is to have a written record of team member engagement, decisions taken, and actions assigned. Therefore, each meeting must be recorded by *one and only one person*. You must agree within the team who is responsible for recording the meeting. It is worth acknowledging that at the meeting itself, so that the meeting is recorded and there are no duplicate records of the same meeting.

FAQ: When should a team meeting be recorded?

When? A team meeting must be recorded when the meeting takes place. The best approach is for the minute taker to write the meeting minutes on a laptop during the meeting, and polish them after the meeting. Create a meeting record at the very start of the meeting. The first step is to record attendance. Start by recording everyone there as "on time" and everyone who is absent as "absent". You can correct for late arrivals later. Start writing notes under the meeting minutes/summary as the discussion proceeds. After the meeting, take some time to polish the minutes so that they will be understandable in the future or by teaching staff reading your minutes.

Some time after the minutes were created, team members will have access to the minutes. They should check the attendance record and minutes and send corrections in the days that follow the meeting (up to one week after the meeting took place). The minute taker should make corrections as necessary during this period. If nobody raises any issues and the edit window ends, the minutes will be considered correct and no more editing will be possible.

FAQ: Which meetings do we have to record via Team Feedback?

Scope? Teams must record their *team coordination/planning meetings or accountability sessions* via Team Feedback. In other words, the short meetings with the *whole team* where you *make decisions* about the direction of the project *must be recorded via Team Feedback*. Subgroup meetings (i.e. meetings not everyone is required to attend), collaborative coding sessions, or code review/inspection meetings (where only minor decisions are made) must *not* be recorded.

Please note that the team should have at least *one team coordination/planning meeting or accountability session per week*. Meeting less frequently than will limit the team's ability to respond to risks and changing circumstances. Meeting more frequently that once a week may be difficult to sustain over longer periods of time (though you are welcome to do so).

FAQ: Should a team meeting be in person, online, or hybrid?

Mode of attendance Teams should decide internally what "attendance" means. I recommend that the weekly meeting a team record on Team Feedback is *in-person*. This approach has several advantages. An in-person meeting requires a more significant commitment to the project than an online meeting. To attend such a meeting on-time and in-full requires some effort. In-person attendance encourages greater engagement with the meeting. If your meeting is online, someone can "attend" by phoning into the meeting while on a bus or train, but their focus is likely to be elsewhere. At an in-person meeting, it is also easier to share information. Hybrid meetings should be avoided. They split the group between the online presence and the in-person presence and makes communication that much harder.

Once the mode of attendance has been agreed, stick to the rules (until your team management approaches are reviewed). Once you start bending the rules, it will become increasingly difficult to enforce them. Consider, for example, a team that normally meets in-person. One person calls in asking to participate in the meeting online. If you allow this, some of the people present will regret the effort they made to travel to the meeting, and feel entitled to ask for the same accommodation at a future meeting.

FAQ: How important are Team Feedback meeting attendance records?

Attendance records It is critically important that team members attend all meetings. Communication and coordination become much harder if some people do not attend meetings, but expect to be informed of meeting outcomes or share their views. To prevent such complications, teams are advised not accommodate absences.

To ensure that team members attend meetings, each team must keep accurate records of meeting attendance via Team Feedback. Accurate attendance record keeping incentivises team members to attend meetings in full and helps the marking team identify students who do not engage adequately with their team. Do not be tempted to mark someone as attending if they abstain from the meeting. If you do this for one person, it would be unfair to refuse to do this for someone else. Without accurate attendance records, it will be difficult to demonstrate later on that a team member did not engage adequately with the team. For these reasons, teams that fail to keep accurate attendance records from the start of the project end up regretting this.

FAQ: Do meetings require an agenda?

Agenda Effective meetings tend to have an agenda. An agenda gives attendees forewarning of what will be discussed and what they need to prepare. The agenda should normally include the discussion topics, an indication of the expected outcome of each topic, the person or people responsible for leading the discussion, and a time budget. During the meeting, stick to the agenda and timings as much as possible.

FAQ: How important are Team Feedback meeting minutes?

Minutes Over time, people tend to forget or misremember what was agreed at team meetings and why. Clear and complete meeting minutes are part of the team's "memory", allowing everyone to review the teams' decisions. The meeting minutes should include all:

- *Agreed decisions and their rationale*. Make sure that the decisions that have been agreed are written up sufficiently clearly to allow future you to make sense of the minutes. Recording the rationale for a decisions allows team members to understand why the team made a particular decision at the time. If a rationale turned out to be invalid (or no longer valid), that may constitute a grounds for reviewing the decisions.
- *Actions*, including the person responsible for completing them and the deadline for completion. The team should keep track of its action log, so as to ensure that actions are completed.

To record good meeting minutes, someone should be assigned responsibility for completing them. Ideally, the meeting minutes write up starts during the meeting and is completed as possible thereafter. Allow team members are reasonable opportunity to submit corrections.

3.6.5 GitHub activity data

FAQ: Do we have to register shared GitHub repositories on Team Feedback?

Repository registration Every team should register *all* shared repositories worked on by the team on Team Feedback. Team Feedback will collect commit data from registered repositories on a daily

basis until the end of the project. If your git repository is set up correctly, GitHub will be able to attribute commits to GitHub accounts. Team feedback uses this to produce engagement statistics for every team member.

How you organise your work in repositories is up to your team. Avoid using more repositories than necessary. In the small group project, one repository should suffice. In the major group project, you probably want to separate the report from the source code. If the software system you are building consists of distinct components that run on different machines, you may or may not store them in separate repositories.

FAQ: How is the code contribution data on Team Feedback used?

Code contribution data The code contribution data is used to assess whether:

1. A team member makes substantial code contributions sufficiently regularly. To assess this, we examine the number of calendar weeks in which a team member contributed a sufficient number of lines of code.
2. Enough of a team member's code is of a usable standard. To assess this, we examine how much of a team member's code is merged with the main branch.

In other words, code contribution data is collected to assess to what extent each team member makes a reasonable effort to engage meaningfully with the team's work throughout the development period. The small group project and major group project chapters in this handbook provide more detailed guidelines as to how this is assessed.

Team Feedback statistics include total lines of code, lines of code in the main branch, total numbers of commits, and commits in the main branch, relative to other members of the team. This data is provided for information only, to give you an idea as to what other members of the team do. However, it is impossible for the marks to make reliable judgements based on such data. Line count data is a notoriously unreliable type of statistics to measure code contributions. Line count data correlates very poorly to effort or code quality. Therefore, we cannot use such statistics, certainly not to determine who did more work or who did less work.

FAQ: Team Feedback is missing some of my code contribution data. Is that a problem?

Missing data Team Feedback should have a record of all your coding activity. If it does not, that is a problem that you must take steps to correct as soon as possible. Before contacting anyone, it is advisable to identify what the problem is. The most probable causes (from most likely to least likely) are as follows:

- You did not configure git correctly on the (or one of the) workstation(s) you are using.
- The missing coding activity occurred less than 24 hours ago and has not been recorded yet.

- You collaborated on this commit with someone else who has not registered the collaborative coding session correctly.
- A team member has altered the commit history of the project and erased some of your coding activity.
- The Team Feedback server has been down and its records are out of date.

To diagnose the problem, you should start by identifying the commits that you believe to be missing from Team Feedback. Record their sha, date/time, and message in a list. For each such commit, look up that commit in the *team's* commit history (*not* your individual commit history) on Team Feedback. Does Team Feedback have a record of the commit?

- If Team Feedback does have the commit on record, there are two possible reasons why the commit is not recorded:
  - If you made the commit and the commit is marked in yellow on Team Feedback, Team Feedback was unable to attribute the commit to you. The problem here is that you did not set up git correctly and GitHub was also unable to attribute the commit to your GitHub account. The commit will be associated with a, usually random looking, email address. In Team Feedback, navigate to "Account" > "Profile". Find the heading "Additional email addresses" and click on "+ Add additional email". Use this feature to add the email address of the unattributed commit. The missing commit will now be attributed to you within 24 hours. It is also advisable to read the handbook to find out how you should be configuring git.
  - If another team member made the commit but you contributed to it, the person making the commit should have registered the collaborative coding session. Contact the committer and ask them to do this as soon as possible.
- If Team Feedback does *not* have the commit on record, check the following.
  - If the commit is less than 24 hours old, wait.
  - Below the list of commits (by the team!), there should be a date indicating when the list was last updated. If this date is more than 24 hours ago and precedes the date/time of your commit, Team Feedback as been unable to collect commit data for your team's repository. Check the module announcements on KEATS to check whether this is a known technical issue, and contact the module organiser if it is not. Do not worry, if your commits are in GitHub, the issue will be corrected eventually.
  - Check if the commit can be found in the shared repository on GitHub. If it is not on GitHub, a team member may have altered the commit history for your team in violation of the rules set out in this handbook.

FAQ: I suspect a member of my team has faked code contribution data. What can be done about this?

Falsified code activity There are ways in which people can create the impression of having made contributions to the code without doing any meaningful work. One approach is to move code around unnecessarily and committing this change. Another is to add unnecessary code, commit the change, remove the unnecessary code again, and commit that change. Doing this type of thing with a view to create the impression of coding activity without the intent of contributing anything constitutes falsification of data. It is a form of misconduct.

If anyone is caught doing this, we will normally remove the offending commits from the code contribution statistics and send a warning the offender against doing this again. In very flagrant cases or repeat offences following a warning, the case will be referred to the College for formal misconduct proceedings.

Please report falsified code activity to the module organiser. Your message must include a list of the offending commits, including their sha, date/time, and message, as well as the reason why you believe this evidence demonstrates data falsification.

3.6.6 Pair programming and collaborative coding sessions

FAQ: How do I get credit for code produced through pair or mob programming and someone else made the commits?

Some teams may wish to tackle certain tasks to two or more people, to be completed through pair or mob programming. If such an approach is adopted, two or more people code using a single machine. One person acts as the driver and types in the code. The other person or people act as navigator(s), guiding the driver. If you adopt this particular approach, you must ensure that the work is attributed to the driver and the navigator(s). To ensure this, you must adopt the following protocol:

- During a pair/mob programming session, *all* commits should be made using a single machine. The author this machine's git installation is configured for is the *committer*. In this way, all work produced collaboratively is attributed to a single author: the committer.
- As soon as possible after the session, the committer must register a *collaborative coding session* on Team Feedback. When creating the session, ensure that the start time precedes the start time of the session and the end time succeeds the end time of the session. Identify all team members involved in the session.
- If this is done correctly, all commits made during the collaborative coding session by the committer will be attributed to everyone identified as a member of the collaborative coding session. Other members of the team should check that this is so and contact the committer if this is not the case as soon as possible.

To register a collaborative coding session, open the submenu of the relevant group project in the left-hand menu. Navigate to "Code repositories" > "Collaborations" (tab). Then click the button labelled

"+ New collaboration". It is important the collaborative coding session is created by the committer. Other members of the team cannot create it (to avoid abuse of the system).

If multiple people collaborate on the same task using their own respective machine, you are simply subdividing a task into smaller ones to be completed individually. This is not pair or mob programming. Collaborative coding sessions should only be created to record genuine pair/mob programming sessions!

3.6.7 Trello board data

FAQ: Do we have to register a Trello board on Team Feedback?

In the small group project, you should use Trello and register your team's Trello board on Team Feedback. This allows the module organiser to track your team's project management activity. In the major group project, teams are not required to use Trello.

To register your team's Trello board, open the submenu of the relevant group project in the left-hand menu, and navigate to "Trello Kanban board".

3.6.8 Peer assessments

FAQ: How can I participate in the peer assessment exercise?

Near the end of each group project, you must participate in a peer assessment exercise. The peer assessments generally consist of a questionnaire administered via Team Feedback. For more information on how to write good feedback, please review section 4.5.

To start writing peer assessments, open the submenu of the relevant group project in the left-hand menu, and navigate to "Peer assessments". This will open a screen with options to read/write peer assessments, provided the deadline for late submission has not passed.

After the peer assessments have been released, you can find the feedback you received on Team Feedback as well. Open the submenu of the relevant group project in the left-hand menu, and navigate to "Peer feedback". If there is still time to respond to feedback, you will have an option to write a response to your peer's feedback. Responses to your peer assessments can be found under "Peer assessments".

3.6.9 Marks and feedback

FAQ: How can I find out my mark and feedback on a group project?

Marks and feedback are released via Team Feedback. For each group project, you will receive three marks along with feedback related to that mark:

- *Team mark/feedback*: This is a mark along with feedback based on what the team collectively submitted. For more information on how the team's work is assessed, please review the Small and Major Group Project chapters as these contain a detailed marking scheme. Your team's work is assessed using this mark scheme and the feedback explains how the mark scheme applies your team's work.

- *Individual mark/feedback*: Your individual mark is derived from the team's mark, either through a major correction (always a significant reduction) or a minor mark distribution (a change in the -5 to +5 percentage point range). The Small and Major Group Project chapters contain detailed individual marking schemes. Individual feedback explains how this mark scheme is applied.
- *Peer assessment mark/feedback*: The peer assessments you write are assessed and marked. The marking scheme for the peer assessments are also published in the Small and Major Group Project chapters of this handbook.

Once released, you can find the marks and feedback as follows. Open the submenu of the relevant group project in the left-hand menu, and navigate to "Dashboard". Once the marks are released, the dashboard includes a new tab labelled "Team/individual" mark where you can find the above three items. Please note that in some circumstances, partial marks may be delayed.

3.7 *Overleaf*

FAQ: What tool should we use to produce the major group project report? Do we have to use LaTeX?

For the major group project, teams must produce a substantial report. This documents ought to be produced using LaTeX.

LaTeX is a typesetting system that is widely used in academia to produce high-quality documents. Organisations outside academia prefer a WYSIWYG editor, such as Word, to produce documents. LaTeXuses a markup language to specify what must be typeset. The LaTeXsource code in a `.tex` file is compiled to produce a PDF document (or some other type of document). The use of a markup language makes it easy to automate the generation of equations, cross-references, tables of content, indices etc. The language allows you to define custom commands you may wish to repeat, declare variables, and (in the rare event that you need to) use conventional programming structures. This document has been produced entirely with LaTeXand everything, including the figures, is generated from code. Although you may not end up using it very much, knowing LaTeXis a useful skill to have. You will be encouraged to use it for your individual project in Year 3.

EXPECTATION: The report that must be submitted with the major group project ought to be typeset with LaTeX, using version control to keep track of developments.

A LaTeXsource document can be version controlled with git because it is conventional source code. This facilitates collaborative authoring of the document. It also allows us to track who is working on the document and when. In each group project, you will be provided with a template to use to write your report. This will allow you to focus on writing the text for the document, without having to worry about the markup language too much. In the major group project, this report document must be version controlled with git, with remote repository on GitHub that is tracked from Team Feedback.

Overleaf is a web application that allows you to edit LaTeXdocuments. King's College London has a subscription to the service, you have

access to the premium features of the application.

3.8 Generic productivity tools

FAQ: Which productivity tools do we require?

The only productivity tools you absolutely need in this module are an up-to-date webbrowser and a Calendar application. will need an up-to-date web browser to access documentation and browser based tools.

Last but not least, you should use a *calendar* application and ensure it contains all your commitments, at university and outside it. I strongly recommend that you use *Office 365* with your College account. Normally, members of the College (such as your teammates) can see when you are available and when you have other engagements. They will *not* see the content of your appointments, unless you give explicit permission for that. Easy access to everyone's availability makes scheduling meetings, especially group meetings, considerably easier.

4
General expectations

4.1 First week of teaching

FAQ: What should I be doing in the first week of teaching?

As a coursework only module, you need to dedicate a substantial amount of time to the SEG during term time. The module has been designed to ensure you can dedicate as much time as you want early on in the module, so there is no excuse to delay your engagement with the module. In fact, early engagement with the module is crucial.

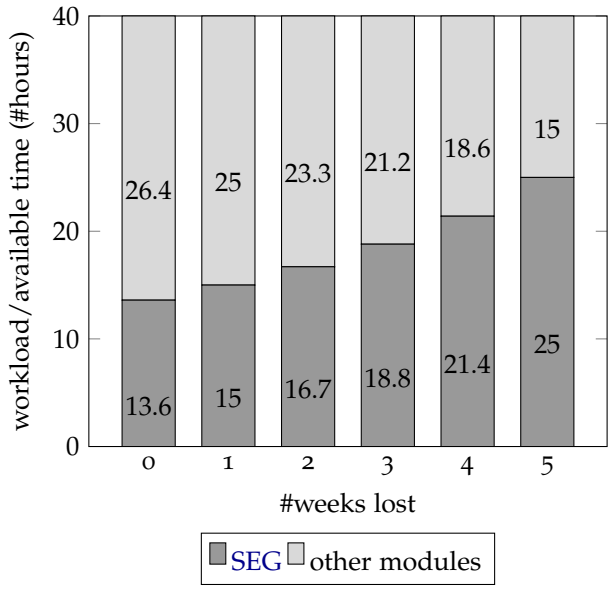

Figure 4.1: SEG workload and time available to work on other modules (expressed in #hours) in relation to the number of weeks no work is done for this module.

Figure 4.1 plots the weekly workload for this module relative the number of weeks you do not or cannot engage with it. As you can see, the SEG workload increases substantially as engagement with it is delayed. Figure 4.1 also shows the amount of time left to work on other modules, assuming a 40 hour work week. Bear in mind that, in Year 2, you have another coursework-only module with similar workload requirements to this one. The time available for that and other modules diminishes quickly.

In the first week of teaching, you should aim to achieve the following:

- Ensure you have access to a development environment with a UNIX command-line interface, Python 3 with pip, and git. This is a means to start learning Python and Django, so do not waste too much time on this. If you find it very difficult to install this software on your desktop or laptop PC, use a Faculty lab machine or student VM, rather than postponing engagement with the course.
- Learn all or most of the Python course, completing the exercises as you encounter them.
- Complete the version control for individuals course.
- Attend the first lecture.

4.2 Independent study

FAQ: What is the "independent study" component of the module?

In Semester 1, you will be learning to program in Python and develop web application with Django, which is a framework developed in Python. You will also learn to use certain tools that support web development, including tools for build automation, automated testing, version control, and deployment of web applications. These topics require you to develop new practical knowledge and skills. The amount of new theoretical or conceptual content, beyond what you already learned in Year 1, is limited.

In my view, such content is best learned outside of lecture theatre. One effective approach is to learn content in small bursts, followed immediately by exercises. Another is to follow along with a demonstration on your own computer, mimicking each step that is demonstrated. In this module, you will be using both these approaches to learn Python, Django, and a range of devops tools. There will be some forms of support to help you out when you get stuck, but you should be going through this material as independently as you can.

In your independent study, consider the following:

- In the first five weeks of Semester 1, before the group project start, you should dedicate a substantial majority of your weekly study time for this module (approximately 15hrs/week) to independent study. Do not postpone this as it will be hard to catch up later.
- *Complete all exercises*. When shown demonstrations, replicate each demonstration on your workstation. The independent study material is very practical and merely watching the videos is not sufficient to learn it.
- Encountering *bugs, errors, and unwanted exceptions* are an important part of the learning process. It can be frustrating to lose hours on attempts to overcome such problems. But don't consider them a waste of time. Resolving bugs when attempting to replicate a demonstration or solving exercises helps you identify

and overcome misconceptions. It also also develops your debugging skills.

- Work through the material at your own pace. Speeding through material you do not already know in a misguided attempt to get ahead will make the later material harder to digest. *Everyone learns certain material at a different pace*. Allow yourself to set the pace that works for you. Of course, a timely start with independent learning helps.

- All independent study material is labelled as "Core", "Recommended", or "Optional". You need to complete the "Core" material before reading week in Semester 1 as it covers the essential knowledge and skills required to participate in the small group project. Knowledge and skills covered by the "Recommended" is referred to in some of the small group project marking criteria for 60-70%. Learn this only after you completed all "Core" material. The "Optional" material concerns more advanced, niche topics. You will need to be more independent to study this material. The "Optional" material is referenced in some of the small group project marking criteria for 80% and higher.

Software Engineering is a rapidly evolving field. New languages, frameworks, and tools are introduced regularly. Existing ones change all the time. As a software engineer, you will need to stay up-to-date with these developments. Therefore, it is important that you develop the skillset needed to learn new technology independently. The independent study materials develop these skills.

4.3 Tutorials

FAQ: What is the roles of large and small group tutorials? Do they cover different material from the independent study component?

Along with the independent study component, Semester 1 large and small group tutorials seek to prepare you for group project work. Beware that the independent study component, the large group tutorial, and small group tutorials serve different purposes and cover different material! You will need to engage with all these activities as one does not substitute another.

The *small group tutorials* aim to develop your practical skillset to manage, and participate constructively with fellow team members software engineering group projects. The activities are designed to put key elements of the software engineering and project management training videos into practice. it is advisable to engage with both concurrently.

The *large group tutorials* aim to complement all independent learning activities: Python/Django training, devops training, and software engineering and project management training. A session will consist of quizzes designed to identify potential misconceptions, a class discussion around project management case studies, and a Q&A activity.

4.3.1 Small group tutorials

FAQ: What will I learn in the small group tutorials?

Each small group tutorial starts with an informal icebreaker activity. This is intended to put everyone at ease before discussing certain topics in break-out groups. It also offers an opportunity to get to know the classmates in your tutorial group a bit better. Getting to know others is an important networking skill that will help you find team mates for the major group project.

A significant portion of each group project is dedicated to discussion, negotiation, and peer review in small break out groups. In these break-out groups, you will learn the most important skills and activities needed to coordinate work with teammates. The small group tutorial sessions include the following activities:

- Agreeing a *team charter* within a small group.
- Performing a *code review*.
- Estimating effort in a group through planning poker.
- *Risk management*indexrisk management:Identifying, analysing, and planning for uncertainty and risk.
- *Agile planning*planning!agile: Writing user stories, specifying a Kanban board process, and managing the development of a consistent user interface.

The end of each session includes a class discussion to reflect on how well the activity went and what you learned (e.g. what you would do again or what you would do differently).

4.3.2 Large group tutorials

FAQ: What will I learn in the large group tutorials?

As the large group tutorials aim to complement the independent learning/training videos, there is some flexibility as to what is covered. Your feedback will be considered and can lead to a change in plans, though I will also take into account what students in previous years needed support with throughout the module.

A typical session consists of four parts:

- *Announcements/guidance*: In this part, I will share important information with you. This information may include issues people have struggled with, changes to the information on KEATS, responses to feedback I received from you, and reminders of important information you should be aware of.
- *Quizzes/exercises*: The quizzes and exercises cover (mostly technical) Python, Django, and devops challenges based on misconceptions students and teams have faced in the past. The purpose of these exercises is to resolve and avoid these misconceptions, to that it will be easier for you to learn.
- *Class conversations*: The class conversations cover project management challenges based questions and challenges group project

teams have faced in previous years. You will be encourage to think about certain problems and share your opinions. These conversations are not intended to be debates. They are not intended to be "won" by any side. While I will share my views at the end of each class conversation, you are entitled to your own opinions. I hope that these conversations will help you learn to think in more nuanced terms about project management and leadership challenges.

- *Q&A*: Each session will conclude with some time for Q&A. Before each session, the large group tutorial section hosts a forum where you can upvote questions of others, or propose your own questions. The forum closes at 5pm the day before the session. The most upvoted questions will be prioritised in the session.

4.4 Working in groups

4.4.1 Core expectations

FAQ: What is expected of me in the group projects?

As explained earlier, the small and major group projects differ in scale and purpose. Consequently, each comes with its own set of expectations. Nevertheless, there is a core set of expectations for both group projects and it is crucial that you meet them. The expectations listed below are fundamental though, so you should adhere to them and expect you team mates to adhere to them.

FURTHER READING: Each group project comes with its own handbook, which you should read as you are participating in these projects. You find them on KEATS in *Small group project → Handbook: Small group project* and *Major group project → Handbook: Major group project* respectively.

- In the group projects, you will be working closely with fellow students. You are expected to *behave respectfully* to one another during lectures, outside of lectures, in meetings, when communicating online, through email, or through collaborative development tools. The College does not tolerate inappropriate or demeaning comments related to gender, gender identity and expression, sexual orientation, disability, physical appearance, race, religion, age, or any other personal characteristic. If you witness or experience any behaviour you are concerned about, please speak to someone about it.
- Group work requires *communication* and *coordination*. As a team, you must agree project objectives, plan and schedule work towards those objectives, monitor progress, and agree remedial actions when plans/schedules fail or other problems arise. Individual members of the team cannot do whatever they want: work on the tasks you you have been assigned and *only* on tasks you have been assigned. *Regular, minuted, whole-team meetings* are the primary method of communication and coordination.
- Team members must make every effort to *attend all meetings, on-time, from start to finish*. Engage actively with the meetings by adopting *all* of the following: listen to others, sharing your views, volunteer to take on tasks you are able to do, show the work that you have done, read the meeting minutes following

the meeting, and report any errors. Your team mates are not your personal assistant: do not expect your team mates to keep track of team decisions or your responsibilities for you. You need to do that yourself.

- Every member of the team must make a meaningful contribution to the team's code base every week of the software build period. That implies nobody is excused from coding! A *week* is defined as period from Monday to Sunday (inclusive). The project's *software build period* corresponds to the weeks in which at least some team members are writing code. In the small group project, the software build period should encompass the full six weeks of the project. In the major group project, it is likely to be a little less than the full ten week period of the project because of the amount of non-coding work involved.

4.5 Feedback and peer assessments

FAQ: How should give feedback to/write peer assessment for my teammmates?

As this is a module is about working with others, you will have to tell others how they are doing or how they did.

You will participate in two peer assessment exercises (via Team Feedback), one after each group project. In a peer assessment exercise, you are required to fill out a form for each of your team member: including a number of multiple choice questions and a feedback field. *The feedback text is a critical component of any peer assessment.* Without any feedback text, a peer assessment will not be awarded any marks. At the very least, explain why you scored your team mate in the way that you did in the multiple-choice section. Without any feedback, the peer review is merely a collection of vague claims. You can also use the feedback field to elaborate on your experiences with a team member, provide some detail and context for your review, illustrate claims with some examples or anecdotes, and reflect what you or your team might have learned in group project exercise.

If there is good communication within your team, the feedback in peer assessments should not come as a surprise. Team members should talk to each other about how the project is going. Concerns should be shared within the team: ideally at team meetings if it concerns everyone, or at a one-to-one meeting if a difficult conversation with a single person is required.

When you write feedback, *be considerate and inclusive*. Another fellow human being will hear what you are saying or read what you have written. Disparaging criticism without nuance or consideration of the reviewee can be hurtful – ultimately, it is counter productive. That does not mean your feedback should be relentlessly positive or minimise valid criticism. On the contrary, it is important to voice genuine concerns in a *timely* manner, and persistently follow it up. Peer assessments should contain an *accurate and complete critique*: do not gloss over the problem.

There are some techniques you can employ to formulate negative feedback considerately and *constructively*. Examples include, but are not limited to the following.

- *Balance negatives with specific, meaningful praise*. Criticism in feedback tends to be more acceptable to the recipient if the feedback also incorporates due recognition to positive experience. Be specific (not vague) when praising your peer in feedback to show you paid attention and appreciated what went well. Make sure that the praise concerns meaningful issues. Faint praise can come across as condescending, or even as criticism in disguise. Praise is only a usable tool if you have something of value to complement.

- You can take the previous method one step further with *the sandwich method*. In the sandwich method, you start by saying something positive about the other person, then introduce the negative feedback, and conclude with more positive feedback. This can work especially when you need to give negative feedback to someone in person. People do tend to recognise the sandwich method, so it is especially important that any positive feedback you want to use here is genuine.

- In your feedback, *focus on actions and behaviours*. Talk about facts, such as commitments made, deadlines missed, GitHub activity, meeting attendance. Our actions and behaviours are not necessarily representatives of our personality and character. We can change our actions and behaviours. It is harder to change our personality or character. If feedback concerns the person, rather than their actions and behaviours, then it becomes something that is difficult to correct or address.

- *Use sentences start with "I"*, such as "I think [...]" or "I feel [...]". By starting a sentence with "I", you are forcing yourself to focus on your perceptions and views of the other person's actions and behaviours. It is another way to avoid making the negative feedback about the person being criticises.

- Always ensure your negative feed back is *specific*. Vague feedback tends to be very demoralising. On the one hand, it suggests concerns are broader than they really are. On the other hand, vague feedback is hard to act on.

- *Provide actionable feedback*. This gives your peer agency and control over a problem, as well as an opportunity to address it.

Negative feedback is not the only tool in your arsenal to encourage your team mates to do the right thing. Negative feedback's counterpart – *positive reinforcement* – can be more effective if you make a point of praising people consistently and regularly for what they do well.

5
Project management

5.1 Introduction

A project is a significant undertaking aimed at achieving a specific set of objectives within a defined timeframe. In software development, building a system or application typically qualifies as a project: it represents a one-off effort that requires focused coordination, planning, and execution.

FAQ: Why is project management so important in this module?

Because projects involve a substantial commitment of resources, especially the time and expertise of skilled people, they must be managed carefully to ensure that this investment results in something valuable. Unlike routine operations, project work is often exploratory and uncertain, which means it carries more risk. Effective project management anticipates these uncertainties and prepares the team to respond when things do not go as planned.

At the heart of any project is a team. In software development, much of the work is creative, and relies heavily on collaboration among individuals with diverse skills and perspectives. For this reason, good project management is not just about tasks, timelines, or tools. It is about enabling people to work well together. This chapter offers practical guidance on how to manage student software projects effectively, with particular attention to teamwork, communication, and navigating the human factors that so often determine a project's success.

5.2 What makes project management difficult?

Managing software development projects is inherently challenging due to the complex interplay of technical and human factors. A substantial software project requires a significant amount of work distributed across multiple people. This collaborative effort must be tightly coordinated to ensure that everyone is aligned with the same goals, that features integrate smoothly, and that the overall product, particularly the user interface, presents a consistent and coherent experience.

FAQ: What makes project management difficult?

One of the key difficulties is that the quality of the software is determined by its weakest component. Even if most parts are well-engineered, a single poorly implemented module or lack of

sufficient testing can undermine the entire product. As such, project managers must ensure that quality is maintained across all dimensions: code, design, documentation, test coverage, and security, to name a few.

In addition to technical challenges, software teams are composed of individuals who may hold differing opinions on important issues, such as the project's direction, team workflows, or even the next steps to take. Disagreements over priorities, tools, or practices can lead to friction or misalignment if not managed carefully.

Ultimately, project management is about enabling people to work together effectively to achieve a shared outcome. While there are many structured techniques for planning, tracking progress, managing risk, and facilitating communication, and these are relatively straightforward to learn and apply, the real difficulty lies in the human side of the equation. Getting everyone to buy into a common vision, agree on how to collaborate, stay motivated, and function as a cohesive team requires soft skills that are much harder to teach. It demands empathy, judgement, adaptability, and trust. These qualities develop through experience. And even with an experienced manager, the trust needed to bind a team together can take time to build and can be easily lost. This is what makes project management in software development not just a logistical challenge, but a deeply human one.

5.3 Leadership

5.3.1 What is leadership?

FAQ: How do I get people in my team to do what they should be doing?

Leadership is central to any successful software project, yet it is often misunderstood. In this handbook, we use the term to mean the ability to guide a group toward positive outcomes through influence rather than coercion. Leadership is therefore rooted in informal authority: the respect and trust your colleagues freely grant you. Leadership is not a power (e.g. as bestowed through a job title) to coerce people into doing what you want them to do.

Crucially, leadership should never be seen as the exclusive domain of one designated "manager". Every team member who has insight, expertise, or a fresh perspective must be ready to step forward and influence the group when the moment calls for it.

Disagreements are inevitable in creative work: teammates come with different objectives, values, priorities, working styles, and talents. Without the safety net of formal authority, these differences can stall progress or, if handled well, spark better decisions. Navigating them takes judgement: the wisdom to know when to compromise, when to hold your ground, and how to prevent a clash of ideas from sliding into interpersonal conflict. In other words, leadership is the social lubricant that keeps collaboration moving when the project grows messy and uncertain.

5.3.2 *Building informal authority through four foundational behaviours*

FAQ: How do I develop leadership or informal authority?

Influence rarely flows from raw charisma alone. Most people must consciously cultivate behaviours that earn trust and inspire commitment. Research on high-performing teams highlights four foundational habits that strengthen informal authority and make conflict easier to resolve (adapted from [Kogon and Blakemore, 2024]) . Figure 5.1 illustrates the cycle.

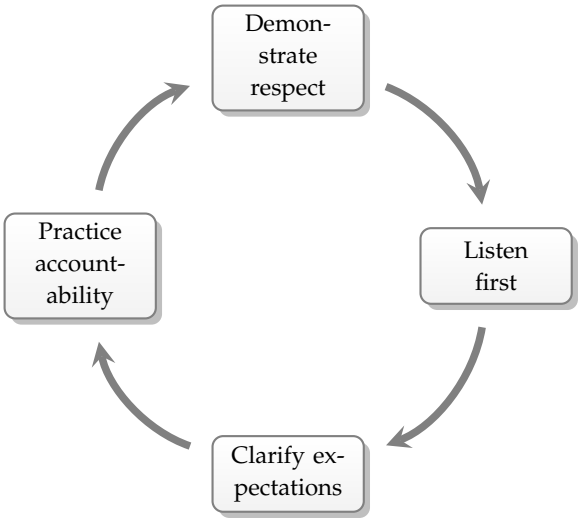

Figure 5.1: Four foundational behaviours to build informal authority

1. *Demonstrate respect*. If people feel you respect them, they are more likely to engage in conversations with (even the more difficult ones). Treat teammates as capable professionals whose time and opinions matter. Respect is conveyed through courtesy, genuine curiosity about their viewpoints, and consideration for their constraints. Respect does not require you to agree with everything people are telling you. It also means confronting reality: raising difficult issues promptly and candidly. Ignoring emerging problems, or saving criticism for anonymous peer reviews, is itself disrespectful.

2. *Listen first*. Take the time to hear out people before commenting, drawing conclusion, or preempting what they will say. This is especially important when having challenging conversation. Unless you take the to listen to others before you share your opinion, you cannot really have considered their views or any important information they might have to share. Failing to listen can cause tension and strain your relationship with your team mates.

3. *Clarify expectations*. Misunderstandings multiply as a project evolves. Re-establish shared goals, priorities, roles, and next steps whenever you sense drift. Anchor the discussion in the team's agreed objectives, documented plans, and the project-management practices covered in lecture. Clear expectations shrink the arena for conflict. When clarifying expectations, draw

on the objectives and plans you agreed as a team, the expectations set out in this document and the group project handbooks, and the project management practices and strategies we discuss in the lectures.

4. *Practice accountability*. Practicing accountability involves recognising concerns certain individuals are responsible for, identifying the nature of the issue, and pursuing remedial action until the concern is removed. Without accountability, problems tend to fester until the team eventually lose control over these problems. Practicing accountability requires three things. (i) Be *transparent*: when you see a problem, speak up. (ii) Attain *buy-in* from your team mates to address the problem. Demonstrating respect, listening first, and clarifying expectations all help you obtain buy-in. (iii) *Follow through* (persistently) until the issue is resolved.

In our discussions of project management, and strategies we will revisit these four foundational behaviours from time to time. They will form the basis of strategies to address inter-personal challenges in project management.

5.3.3 The role of the project manager

FAQ: What is the role of a project manager?

While leadership skills are important to everyone, they are especially important for project managers. the project manager bears special responsibility for orchestrating the whole effort. Their mission is to steer the team toward the project objectives while honouring constraints of scope, schedule, budget, and quality. To do so they must:

- *Maintain a wide-angle view.* Individual specialists focus deeply on their own tasks; the project manager watches the big picture, anticipates cross-stream dependencies, and highlights looming risks before they bite.

- *Clear obstacles.* Rather than dictating technical solutions, the manager removes blockers, securing resources, negotiating with external stakeholders, or adjusting priorities, so experts can do their best work.

- *Cultivate trust in both directions.* The team must trust the manager's guidance, and the manager must trust the team's expertise. That trust is earned through the same four behaviours outlined above, amplified by transparency in decision-making and a willingness to admit mistakes.

- *Make success invisible.* When project management is effective, progress feels smooth and crises are rare, so the contribution can go unnoticed. Poor management, by contrast, is painfully obvious. Accepting that asymmetry is part of the job.

Good software development project managers, therefore, rely heavily on people skills, blended with broad technical systems thinking.

5.4 *Project lifecycle*

Every completed project progresses through a lifecycle: from initial conceptualisation to eventual closure. At each stage of this lifecycle, teams face different concerns and decisions. Understanding these stages helps you focus on the right issues at the right time and anticipate future challenges before they become problematic.

FAQ: What is the overall structure of a typical project?

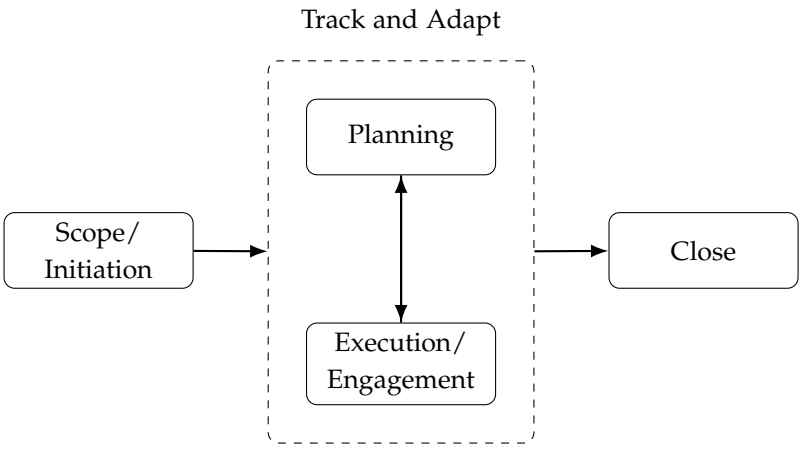

Figure 5.2: Project lifecycle

Kogon et al. describe the project lifecycle as a five-stage process [Kogon and Blakemore, 2024], as shown in Figure 5.2. In what follows, we summarise each stage and highlight the key decisions, pitfalls, and practices that student teams should consider.

5.4.1 *Project initiation/scope*

At the outset, the project must be clearly defined. This includes answering a number of foundational questions:

FAQ: What are the main decisions to be made at the start of a project?

- What are the project objectives?
- Who are the key stakeholders in this project and how will the project outcome affect them?
- What are the priorities if trade-offs are required?
- Who is part of the project team?
- What constraints (e.g., time, resources, budget, quality) must the team operate under?
- What is out of scope for this project?
- What does success look like? How will it be measured?

Projects often fail because these questions are inadequately addressed. Initial project proposals are frequently overambitious relative to available resources, and failing to identify key stakeholders early can lead to wasted effort and misaligned deliverables. Misunderstanding how success will be evaluated, especially by stakeholders with decision-making power, can lead to disappointment at project completion.

In your group projects, these concerns are just as relevant. You may receive briefs that are intentionally broad or ambitious, and it is your team's responsibility to refine them into realistic, achievable objectives. Functionality will be assessed not only on what your software does, but also on the value it provides to its intended users. If your project involves a real client, these users are real people. Moreover, your assessment will follow detailed marking criteria. Understanding how your project will be graded is essential from day one.

5.4.2 Project planning

FAQ: What does project planning involve?

Planning involves identifying the activities required to achieve the project's objectives, and organising people and resources to carry them out. This means creating a project schedule and determining who does what, when, and how. Planning is difficult because it requires you to predict what is achievable given the available time, effort, and information. However, skipping planning leads to inefficiencies. Teams risk duplicating effort, missing critical tasks, or working on features that ultimately are not needed.

One critical part of planning is risk management. Projects rarely go exactly as planned, and anticipating what might go wrong helps teams avoid or minimise negative outcomes. Risk management involves:

1. *Identifying risks*: Reflect carefully on what could go wrong in the project? If your team fails to consider a potential risk, you cannot manage it.
2. *Analysing risks*: For each identified risk, consider its probability and its potential impact.
3. *Risk planning*: High-probability or high-impact risks should be addressed. Finally, risks with high probability or high impact need to be managed. For example, teams could take actions to avoid certain risks (to reduce the probability), or prepare a contingency plan (to reduce the impact).

If a project contains high-probability, high-impact risks that cannot be managed, it may not be worth pursuing in its current form.

Beyond technical tasks and timelines, you are working with people: people who have their own goals, pressures, and commitments outside your project. A good project plan accounts for these human factors. This includes:

- *A communication plan*: Regular meetings (e.g., weekly at the same time and place), communication tools (e.g., Slack, WhatsApp), and agreements about availability and off-hours.
- *A task allocation and review process*: The team needs a clear understanding of how tasks are assigned, completed, and reviewed. Documentation, expectations for code quality (e.g., testing, refactoring), and demo/reporting practices should be agreed upon early.
- *Rules for version control*: Consistent use of Git (or equivalent) avoids confusion, merge conflicts, and overwritten work

As the project progresses, your plan will need to evolve. Unanticipated issues will arise, and you will need to adapt your schedule, risk strategy, and working agreements accordingly.

5.4.3 Project execution/engagement

FAQ: We have a plan. What are the challenges in getting the plan executed?

This phase is about doing the work. The plan is put in to practice, tasks are completed, features are built, and deliverables are produced. However, the execution phase often brings a new challenge: maintaining team engagement. Initial excitement tends to fade. Team members get busy with other modules or responsibilities. Sometimes even clients become less responsive. In group projects, this disengagement often takes teams by surprise, and it can stall progress.

The most effective way to sustain engagement is to develop a practice of accountability, built around predictable routines and shared expectations. For example:

- Hold *regular, time-boxed team meetings* (e.g., weekly stand-ups of 30–40 minutes).
- Begin your meetings on time, stay on task, and keep it short.
- Use a shared project board (e.g., Kanban) to track work and guide discussion.
- As part of your meeting, have each team member reports on what they committed to, what they have completed, and any obstacles they face.
- Task assignments should be made collaboratively, allowing people to choose work that aligns with their interests and strengths.
- When problems arise, assign someone (typically the project manager) to clear the path forward.

Avoid behaviours that disrupt accountability:

- Irregular or ad hoc meetings make it hard to coordinate and deliver. Attendance may be poor. Deadlines remain unclear.

- Waiting around for team members to arrive, perhaps engaging in text conversations with them complaining about London public transport, disrupts the focus of the meeting. Team members should make every effort to arrive on campus well before the meeting so that the meeting can start on time in spite of minor disruptions.
- Off-topic chatter lengthens meetings and undermines focus.
- Excuses or blame distract from the immediate goal: making decisions and moving forward. The team needs to focus current progress and decisions about next steps. Accusatory conversations detract from that.
- Top-down task assignment (especially from a self-appointed "leader") can breed resentment.
- Ignoring obstacles you know to block a recently assigned task's progress sets up the person responsible for failure. Every team member, and especially a project manager, should aim to anticipate problems and speak up when they can foresee them.

Despite best intentions, teams often struggle to practice accountability because team members take too long to voice their concerns. Sometimes this delay is caused by a misguided attempt to avoid conflict by covering up the truth. However, in many cases, the team is simply slow in recognising that a significant problem exists. It pays for group project teams to adopt certain agile development practices that promote speedy recognition of delays and other concerns:

- *Assign small tasks*. A task or a bundle of tasks is only "small enough" if the person assigned to it can complete the work before the next week. This rule of thumb supports weekly tracking: a task should be completed or reported as incomplete in the next accountability meeting. If it is not done, it suggests that either the task was too large or the team member failed to deliver. Irrespective of the underlying reason, both situations require discussion. Assuming a weekly 15-hour workload allocation for this module, tasks should generally take no more than 10 hours, with a deadline of 7 days or less.
- *Assign vertical tasks*. A vertical task includes all the layers needed to deliver a small, functional improvement, including UI, control logic, helper functions, and data storage. Vertical tasks immediately add value to the software and are usually more independent of one another, reducing the likelihood of delays cascading across the team. In contrast, horizontal tasks, those focused solely on one aspect of the software (e.g., all UI screens), often depend on other layers being in place first. Horizontal task specification introduces additional risks that delays have knock on effects.

- *Discuss and agree what it means for a task to be complete*. In teams where the members do not have a shared set of expectations of what constitutes a completed task, delivery of work can lead to disappointment. As a general rule, you should:
  - Require each developer to write their own automated tests immediately. In other words, each team member should write the test code for their own source code. They must do this before, or immediately after writing the source code. Tests protect source code against the introduction of bugs: if someone introduces to source code breaking changes, complete automated tests will pick these up forcing the author of problematic code to correct their work. Without tests, source code breaking changes can be introduced unchallenged. Procrastination and delegation of testing are both recipes for poor or late testing.
  - While attaining high code cleanliness and design standards require substantial code inspections and review, it is sensible to take steps to ensure some code cleaning occurs before code is shared. Teams that do not agree common standards may end up with significant differences in the quality of work of different team members, which can breed resentment. To avoid this, agree a standard of code cleanliness that each team member can and should achieve in their own work.

Adopting these practices early supports regular progress, early identification of issues, and a healthier team dynamic based on shared expectations and visible contributions.

5.4.4 Track and adapt

FAQ: Our plan is not working. What do we do?

Ignoring known blockers during planning or handover leads to predictable failures.

- *Scope creep*: New feature ideas sneak in, expanding the project beyond what is feasible. This often comes at the expense of code quality or testing.
- *Reduced team capacity*: If some members disengage or do not engage at all, the remaining workload may become unsustainable. The scope needs to be reduced.
- *Technical debt*: Features are delivered with incomplete testing or inspection, leaving a backlog of work that must be addressed.
- *Disruptive behaviours*: A team member might disregard task boundaries, overwrite others' work, or over-communicate. This may indicate that your working arrangements need to be revised or reinforced.

These issues typically appear slowly, making them easy to ignore, until they become urgent. That is why tracking progress and

reflecting on performance regularly is crucial. Build habits that help your team detect and respond to early warning signs.

5.4.5 *Project close*

FAQ: What are the main decisions to be made at the end of a project?

In the context of the student group project, the closing phase culminates with submission. However, this stage involves more than simply uploading a file to KEATS: it is a critical part of the project lifecycle that requires careful coordination, attention to detail, and reflection.

Before submitting your work, the team must perform thorough quality control:

- *Review your submission*: Have you included everything required for the examiners to assess the work? Conversely, have you excluded any unnecessary items (e.g., the full Git repository)?
- *Verify instructions*: Do the installation and setup instructions work on a fresh machine that has not been configured for development?
- *Confirm deployment*: Is the software deployed, seeded with relevant data, and accessible using the required user credentials?

Preparing and completing the submission is a collective team responsibility. Therefore, it requires a coordinated team effort: one that requires adequate time to complete. Beware that ensuring everything works as intended at the point of submission can be surprisingly complex.

Start submitting early! Do not leave submission until the final hours before the deadline. Servers are often under heavy load near submission deadlines and may respond slowly, or fail altogether. Starting early gives your team time to respond to unexpected problems without unnecessary stress.

Once submission is complete, take time to reflect. Project experience help you develop critical professional skills, including effective communication, collaboration with others, planning, risk management, and decision-making. Your learning can be maximised via critical self reflection. Ask yourself:

- What went well in this project?
- What challenges arose, and how did you respond?
- What would you do differently next time?

Avoid falling into the trap of assigning blame. While it is natural to feel frustrated if things did not go perfectly, more valuable insights often come from reflecting on your own actions: what you contributed, how you communicated, and how you adapted.

The group projects provide multiple opportunities to learn from your experience. You will write peer assessments for your team mates. You will receive feedback from your peers in the form of the

peer assessments they wrote. You will receive detailed feedback from markers once assessment is complete. You may also chose to have a constructive debrief with your teammates. Engage fully with these opportunities. They are designed not just to assess your performance but also to help you learn.

Last but not least, feel free to celebrate the end of your project and your successes. Recognising what you have achieved together is not only satisfying. It is a great way to build friendships and connections with fellow students who may be your future colleagues or collaborators.

5.5 Communication

FAQ: What is the role of communication in project management?

The core activity of project management is *communication*. All the decisions that must be made throughout a project's lifecycle rely first and foremost on good communication. Take scheduling project tasks, for example. This may seem like an optimisation problem. However, finding a good solution to the problem is rarely difficult unless the project is particularly large or severely constrained. Scheduling becomes difficult when different people have a different understanding of a task's expectations, when the people doing the actual work realise the task is much harder than the rest of the team appreciate, or when something goes wrong in the completion of a task but not everyone is aware of the problem or appreciates what it is. Resolving these issues requires good communication.

This section aims to identify the diverse means of communication you use in the group projects, what you need to communicate via each means, and why it is important.

5.5.1 Informal communication

FAQ: Do team members need to talk to each others?

Any project in a small team involves a considerable amount of informal communication. Whether it is to clarify a detail of a task, to ask or receive some help with a problem, or simply to vent one's frustrations or encourage a team mate, project teams benefit from open communication channels that can be used without too much formality.

To promote informal communication, you can use a number of approaches and tools. One approach is to work together in the same room (you can use the labs, but take care not to disturb others using the same space for work). This allows for the most effortless informal communication, but it does involve some organisation and travel. Another is to use a messaging and/or video conference service. There are messaging services that are designed for professionals, or for software engineers in particular, but any tool everyone in the team feels comfortable with is fine. When relying on messaging, beware that your working hours may not be the working hours of your team mates, and everyone is entitled to their personal time and space. When using a messaging service, teams should have a

conversation about boundaries, such as out-of-hours messaging.

Informal communication *complements* a range of more formal forms of communication. It cannot be a substitute for the latter. For example, you should not use your messaging tool or informal chats to make important decisions, or allocate work to someone who does not attend meetings.

5.5.2 *Team meetings*

FAQ: How do we organise team meetings?

Important discussions and decisions about the direction of the project, including task allocation and holding the team accountable, should be made at a team meeting. This ensures that everyone in the team has a chance to consider and comment on the decision, there is a clear decision point (rather than an endless discussion in a messaging board), and everyone is informed of the decision.

A team meeting is a particular type of gathering where the team informs itself about the state of the project and makes key decisions about its future direction. Other occasions where the team gets together, such as a brainstorming session, a collaborative coding session, or a presentation, will not be referred to as meetings here. To be effective, a team meeting needs to be *purposeful*. It requires everyone's full *participation* and *concentration*. To achieve this, you will keep your meetings *short* and *on point*. Key aspects of the process and outcomes need to be *documented in writing*, so that there is a record of the team's decisions. To achieve this, teams should follow this protocol:

- *Scheduling*: To schedule meetings efficiently, schedule all meetings once at the start of the project by booking a regular time and place (i.e. same time and place each week) at the start of the project. Ad hoc scheduling of individual meetings makes organising meetings time consuming, and this gets only harder as a term progresses. Teams that do not schedule their meetings at the outset tend to have a hard time keeping the team engaged, and sometimes fail to organise any meetings. Your meetings should have a clear start *and end* time. People can only focus for a limited period of time, and they also need to be able to meet other commitments. As a general rule, it should be possible to complete a team meeting in 30 minutes, but I recommend scheduling 1-hour slot per meeting to allow for inefficiencies.
- *Roles*: Prior to the meeting, assign one team member to chair the meeting and a second team member to take minutes. The chairperson will be responsible for guiding the conversation, ensuring all attendees have a equitable chance to participate. The minute taker is responsible for preparing and distributing the meeting agenda, taking the meeting minutes, and making corrections as necessary.
- *Agenda*: Every meeting should have a clear agenda, written up and shared with all members of the team well in advance of the

meeting itself. This ensures that everyone knows what to expect at the meeting, and can prepare accordingly. For each agenda topic, consider carefully what outcome you are looking for (e.g. agreement, information, decisions on actions, etc.). Avoid open-ended discussion topics: these are best handled through preparation outside meetings (see below). Assign each item to a person or persons responsible for presenting them. Sometimes, this can be a single person presenting their findings of an information gathering exercise. Sometimes, every member of the team will be required to report something. To keep the meeting concise, it is advisable to add timings to each agenda topic (and stick to the timings). Standing items on the agenda will normally include: approval of the minutes of the previous meeting, a review of the action log (see below), and any other business not already in the agenda (if time permits).

- *Preparation*: The key to an efficient meeting is good groundwork before the meeting. Consider, for example, deciding what technology stack to use to develop your software. That decision should consider a range of factors, including the team's training requirements, available tools (e.g. for automated testing, test evaluation, and other build automation tasks), and ease of deployment. Before considering this decision at a meeting, the team should have collected all necessary information. Preparation could include all team members trying out the technology stack, and trialling build automation tools and deployment. Preparation tasks are typically assigned through an action log (see below).

- *Attendance*: Attendance at team meetings is mandatory. Regular absences preclude a team member from engaging adequately with the team. As an incentive to attend, get into the habit of recording attendance accurately from the outset.

- *Meeting*: During the meeting, the team should have a focussed discussion of all agenda items. It is important to stay on-topic: leave out small talk or anything unrelated to the agenda. While these other types of conversations are important, they prolong and detract from the meeting. The chairperson's role is keep the meeting on track. Often, meetings do not go entire to plan, especially when an agenda item does not resolve itself through conversation. In those situations, the discussion should focus on identifying what actions will help resolve the matter in the future, perhaps by the next meeting. The chairperson is typically also responsible for holding participants accountable for the actions they are due to deliver on. The chairperson must also ensure everyone has a fair chance to participate and this may involve calling on dominant voices to listen to others and on quiet people for their views and concerns.

- *Minutes*: The meeting minutes are the definite record of the out-

come of the meeting. It typically includes all decisions that were made during meeting, ideally accompanied by the underlying rationale. Where a decision involves significant risk or controversy, alternative options and their rationale should be documented too. The minute taker is responsible for writing the minutes in a timely manner, ideally during and immediately after the meeting, sharing them with the rest of the team (you will be using Team Feedback for this), and acting on requests for corrections. The team members are responsible for reading the minutes and pointing out any errors.

- *Action log*: Alongside the minutes, the minute taker should also maintain an action log of tasks arising from your meetings. Whenever an agenda item is not resolved into a decision or agreement, more preparation work will normally be required. These should be compiled into a "To Do" list for the team. In essence, an action log is just a list of names/descriptions of actions, the person or people responsible for completion, the action's deadlines, its status, and possibly some notes (containing information such as reference to the minutes of the meeting in which the action was created). It is advisable to use a single action list for the entire project, so that you can keep track of all outstanding actions in a single list.

5.5.3 *Project scheduling and monitoring*

FAQ: How do you communicate project plans?

The bulk the work associated with your software engineering group project consists of software development/engineering tasks. These tasks are distinct from the actions arising from meetings: they are substantial pieces of work that contribute directly to the project objective and may require careful scheduling and monitoring. A typical software development task involves source code development, test code development, and quality control. Delays in one task can impact the work of others. In the group projects you will be doing, a simple action log is a somewhat overly simplistic way of managing such tasks.

One common approach is to use a Kanban board. In a Kanban board, tasks are written on cards placed on a board with several labelled columns. In agile software development projects, the cards typically contain user stories describing a task. The column a card is in represents the status of the task. Cards move from left to right. Thus, the column sequence used in the diagram represents the team's process. Figure 5.3 illustrates this idea with an example Kanban board for a simple library application. Here, the software development process maintains a backlog of considered tasks. Allocated tasks have two active stages – "In Progress" and "Under Review" to complete – before they are considered done.

When using such tools, take care to communicate its contents clearly. When using a Kanban board, you need to have a clear understanding of what is required for a card to move from one

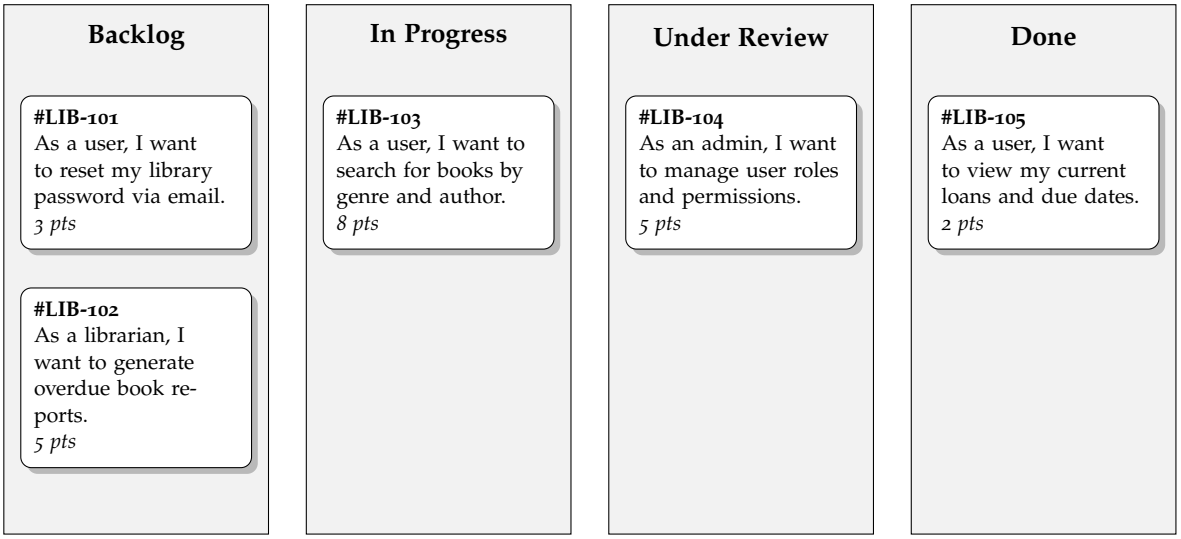

Figure 5.3: A sample Kanban board

column to the next. Adding content into these tools is only half the work (and the least useful part). Make sure that the team regularly review the information contained in them to guide future planning, and to adjust your project's course as necessary.

5.5.4 *Version control*

FAQ: In what ways are version control tools communication tools?

In team based software development, version control tools provide the means to share code and information about that code. In this module, we will be using git and GitHub for version control. Version control tools provide an important means of communication.

As you develop code, it is important that you share your work with the rest of team immediately by making commits and pushing them to the remote repository. This signals to team mates that you are working on a task. On the days that you work on the project's code base, you should be making and pushing at least one commit per day (ideally more if you are doing anything substantial).

- *Commit messages*: A commit message in a Git repository should clearly and concisely communicate the purpose and scope of the changes introduced in that commit. It serves as a historical record for other developers (and your future self), helping to explain what was changed and why. A well-written commit message typically begins with a short, imperative-style summary (e.g., "Fix broken image rendering on mobile") no longer than 50 characters, followed optionally by a more detailed explanation in the body if needed. The message should focus on the intent of the change rather than the technical details of how it was implemented, as the code itself shows the "how". Good commit messages improve collaboration, make debugging and reviewing easier, and enable meaningful version history and change logs.
- *Issues*: If you are using them, a Git issue should clearly describe a problem, feature request, or task in a way that is understand-

able to others who may work on it or review it later. It should include a concise and descriptive title that summarises the issue, followed by a detailed explanation that provides necessary context. For bugs, this typically includes steps to reproduce the problem, expected vs. actual behaviour, relevant error messages or logs, and environment details. For feature requests or tasks, the issue should describe the motivation, desired outcome, and any constraints or dependencies. A well-written issue helps prioritise work, facilitates effective collaboration, and ensures that team members have a shared understanding of the problem or goal being addressed.

- *Pull requests*: A Git pull request is a request to merge a development branch into the main branch. The request should clearly communicate the purpose, scope, and context of the proposed changes to facilitate efficient review and collaboration. The title should be concise but descriptive, summarising what the pull request does (e.g., "Add pagination to search results"). The request may reference the task or user story it implements. The description should explain why the changes are being made, referencing related issues or tickets when applicable, and outline what has been changed at a high level. If relevant, it should include changes that could be deemed out of scope of the original task, known limitations, or any points requiring special attention during review. A good pull request helps reviewers understand the intent behind the changes, reduces back-and-forth clarification, and ensures the code can be confidently merged and maintained.

In the group projects, we will be using the commit history to track coding activity and contributions. Therefore, *you must preserve the commit history of your repository at all times*. Teams must not perform hard resets, rebase, reflog + reset, branch deletion, or similar operations on their repository. You can revert a commit as that operation preserves the commit history.

The most critical operation in a shared repository is that of merging branches. When merging two branches in a Git repository, conflicts can occur if the same lines in a file were changed differently on both branches, requiring manual resolution. Additionally, unintended changes may be introduced if one branch is outdated or not properly tested, leading to broken functionality or regressions. If the merge is forced or not reviewed carefully, it may also overwrite or discard important work from one of the branches. Therefore, every team needs to agree on a carefully designed process – one that includes appropriate checks and balances – to merge development work from a branch with the rest of the code.

5.5.5 Code

FAQ: In what ways is code a communication tool?

The ratio of the amount of time software engineers spend reading code vs. writing code is said to exceed 10:1 [Martin, 2009]. When

working in a team, much of that code will be written by others. Thus, code is a form of communication in its own right. In a larger project, it is important to invest some of your time to write clean code that you and your teammates are able to make sense of throughout the development period.

The importance of clean code extends to automated test suites. As these test suites prescribe how the code is expected to behave, developers can use these to understand the software's specifications.

Writing clean code in a group project requires a concerted effort from the whole team. It takes somewhat more effort to identify the cleanliness of an individual developer's code compared to the functionality that that developer contributed. When time is limited, individual developers may be tempted to cut corner in an effort to meet deadlines and report good progress at meetings. To ensure high code cleanliness standards are maintained throughout the code, the team's project management should include the following:

- At the outset of the project, discuss and agree the *code cleanliness standards* expects to maintain throughout the development period. These standards should be specified in writing. The team must also discuss and agree *a process* to enforce the standards agreed by the team. Normally, such a process will include code reviews/inspections at specific times. A good time to perform a code review is when code is ready to be merged (e.g. when a pull request is raised).

- During the project, enforce the standards and the process. During this stage, you must practice *accountability*. The team will not be able to attain the expected standards if substandard code is allowed to pass inspections, or if team members are allowed to deviate from the process. If you have concerns, speak up and use the four foundational behaviours as a guide.

5.6 Waste

FAQ: How do we avoid wasting time and resources in our project?

Even when everyone in your team seems suitably engaged and active, many team members find themselves disappointed with the results their team is delivering. It can be temptingly convenient to blame this on your team mates. Once you have come to that conclusion, a typical response is either to push them to do more work, or resent them for not being good enough. A common cause of poor productivity, however, is *waste*. Therefore, productivity problems are best addressed by seeking and removing causes of waste.

Typical causes of waste include [Hooker and Moir, 2022]:

- *Software defects*, such as bugs or source code that does not behave as required. Undetected software defects become increasingly intractable as a code base grows. This is especially problematic

when software defects accidentally are introduced into a module as a developer is working on a different module or refactoring old code. Writing automated tests as soon as possible can help avoid many defects.

- *Relearning* something you had already learned previously is duplication of work, and therefore wasteful. You may need to relearn something if you stop a task, and pick it up again at a later date. If you delay testing, you may need to relearn the source code that requires testing. If you delay refactoring, you may need to relearn the source code to be refactored.
- *Task switching* is necessary when a developer is working on multiple tasks simultaneously. Some teams or individuals are tempted to take on more tasks, perhaps because they feel they are not producing enough work or because they struggle to finish work. However, adding more tasks to your "To Do" is not going to help you complete tasks any faster. On the contrary, every time you switch tasks, you need to switch contexts and refocus. Some relearning may be needed. Managing your workload and priorities also becomes harder, and this can cause stress. Ultimately, you will become less productive. Avoid this by only taking on a new task after finishing another.
- *Incomplete or partially done work* provides no value to end users. It is not generally awarded marks in the marking scheme. Effort expended on work that is not completed is wasted, and might have been put to better use on tasks that do add value. Avoid incomplete code by assigning small, vertical tasks.
- *Delays* occur when a task needs to be put on hold. A common reason for a task is be blocked is late delivery of a dependency. Delays create waste due to relearning and task switching. They also increase the risk that the task will not be completed. Avoid delays by limiting dependencies between
- *Handoffs* of tasks – i.e. reallocating work to others in the team – creates extra work. The extra work includes handoff communication, relearning, and task switching. It is best avoided by keeping tasks as independent of one another, and by keeping them small in scope.
- *Excessive features* are software features that are not required, or beyond the capacity of the team to deliver. Extra features increase the amount and complexity of source code that needs to be built, maintained, refactored, and tested. But the effort needed to produce the source is not readily available to the team, so something (e.g. code quality or robustness of features) needs to be sacrificed. This tends to be wasteful because the value gained is generally lower than that which has been lost. Avoid excessive features by developing the software in small increments, each one adding something of value.

If you have concerns your team productivity, reflect on your ways of working. Try to identify wasteful practices, so that you can eliminate them.

6
Software quality assurance

Software quality refers to a wide and diverse range of desirable characteristics of software. The relative importance of different characteristics varies from project to project. Nevertheless, there are certain characteristics, such as functional suitability, reliability, and maintainability, are important in all projects. Functional suitability is the degree to which a software system meets the stated or implied requirements. In other words, functional suitability refers to the extent to which a system meets the objectives it was built for. Reliability is the ability of a software system to perform its required functions consistently and without failure. Reliability includes, for example, a system's ability to fail gracefully under fault conditions (e.g. not losing data if an operation is interrupted). The ease with which a system and its component can be modified, corrected, or enhanced to improve performance or adapt to changing requirements. Functional suitability, reliability, and maintainability are ensured by writing clean code, employing sound design, and building comprehensive automated test suites. This chapter focusses on these means to attain software quality.

FAQ: Does software quality matter?

6.1 Clean code

Clean code is not just about writing code that works. It is about writing code that can be easily understood, maintained, and extended by your team and future developers.

FAQ: Does clean code matter?

The following checklist provides a set of practical guidelines for writing clean, maintainable code in your software engineering group project. These principles are summarised from Robert C. Martin's *Clean Code* textbook [Martin, 2009]. You can use this checklist in your team's code inspections. Alternatively, select a subset of these principles for code inspections and leave a broader range for code reviews.

6.1.1 Naming

Use meaningful and descriptive names for variables, functions, classes, and files. Names must make meaningful distinctions. Clear names reduce the need for comments and make your code more self-

FAQ: What is a "good" name?

documenting. It is much more important to make code easy to read than making it easy to write.

```
# Good
total_price = calculate_total(cart_items)

# Bad
tp = ct(c)
```

FAQ: Can we create our own abbreviations, so that our variable names are shorter?

Avoid abbreviations unless they are widely understood (e.g., `id`, `url`). Abbreviations often save only a few keystrokes but greatly reduce clarity. You should not be introducing new abbreviations specific to your project, if they are not already used in day-to-day parlance. Doing so increases the amount of knowledge developers require to engage with the code.

```
# Good
user_id = 123

# Bad
uid = 123
```

FAQ: Do we need to coordinate the way we name things?

Use consistent naming conventions. For example, you may use `snake_case` for variables and functions and `PascalCase` for classes. Consistency helps readers form expectations and improves readability. If you are inconsistent with the naming conventions, such as use `camelCase` for some of your variables, then developers will need to remember whether each variable uses `snake_case` or `camelCase`. As part of your conventions, use the same word to refer to a particular operation or thing.

```
# Bad: inconsistent naming of operation to read/load a file
def read_users(filepath):
    ...
def load_users(filepath):
    ...
```

FAQ: Do I need to be able to say a name out loud?

Choose pronounceable names to ease verbal communication within the team. If you cannot say the name out loud, it is harder to discuss code in meetings and code reviews.

```
# Good
database_connection

# Bad
dbcnx
```

FAQ: Can variable names be misleading?

Avoid misleading names that suggest incorrect behaviour or data type. A name causes the reader make assumptions about the data a variable contains or a function returns. When you write the code, that may

not be a concern because you recall what the variable contains. However, clean code is written with its future readers in mind, who will not have access to the writers initial recall. A misleading name can cause incorrect assumptions, leading to bugs in the code (or at least more challenging debugging efforts).

```
# Bad
def is_valid():
    return None # Misleading; suggests a boolean return

# Better
def get_validation_errors():
    return ["Missing␣field:␣name"]
```

6.1.2 Functions

FAQ: How long can a function be?

Functions should be small Smaller functions are easier to understand, test, and reuse. As a rule of thumb, aim to limit functions to 5 – 15 lines of code. You can achieve this by using additional functions (and classes/methods) to break down the body of a function, ideally in such a way that each function operates at one level of abstraction.

FAQ: How much work can a function do?

Functions should do one thing, and do it well. A function that tries to do too much is hard to understand and maintain.

```
# Good
def save_user():
    ...
def send_welcome_email():
    ...

# Bad
def save_user_and_send_email():
    ...
```

FAQ: How should functions be named?

Function names should clearly state their purpose and side effects. Normally, that means a function will named by a verb or a verb phrase. Avoid generic names like ‘doStuff‘ or ‘handle‘.

```
# Good
def delete_temp_files():
    ...

# Bad
def process_files():
    ...
```

Instead, be specific if you have to be. If you need a rather long verb phrase because your function does different things or has side effects, then the issue is that your function is doing too much.

Instead, be specific if you have to be. If you need a rather long verb phrase because your function does different things or has

FAQ: Can functions have side effects?

Avoid side effects unless they are intentional and clearly documented. A function has a side effect if it changes the state of your system even though that is not appear to be the purpose of your function. Side effects make the behaviour of your system unpredictable. They can introduce bugs, especially if they are not obvious to the caller.

```
# Bad
def calculate_tax(user):
    user.tax_due = 100 # Unexpected side effect
    return 100

 # Not as a bad
 def calculate_and_save_tax(user):
    # Updates tax field of user record in database
    user.tax_due = 100
    return 100

# Good
def calculate_tax(user):
    return 100
```

FAQ: How much nesting in a function is ok?

Avoid deeply nested functions. Deep nesting makes functions hard to read as the reader needs to keep track of the control logic at multiple levels. Usually, a function with deep nesting also contains source code at different levels of abstraction. Using sub-functions to separate the different levels of abstraction reduces the number of levels of nesting in a single function body and makes the code more readable. Ideally, your function has just one level of nesting. But that takes considerable discipline and extensive clean up. If you cannot achieve that, aim for two levels of nesting at most.

FAQ: How many parameters can a function have?

Functions should have as few parameters as possible. Too many parameters make functions harded to understand and call. The ideal function has not parameters at all, but that removes what makes a function a function. One or two parameters is fine, but group parameters where possible. Avoid more than three or more parameters whenever that is possible.

```
# Acceptable
def create_invoice(customer, items, discount):
    ...

# Better (grouped)
def create_invoice(invoice_details):
    ...
```

tion to have

Use default arguments or object parameters when appropriate. Default arguments simplify function calls while preserving flexibility. Specifically, the code that calls the function tends to look simpler because some parameters do not need to be specified, making it easier to read.

```
def connect_to_server(host, port=22):
    ...
```

6.1.3 Code structure

Organise code into logical, cohesive modules. Group related functions, classes, and data into the same module or package. This improves code discoverability and separation of concerns.

FAQ: Does it matter what file/package you put code in?

```
# Good: user−related code grouped in one module
# user.py
class User:
    ...
def create_user():
    ...
def get_user_by_id(user_id):
    ...
```

Do not repeat yourself. Write DRY code. Repetitive code, also known as WET ("We Enjoy Typing"/"Waste Everyone's Time") code, is significantly harder to maintain than DRY ("Don't Repeat Yourself") code. Code duplication lengthens the code base unnecessarily (making it harder to read), multiplies the places bugs can hide, makes updates error-prone (you will forget to change one copy), and slows teammates trying to understand what is really happening. Often, code duplication can be avoided by extracting common logic into a single, well-named function or method. For example, instead of:

FAQ: Is code repetition acceptable?

```
# Bad
x = 5
y = 8

x_squared = x * x
y_squared = y * y
print("x^2=", x_squared)
print("y^2=", y_squared)
```

write:

```
# Better
def squared(n):
    return n * n

for var, name in [(5, "x"), (8, "y")]:
```

```
    print(f"{name}? = ", squared(var))
```

FAQ: Does the order of functions/data matter?

Keep related functions and data close together. Co-locating related code helps maintain mental context and reduces the need for jumping between files.

```
# Bad: scattered definitions
# helpers.py
def log_user_activity(): ...

# user.py
class User: ...

# Good: related logic placed together
# user.py
class User:
    ...
def log_user_activity(user): ...
```

FAQ: Does formatting matter, assuming the code compiles correctly?

Use consistent indentation and formatting throughout the codebase. Inconsistent formatting is distracting and error-prone. Follow your team's agreed-upon style or adopt a widely used style guide (e.g., PEP 8 for Python).

```
# Good
def get_username(user):
    return user.name

# Bad
def get_username(user):
      return user.name
```

FAQ: How long can a source code file be?

Limit the length of source files. Split large files into smaller ones when needed. Large files become hard to navigate and understand. As files become longer, the reader increasingly needs to scroll through the file to find the code they wish to read or edit. Instead, you should use modular design to separate features logically and keep file sizes small. It is difficult to be prescribe how long a file can or should be. This will vary depending on the nature of the source code of the project. Obviously, one cannot set hard limits.

In the small group project, however, there are specific expectations set out in the marking criteria. The limits are relatively generous. We use specific limit to encourage teams to coordinate their inspection/review processes to ensure that limits are adhered to.

FAQ: How many entries can a directory contain?

Limit the number of files in a directory. Organise a directory into subdirectories when needed. When your code base becomes quite large, limiting the lengths of files will result in a larger number of files. Eventually, directories will become difficult to navigate.

Place higher-level concepts above lower-level details in source files. Define public-facing or summary-level functions and classes at the top of the file. Place helpers or implementation details below to reflect a top-down reading order. This structure limits the amount of scrolling developers need to do to engage with the public-facing functions/classes they need to use.

FAQ: What order should functions, methods, and classes appear in within a source code file?

```
# Good
def run_pipeline():
    data = load_data()
    results = process_data(data)
    save_results(results)

def load_data():
    ...

def process_data(data):
    ...

def save_results(results):
    ...
```

6.1.4 Comments

Document public classes, methods, and functions, ideally in a format suitable for automated documentation generation tools. This helps users and teammates understand how to use your code. In Python, follow the docstring conventions (e.g., PEP 257 or NumPy/Sphinx style).

FAQ: What kinds of comments should our code contain?

```
def calculate_tax(price, rate):
    """
    Calculate the tax for a given price.

    Parameters:
    price (float): The price before tax.
    rate (float): The tax rate (e.g., 0.2 for 20%).

    Returns:
    float: The tax amount.
    """
    return price * rate
```

Write comments only when the code cannot be made self-explanatory. You should use comments sparingly. After all, comments are extra stuff for the reader to read, and we are aiming to write clean code to make the code base easy to read. You should aim to write clear code rather than comment everything. If you can write the code well, with good names and short functions, comments in the bodies

FAQ: Is there such a thing as excessive commenting?

of functions and methods, or for private datatypes are generally unnecessary. If you use comments outside the headline public functions, methods and classes, use them to clarify non-obvious intent, not to restate the code.

```
# Acceptable if the logic is complex
# Using binary search to improve lookup efficiency
def find_item(sorted_list, target):
    ...
```

FAQ: Can I use comments to explain how my code works?

Avoid redundant comments that restate what the code already expresses. Your code should explain how your code works, provided Redundant comments clutter the code and can quickly become outdated.

```
# Bad: redundant
i = 0 # Set i to 0

# Good: self–explanatory code needs no comment
index = 0
```

FAQ: Can I use a comment to explain how a particularly difficult line of works?

Use comments in the body of the code to explain why *something is done, not* what *is done.* You should always avoid comments inside the bodies of functions and methods, or comments to explain certain lines of code. If you find that the code hard to read, try to clean the code rather than add comments. If you do need comments, use them to explain reasoning, trade-offs, or non-obvious decisions. The "what" should be clear from good naming and structure.

```
# Why: API fails silently if sent too many requests at once
time.sleep(0.5) # Throttle to avoid rate limiting
```

FAQ: Can comments become outdated?

Keep comments up to date. Delete outdated or incorrect comments. Comments can become outdated rather easily. When we are editing code, we are working towards changing its behaviour. In the process, we may not spot that a comment is no longer valid. After all, the compiler or parser is not going to complain about an incorrect comment. In general, a wrong comment is worse than an unnecessary: it misleads and wastes time.

```
# Bad: comment does not match the code
# Multiply by 2
value = value * 3
```

A classic example of a comment that tends to become outdated very quickly is a comment that refers to line numbers.

FAQ: Can we use TODO comments or comment out code?

Remove noise, such as "TODO" comments and commented out code. As a temporary measure, while working in a branch, it can be useful to incorporate "To Dos" or comment out code. However, leftover code and placeholders clutter the file and confuse future readers. Before merging your work with the main, these types of comments need to

be removed. Clean up after yourself and use issue trackers or pull requests to keep track of concerns that transcend your task.

```
# Bad: commented–out legacy code
# def old_function():
# pass

# Bad: vague TODO with no owner or deadline
# TODO: fix this later
```

6.1.5 Formatting

Use consistent spacing, indentation, and bracket placement. Formatting rules and conventions vary from programming language to programming language. The key is to be consistent with the languages conventions and, where there is some freedom to choose formatting, you are consistent within your team. Consistent formatting enhances readability and prevents subtle bugs, especially in indentation-sensitive languages like Python. In other words, it should not be possible to see who wrote what code based on the formatting, because everything should look consistent. To achieve this, you will need to discuss formatting with your team mates.

FAQ: What rules should we follow with regards to spacing, indentation, and bracket placement?

```
# Good
def is_even(n):
    if n % 2 == 0:
        return True
    else:
        return False

# Bad: inconsistent indentation and spacing
def is_even(n):
 if n%2==0:
  return True
 else:
     return False
```

Use blank lines to separate logically distinct sections of code. Blank lines provide visual structure, making code easier to scan and comprehend.

FAQ: How should we use blank lines in source code?

```
# Good
def load_config():
    ...

def connect_to_server():
    ...

# Bad: no separation
def load_config():
```

```
    ...
def connect_to_server():
    ...
```

It is important to be consistent in your use of blank lines. Do not use excessive whitespace. Avoid the use of blank lines inside the body of a function. Instead, make your function body smaller so that blank lines are not needed.

FAQ: How should units in a source code file be organised?

Group related code together and separate unrelated code. Keeping related logic close together helps maintain focus and context while reading.

```
# Good
def create_user():
    user = User()
    save_to_db(user)
    send_welcome_email(user)

# Bad: interleaved with unrelated logic
def create_user():
    user = User()

def log_usage():
    ...

def save_to_db(user):
    ...
```

FAQ: Is there a maximum line length

Keep line lengths reasonable. As with other length limits, this is a judgement call. However, lines of code can be too long because long lines are hard to read and may wrap awkwardly on some displays or printouts. Therefore, you need to keep line lengths short. Consider that some developers may be working on the code using a smaller laptop, and that they should be able to view each line of code in full without wrapping. As a guide, you will want to keep line lengths under 100 characters (or less).

6.1.6 Error handling and control flow

FAQ: Which is better: throwing exceptions or returning error codes?

Use exceptions rather than error codes where possible. Exceptions separate normal logic from error-handling logic, making the code cleaner and easier to follow. Therefore, where errors occur, you should be throwing exceptions rather than returning error codes.

```
# Good
def get_user(user_id):
    if user_id not in user_db:
        raise ValueError("User ID not found")
    return user_db[user_id]
```

```
# Bad
def get_user(user_id):
    if user_id not in user_db:
        return -1 # error code
    return user_db[user_id]
```

Avoid deeply nested code by returning early when conditions are not met. Deep nesting makes code harder to read. Error/exception handling is a common reason for nesting code, but even here you should limit the amount of nesting. One approach you can take is returning early: this reduces indentation and improves readability, especially for guard clauses.

FAQ: Does nesting structures that perform error handling (e.g. try-except blocks) count towards nesting limits?

```
# Good: return early
def process_item(item):
    if not item:
        return
    if not item.is_valid():
        return
    item.process()

# Bad: deeply nested
def process_item(item):
    if item:
        if item.is_valid():
            item.process()
```

Handle all expected error conditions gracefully and clearly. If you silence errors, perhaps by catching them in a try-except block in a Python application for example, your code will continue to run even though it does so under abnormal conditions. The cause of the error may lead to unwanted behaviours that become difficult to diagnose. Therefore, silencing errors is a bad idea. Think about what might go wrong and how it should be communicated. Avoid crashing or silent failures.

FAQ: Can we silence errors?

```
# Good
try:
    data = fetch_data()
except NetworkError as e:
    log_error(e)
    show_error_message("Network␣issue.␣Please␣try␣again␣later.")

# Bad: unhandled exception may crash the program
data = fetch_data()
```

6.1.7 Testing Considerations

FAQ: The code I write is really hard to test with automated tests. Is that a problem?

Write code that is easy to test (e.g., avoid global state). Code that relies on global variables or complex side effects is harder to test reliably. Where possible, use pure functions. These are functions that always produce the same output for the same input, without side effects. One technique to achieve this is dependency injection. Here, the resources required by a function – the dependencies – are passed as arguments rather than created within the function.

```
# Harder to test code
import requests

def get_weather():
    response = requests.get("https://api.weather.com/today")
    return response.json()

# Easier to test code
def get_weather(http_client):
    response = http_client.get("https://api.weather.com/today")
    return response.json()
```

The second function takes an HTTP client object as an argument. This can be an object that makes genuine HTTP requests. It can also be a mock client that returns fake (weather) data for testing purposes.

FAQ: How can we make code more testable with automated unit tests?

Design small, independent units that can be tested in isolation. Small functions, methods, and classes with clear responsibilities can be tested with minimal setup and provide clearer test failures. Therefore, designing functions, methods and classes in such a way not only makes the cleaner but easier to test too.

```
# Good: isolated, no external dependencies
def is_valid_email(email):
    return "@" in email and "." in email.split("@")[-1]

# Bad: function depends on external config
def is_valid_email(email):
    return re.match(CONFIG["email_regex"], email)
```

FAQ: Does test code need to be clean?

Keep test code clean and readable, following the same standards as production code. Test code is code that prescribes how your source code needs to behave. It is, therefore, important information that those who read your code may need to consult. Test code should be as readable and maintainable as the code it validates. Use clear naming, setup methods, and consistent structure.

6.1.8 Practices

Remove dead code and unused variables promptly. Dead code clutters the codebase and makes maintenance harder. If it is not used, remove it. Version control will preserve history if this is ever needed.

```
# Bad: unused variable and commented–out function
def process_data(data):
    temp = 42 # unused
    # def old_processing(): ...
    ...

# Good: keep only what is needed
def process_data(data):
    ...
```

Refactor code continuously to improve clarity and structure. Refactoring keeps the codebase clean and manageable. It is easier to maintain clean code than to clean up a mess later.

Use tools such as linters and formatters to enforce coding standards. Automated tools (e.g., `black`, `flake8`, `pylint`) help enforce consistency and catch style issues early.

Perform regular code reviews in addition to code inspections to maintain code quality. Code reviews help find issues early, encourage shared ownership, and spread knowledge across the team.

6.2 Principles of good design

FAQ: What is software design? Where the handbook mentions software design, what are you talking about?

Effective software design is a cornerstone of maintainable, robust, and scalable systems. While surface-level concerns such as user interface aesthetics or code formatting practices may contribute to a positive development experience, the internal architecture of a software system determines its long-term viability. This section outlines foundational principles that guide the organisation of internal software structure, focusing on modular decomposition, separation of concerns, control of complexity, and the relationships between software components.

6.2.1 Modularity and decomposition

FAQ: What is the main issue we should think about when designing software?

At the heart of good internal design lies the principle of modularity: the division of a software system into discrete, self-contained components known as modules. A module may refer to a class, a file containing some functions, a package, etc. Each module encapsulates a specific subset of the system's functionality and interacts with other modules through well-defined interfaces. This decomposition facilitates independent development and testing, enhances reusability, and reduces the cognitive load on developers by allowing them to reason about individual modules without the need to comprehend the entire system at once.

Modular design further supports evolution and scalability. When new features are added or existing ones are modified, changes can often be localised to specific modules, thereby minimising the risk

of unintended side effects. To be effective, however, modularity must be implemented with attention to both cohesion and coupling.

6.2.2 *Cohesion and separation of concerns*

FAQ: How do we decide what goes into a particular module and what does not?

Cohesion refers to the degree to which the elements within a module contribute to a single, well-defined task or responsibility. A module is said to be highly cohesive when its constituent functions and data structures are directly related to one another and to the module's primary purpose. High cohesion fosters clarity and purpose within a module, making it easier to understand, test, and modify.

Closely related to cohesion is the separation of concerns. This design principle advocates for dividing a system into distinct features or behaviours, each addressed by a separate module or layer. For example, data persistence, business logic, and network communication should each be handled by different parts of the system. Separation of concerns promotes clearer boundaries between responsibilities, reduces duplication, and simplifies the process of identifying and correcting defects.

Both cohesion and separation of concerns are undermined when a module takes on multiple, unrelated responsibilities or when the responsibilities of one module leak into another. Therefore, careful analysis and discipline are required during the design phase to ensure responsibilities are appropriately distributed.

6.2.3 *Minimise unnecessary complexity*

FAQ: Should we design a solution that considers our future plans, or is it ok to keep our design as simple as possible?

Another fundamental principle of internal software design is to keep the design as simple as possible. While some level of complexity is inherent to any non-trivial system, complexity that does not contribute directly to the system's requirements or maintainability constitutes a liability. Complex designs are harder to understand, more error-prone, and more costly to modify.

To manage complexity effectively, designers should favour simple and predictable patterns over intricate or speculative architectures. Generality should be introduced only when it is justified by demonstrated need, not in anticipation of hypothetical future scenarios. Adhering to the maxim "Do not add functionality unless it is necessary" helps prevent the accumulation of design debt and preserves the system's conceptual integrity. In other words, design for what you need now, not for what you currently think you will need in the future. Future plans often do not materialise or change, so design work intended to solve future problems risks complicating your system unnecessarily and wastes current time. Instead, keep the design simple but write the code in a way that makes it easy to change as your requirements evolve.

tions
nised?

6.2.4 *Coupling and component interactions*

Coupling describes the degree of dependency between different modules. In a well-designed system, modules should be loosely coupled. That is, changes to one module should have minimal impact on others. Ideally, modules are couples with as few other modules as possible. Loose coupling is achieved by minimising shared knowledge, using abstraction barriers (e.g., interfaces or abstract classes), and avoiding tight interconnections such as global variables or hard-coded dependencies.

Loose coupling enhances a system's flexibility and adaptability. When modules interact through minimal, stable interfaces, it becomes possible to replace or refactor one module without extensive changes elsewhere. Conversely, high coupling leads to brittle systems where small changes ripple unpredictably, increasing maintenance overhead and regression risks.

It is important to note that coupling and cohesion are not independent: increasing cohesion within a module often leads to reduced coupling between modules, thereby reinforcing both qualities. Thus, a coherent approach to modularisation simultaneously advances multiple design goals.

6.3 *Software testing*

FAQ: What is the role of software testing?

Rigorous software testing is essential to ensure the reliability, correctness, and maintainability of any non-trivial software system. Testing provides empirical evidence that a system behaves as intended under specified conditions, and it serves as a safeguard against the unintended consequences of code changes. In modern software engineering practice, testing is not a one-off activity but an ongoing discipline that accompanies software development throughout its lifecycle.

6.3.1 *Automated vs manual testing*

FAQ: What is the role of manual testing?

Manual testing Manual testing typically involves exploratory activities performed by human testers, who interact with the software in search of unexpected behaviours, usability flaws, or integration issues. Manual testing can be valuable in uncovering edge cases that are difficult to anticipate programmatically. In practice, this looks as follows. As a developer, you should be developing a comprehensive test suite alongside the source code. It is also good practice to run the application and experience interacting with it yourself. In the process of doing so, you may encounter bugs (even if you produced a substantial test suite and all tests pass). When this happens, write an automated test that replicates the conditions causing the bug.

Manual testing also plays a role in user acceptance testing and interface evaluation. Use it to assess whether the user interface is consistent, information is easy to find, the UI scales well to cope with large volumes of data, and user objectives can be achieved

in an intuitive manner. It is also crucially important to thoroughly test a deployed system (especially prior to submitting it for marking!). Make sure that the use can log in with the provided user credentials and that all features work normally within a production environment (which is more restrictive than a testing environment). Also ensure that all static resources load normally.

FAQ: What is the role of automated testing?

Automated testing Automated testing consists of code that executes part of your source code and compares its actual behaviour with the expected behaviour. Automated testing allows tests to be defined once and executed repeatedly at negligible cost. The test code specifies how the source code is required to behave, thereby providing software specifications. It provides a reproducible safety net, making it possible to detect regressions from the required behaviour, verify correctness under a wide range of conditions, and support continuous integration and deployment practices. Importantly, automated tests can be run frequently, on every commit if desired, ensuring that defects are detected as early as possible, when they are least expensive to fix. While certain aspects of software need to be tested manually or inspected, development teams should rely on automated tests as much as they can.

FAQ: Can we substitute automated testing by more extensive manual testing?

Manual testing instead of automated testing? Manual testing is inherently limited by its cost, subjectivity, and inconsistency. It cannot be repeated frequently or reliably across versions, and it does not scale to large or rapidly evolving systems. While manual testing can *complement* automated approaches, particularly during exploratory or usability evaluation, it must not substitute for automation. Software systems should be designed and implemented with the explicit goal of maximising test automation. Without it, long-term quality assurance is infeasible. Therefore, in the group projects, you are expected to produce software systems with comprehensive automated test suites.

6.3.2 Evaluating test quality

FAQ: Why should we evaluate test quality?

The presence of automated tests alone does not guarantee adequacy. In fact, as Dijkstra famously observed "Program testing can be used to show the presence of bugs, but never to show their absence!" To *minimise* the risk of missing bugs, it is necessary to assess the quality and coverage of automated test suites, and address shortcomings. While no approach to evaluating test quality can guarantee an automated test suite is adequate, it can help identify and limit shortcomings.

FAQ: How do code coverage tools help evaluate automated test suites?

White-box testing and code coverage White-box testing techniques assess the structure of the source code itself. One of the most commonly used white-box tools is code coverage analysis, which measures the proportion of the code base executed by a test suite.

Coverage metrics may include:

- Statement coverage: whether each line of code has been executed.
- Branch coverage: whether both the true and false paths of conditional statements are exercised.
- Path coverage: whether all possible execution paths through a program are tested.

While high coverage is desirable, it is not a substitute for meaningful tests. Superficial assertions may lead to high coverage without truly verifying correct behaviour. As such, coverage analysis should be used in conjunction with thoughtful test design, not as an end in itself.

FAQ: Is high code coverage enough to have a good automated test suite?

Black-box testing Code coverage tools can identify obvious problems with automated test suites. However, high code coverage does not guarantee that a test suite is good. If your team relies solely on code coverage statistics to assess test quality, the developers in your team are incentivised to produce spurious tests that simply execute much of the source code, without making a meaningful attempt to find errors.

Black-box testing techniques usefully complement code coverage reports. They evaluate the software from an external perspective without regard to internal implementation details. Black-box quality assessments often include:

- *Equivalence partitioning*: dividing the input space into equivalence classes and testing representative values from each.
- *Boundary value analysis*: testing inputs at and near the edges of valid input ranges.
- *Fuzz testing*: supplying random or malformed inputs to uncover unexpected behavior or crashes.

These approaches are particularly useful for identifying input-handling errors and integration issues that may not be visible through structural analysis alone.

6.3.3 Ensuring Comprehensive Test Suites

FAQ: Who should be writing automated tests?

Who? Developers should write automated tests for their own code. They should have the best understanding of the requirements to complete the task they were assigned, and they developed the solution.

Outsourcing tests to other members of the team is both inefficient and risky. The tester needs to familiarise themselves with the task at hand, something the developer already did. The tester is likely have questions for the developer to help them understand the task or the solution. The tester may also fail to recognise certain flaws in a solution and not test for these.

FAQ: When should we write automated tests?

When? Tests should be written as soon possible, either before or concurrently with the source code. Test writing slows down the rate at which a developer can produce functionality. It can also be a little be boring. Do not procrastinate, however. The longer you wait to write tests, the worse your understanding of the task and solution approach will be. Therefore, postponing automated test writing is likely to lead to poorer tests. While your team is using source code not covered by automated tests, bugs introduced in that source code are much harder to discover. Consequently, if you postpone test writing, the team will benefit less from the tests.

FAQ: What strategies can we adopt to ensure we produce good test suites?

How? Achieving comprehensive test coverage is not an incidental byproduct of software development but a result of deliberate practices and cultural norms. Several strategies can help ensure that test suites are developed in tandem with, and at the same quality level as, production code:

1. *Test-Driven Development* (TDD): In TDD, developers write tests before implementing the corresponding functionality. This approach ensures that testing is an integral part of the development process rather than an afterthought. TDD works particularly in combination with black-box testing.

2. *Continuous Integration* (CI): Automated test execution should be embedded into the build pipeline using CI tools. Builds should be considered invalid unless all

3. Code Review Standards: Code review processes should include evaluation of test coverage and quality. Contributions should not be accepted without accompanying tests, unless explicitly justified. The Semester 1 training materials explain how you can set this up for a Django project in GitHub.

4. *Coverage thresholds*: Enforcing minimum coverage levels using tools such as coverage.py, ensures that all changes meet a baseline level of test thoroughness. While this does not guarantee adequacy, it discourages neglect. Coverage statistics are also easy to verify, without meaningfully adding to the workload involved in your quality assurance process.

5. *Reward thorough testing*: Developers in your team must be motivated to write effective tests. If your team's meetings or accountability sessions focus solely the functionality that team members have implemented, then you are motivating team members to produce source code at the expense of quality assurance. Do not consider a task complete unless it comes with a comprehensive test suite, and give people who produce excellent test suites recognition for their work.

7
Small group project

7.1 Group allocation

7.1.1 Approach

FAQ: How are students allocated to teams in the small group project?

Prior to the start of the small group project, you will be allocated to a team of four or five people. This allocation is not random. However, we have no reliable data to predict how well people will work in a team. Academic attainment tends to be a poor predictor.

Instead of trying to predict how people will work in a team, we ask that students make a commitment to their future team, expressing how long and how intensely they are seek to collaborate with team members. Students are then allocated to a team of people who have made a roughly similar commitment. The expectations arising from the commitment you have made will be enforced through the individual marking scheme. Specifically, major downwards corrections will be made where team members fail to meet up to their commitments (along the lines set out in the marking scheme).

To be allocated to a team, you must complete a registration form (available via KEATS) by the registration deadline. You will only be allocated to a team if you complete this form. The form will ask a number of question and remind you of the commitments you are making. However, the key pieces of information are as follows:

- The project schedule you wish to adopt. You will have a chance to select to work along a six week (normal/full) schedule, a four week (shortened) schedule, or three/two week (extra short) schedule. All schedules work towards the same deadline, but the longer schedules start earlier. The six week schedule requires that you organise during reading week and the project proper immediately after reading week. We aim to allocate you to a team where everyone chose the same schedule. You must achieve the pre-requites for participating in the small group project, as set out in Section 7.2 *before* the scheduled start date of the project[1].

[1] If you struggle to keep up with training in Semester 1, give yourself enough time to catch up by choosing a shorter schedule.

- The number of hours you dedicate to working on the small group project per week. You have a choice of 15 hours (intense), 10 hours (normal), and 7 hours (relaxed). You must dedicate this

number of hours in each calendar week (Monday–Sunday) of your chosen project schedule. This time includes team meetings and other work related to the small group project, but excludes independent learning and training. Beware that a higher intensity schedule will leave less time for other modules. It may not be possible to ensure that everyone in your team has selected the same option here, but we aim to achieve as close a match as possible.

- *Optionally*, the gender, and ethnicity you identify with[2]. When provided, we aim to avoid allocating you to a team where you stand out due to your gender identity or ethnicity. Where possible, this is achieved by ensuring you are not the sole member with a given gender identity or ethnicity in your team. Otherwise, we aim to introduce significant diversity in your team. Please note that your gender and ethnicity can only be taken into account if your provide that information.

- Confirmation that you have read and understood the expectations.

[2] We will not ask about disability. If you have a King's Inclusion Plan (KIP) and it is shared with module organisers, you will be contacted separately to address any requirements set out in there.

There will be two rounds of allocation. In the first round, everyone who submitted the registration form on time will be allocated to a team. A second round with a later registration deadline is organised for everyone who failed to submit their registration form on time.. The six week schedule is only available as an option for on-time registration. If you do not submit a registration form before the second deadline, you cannot be allocated to a team.

7.1.2 Rationale

FAQ: Why do you allocate students in the small group project based on student commitment, gender, and ethnicity?

Forming groups for student group projects is a challenging task. The current system has emerged from many years of experience and experimentation aimed at finding a better way to achieve this.

Over the years, we have made many attempts to group students based on academic attainment. These have included using Year 1 marks, in-term tests, and self-assessment. All these approaches have been severely flawed for two reasons. Firstly, while poor academic attainment tends to be a good predictor that student will struggle in a team, high academic attainment is a poor predictor for how well a student will fit in a team. As high academic attainment is much more common in Year 2 than poor academic attainment, academic attainment is a poor predictor for the team a student should be allocated to. Secondly, the most common challenge students face in group project is a mismatch in expected engagement. Specifically, if there are significant difference in expectations regarding the number of weeks students want to commit to the project, or the number of hours per week team members wish work, then some members of that team will struggle. The allocation scheme aims to address that.

Of course, that may raise questions about ability. However, the small group project is not intended to be overly technically challenging[3]. In fact, every student in the class is expected to work to the same standard. Following sufficient training, designing, building, and testing CRUD operations using Django and related Python packages, and using standard software engineering tools should not especially technically demanding for student of the caliber that reach Year 2 of Computer Science course at King's College London. Developing this type of software to high quality standards will probably come more natural to some student than to others, but software quality is collective team responsibility, achieved though careful organisation of the work, and good collaboration and communication.

[3] You have Practical Experiences of Programming (5CCS2PEP) to do that!

7.2 *Prerequisites to participate*

In the small group project, you will need to develop a Django application in collaboration with your peers. Before you are able to participate in such a project, it is essential that you possess the basic skills needed for collaborative software development with the technology stack we are using. In particular, *before you start working on the project*, you must be able to:

FAQ: What knowledge and skills must team members have acquired before starting the small group project?

- Apply fundamental computer literacy skills to set up and manage your development machine, and deploy a Django application using a service such as PythonAnywhere.
- Use the range of development tools we use in this module for Django development. In particular, you need to be able to use git/GitHub version control software to share your work with the team in a professional manner, use version control and Team Feedback tools to ensure that your work is correctly attributed to you, integrate work from branches with that of the main, and use continuous integration to ensure that problems arising from integration are spotted early. In other words, you need to be able to use the tools the project requires.
- Build and refine models, views, templates, and related components required for a full-stack implementation of Create, Read, Update, Delete (CRUD) operations, as well as comprehensive automated test suites of what you have implemented. This should enable about to produce realistic estimates of the effort involved in a task, and take on small, vertical and complete these within a week. You also require sufficient knowledge of Django in order to participate meaningfully in code inspection/review exercises. In other words, you team mates have a reasonable expectation that you can collaborate in all aspects of the development work and produce usable work.
- Manage your time effectively, so that *in every week of the team's development period, you can dedicate about a quarter of your time to*

the small group project. Every piece of work you produce needs to be reviewed and integrated by your team mates. As the team learns the cadence at which it can produce work, new tasks need to be assigned. To enable the team to do this, you need to be produce work for the team consistently in every week you participate in the project. Beware that mark scheme is designed to enforce this expectation.

- Recognise how the project management techniques you have learned the first half of Semester 1 apply to the project, and apply key skills including effective communication, agile planning/scheduling, effort estimation, and risk management.

It takes a considerable amount of time to acquire these skills. Normally, you need to start the learning process from the very start of Semester 1 in order to be ready to start the small group project immediately after reading week. You also need to organise your work on other modules in such a way that you can engage with the small group project team on a weekly basis. Of course, we all learn at a different pace. Therefore, you can participate in the small group project on different schedules.

7.3 Project schedules

FAQ: Why are there small group project?

Software development projects are exposed to risks and challenges. To manage those risks and tackle those challenges, teams need time. With more time, teams have more space to recognise risks and challenges, discuss them, formulate a plan, and execute that plan. The small group project is designed to require six weeks of low intensity work. Teams are strongly recommended to use the full six weeks available to them, though working at a slow pace (up to 10 hours per week, or 25% of the working week), leaving time to complete other work concurrently.

In practice, many students do not follow that recommendation for a variety of reasons. Some are simply not ready to participate in the small group project when the assignment is released because they lack the prerequisite skills and knowledge. Although I would recommend against this, some students choose to focus all of their effort on a single coursework at any one time.

Members of the same team choosing to work to different schedules is a major cause of friction in teams. To address this, three different teamwork schedules are proposed. As part of the allocation process, you must choose one of the schedules and you will be allocated to a team of students who have selected the same schedule. Therefore, by choosing a schedule, you are making a commitment to the team that you join. Each schedule comes with a set of expectations that are enforced through the marking scheme. Think carefully about the implications and risks of selecting a particular schedule. Once you are allocated to a team, it will no longer be possible to make changes. If you find yourself unable to meet the

expectations of a schedule after allocation, the only remaining option will be to apply to defer the small group project by means of a MCF.

7.3.1 Six week schedule

FAQ: What are the pros and cons of choosing the six week schedule?

As the name suggests, the six week schedule spreads out the build of the software over a period of six weeks from the first week of teaching after Reading Week until the deadline. You are strongly recommended to choose this schedule, *provided you are capable of meeting the expectations your teammates will have of you*. The six week schedule has important advantages:

- Because the project duration is spread out, the development pace can be slowed down. This gives teams ample time to design the product, control software quality (through code inspections and review), and correct problems as they arise.
- When a new team starts collaborating, planning is extremely hard because it is nearly impossible to predict what team members are capable of producing in a given amount of time. Over time, you observe the work your team members produce and you have opportunities to inspect the code quality. This makes it easier to plan work. The six week schedule, you have the time to learn about the capabilities of your team mates and improve the quality of your plans.
- Projects are inherently risky. Under a six week schedule, the team has time to recognise any risks that materialise and respond.
- If some of your team's initial arrangements do not work for you, the team has time to incorporate a mid-project review and change its ways of working.

To join a team following the six week schedule, you must meet the following requirements:

- You must meet all the prerequisites specified in Section 7.2 by the end of reading week. It is essential that you can independently implement useful and fully tested user stories in Django, contribute them via a git branch, ensure that authorship is attributed correctly, inspect code, and integrate a git branch in to the main after receiving permission from your team to do so. You must be capable of doing this from the very start, without requiring help from team mates or time to figure this out.
- You must be able to attend a kick off meeting during Reading Week so that the team can hit the ground running when the small group project starts immediately after Reading Week.
- You must be able to balance your work on the small group project with other coursework and study commitments, in-

cluding the substantial Practical Experiences of Programming (5CCS2PEP) coursework taking place concurrently.

- During every one of the six weeks of the project, every team member will be expected to attend the team meetings, take responsibility for an incomplete task, produce a solution by the next week (at the latest!), and present their result and solutions by the next week. This must manifest itself in the form of meaningful weekly code contributions attributed to you on Team Feedback, that eventually make their way to the main branch. Normally, failing to meet these expectations in one of the six weeks will be permitted without MCF.

It is your individual responsibility to meet these expectations. Individuals choosing to join teams operating the six week schedule, but failing to meet these expectations can be very disruptive. To disincentivise this, the individual marking scheme will be applied strictly. For example, failing to produce meaningful weekly code contributions in all (but one) of the project weeks leads to major mark corrections. If, for instance, all team members contribute in only four weeks of the project, then all team members will see their mark reduced because none met the requirements for the six week schedule!

Teams operating a six week schedule must ensure the following:

- The team must not delay the start of the project under any circumstances! In particular, lack of engagement by one or two team members is not a reason to postpone the project. Everyone in your team will have signed up to a six week schedule, so the team should be enforcing that. If certain members do engage belatedly, manage their late integration into the team carefully. Only assign tasks at an in-person meeting. Do not rely on them to complete important task (especially not testing other people's work). Assign small tasks to be completed very quickly. Monitor progress. From the outset, agree what will happen if work is not delivered.

- The project should be treated as a marathon, not a sprint. The team needs to pace itself. Rushing large coding tasks is not going to be sustainable week after week, and this would defeat the purpose of spreading out work over six weeks. You will be doing other work alongside this project, so constant communication within the team is also unsustainable.

- Take the time to agree sensible working practices at the outset. You will be doing other work alongside this project. Therefore, constant engagement with the project is excessive. Set out a regular schedule of meetings. Agree normal work hours, and on what days and times team members should be able to switch off from the project. It is also good practice to reserve some time to review these practices at the midpoint of the project.

Some arrangements may not work out as intended, and the team should have an opportunity to review them.

Beware that it is only sensible to participate in the small group project on a six week schedule if you are certain you are capable of meeting the schedule's requirement by the end of Reading Week. Your mark will be adversely affected if this is not the case. If you select this schedule in the group allocation form, you will be allocated to a six week schedule team and it will not be possible to change teams after allocation.

7.3.2 *Four week schedule*

FAQ: What are the pros and cons of choosing the four week schedule?

The four week schedule is a shortened schedule whereby the project start is delayed about two weeks. The team should organise a kick-off meeting and an initial planning meeting some time in the two teaching weeks following Reading Week, with a view to start coding immediately after this two week period has ended.

To join a team following the four week schedule, you must meet the following requirements:

- You must meet all the prerequisites specified in Section 7.2 within two weeks after reading week. It is essential that you can independently implement useful and fully tested user stories in Django, contribute them via a git branch, ensure that authorship is attributed correctly, inspect code, and integrate a git branch in to the main after receiving permission from your team to do so. You must be capable of doing this from the very start, without requiring help from team mates or time to figure this out.
- You must be able to attend a kick off meeting and a planning meeting with your team mates during the two teaching weeks following Reading Week. It is important that you participate in these meetings so that the team can incorporate you in their plans.
- The four week schedule gives you a little bit more time to prepare to contribute to the small group project. Use this time well to ensure you catch up with the Software Engineering Group Project materials, and complete as much as possible of other significant coursework and study commitments. Pay particular attention to the substantial Practical Experiences of Programming (5CCS2PEP) coursework.
- During every one of the four weeks of the project, every team member will be expected to attend the team meetings, take responsibility for an incomplete task, produce a solution by the next week (at the latest!), and present their result and solutions by the next week. This must manifest itself in the form of meaningful weekly code contributions attributed to you on Team Feedback, that eventually make their way to the main branch. Normally, failing to meet these expectations in one of the four

weeks will be permitted without MCF. However, you will need to catch up.

- You must ensure that the data collected by Team Feedback about your participation and code contributions is correct by the time of submission. Consult the Team Feedback Help pages for advice is this is not the case.

Teams operating a four week schedule must ensure the following:

- The team must start the project on time: not too soon and not too late. Specifically, the kick-off and planning meetings need to take place in the two weeks following reading week. The team must be coding throughout the final four weeks of Semester 1.
  - Do not delay the start of the project under any circumstances! In particular, lack of engagement by one or two team members is not a reason to postpone the project. Everyone in your team will have signed up to a four week schedule, so the team should be enforcing that. If certain members do engage belatedly, manage their late integration into the team carefully. Only assign tasks at an in-person meeting. Do not rely on them to complete important task (especially not testing other people's work). Assign small tasks to be completed very quickly. Monitor progress. From the outset, agree what will happen if work is not delivered.
  - Do not start coding prematurely. This unfair to others who have signed up expecting a four week schedule, not a five or six week one. Premature code contribution will be ignored in assessing individual code contributions.
- The project should be treated as a marathon, not a sprint. The team may be tempted to compensate for the shortened development period by attempting to do more work in each of the four weeks. While the team can certainly take that approach try and produce six weeks worth of functionality, there is a significant risk that this will undermine code quality and testing. In general, it will be better to scale down the ambition a little, even if the team is prepared to work at a faster pace than a team on a six week schedule.
- Make sure to develop tests and inspect code throughout each of the four weeks of the project. Resist the temptation to get ahead after a later start by focussing solely on functionality. Such a practice would undermine testing and code quality, limiting your mark irrespective of how much functionality the team managed to implement.
- Take the time to agree sensible working practices at the outset. You will be doing other work alongside this project. Therefore, constant engagement with the project is excessive. Set out a

regular schedule of meetings. Agree normal work hours, and on what days and times team members should be able to switch off from the project.

The four week schedule is a compromise approach between a six week and a three/two week schedule. It provides more time to prepare for participation in the group project, whilst also leaving time for quality control, testing, building at least some experience of what team members can produce in a week. Note that to participate on a four week schedule, you will have to be able to contribute within two weeks after reading week. You will also have to plan ahead as there will still be some overlap with the 5CCS2PEP coursework. Your mark will be adversely affected if this is not the case. If you select this schedule in the group allocation form, you will be allocated to a four week schedule team and it will not be possible to change teams after allocation.

7.3.3 Three/Two week schedule

FAQ: What are the pros and cons of choosing the three/two week schedule?

The three/two week schedule condenses the small group project into the shortest possible time period. The idea of this schedule is that students complete all other coursework due during Semester 1 before starting the group project. They then focus all their attention on the small group project. *The three/two week schedule is **not** recommended because it is inherently risky*. It is offered because, every year, there are teams that adopt this schedule, irrespective of recommended practice. If you choose to work in this way, familiarise yourself with the risks. Note that if you chose to expose yourself to these risk, it is your responsibility to mitigate them and live with their impact!

The risks of this shortened project schedule include the following:

- Relatively minor setbacks can have a disproportionate impact on your participation in the project. For example, a bout of influenza or a bad cold can prevent you from participating in the project long enough that you will need to defer. As you would be completing the project in late Autumn, this is a very real possibility.

- Quality control ensuring high standards of design, code cleanliness, and testing takes time. Normally, code is produced along with an automated test suite and then refined and improved. By shortening the project duration to only two to three weeks, the team leaves itself little time to review work in order to ensure standards are upheld. Moreover, by completing the project closer to the deadline, the team may feel it needs to rush work. Consequently, quality standards are likely suffer.

- Planning and scheduling will be difficult. Because the project duration is so short, teams do not have time to experience the

standard and timeliness of work each team member is capable of producing, and then use that experience to plan work in the next cycle. Once tasks are allocated, it will be difficult to predict whether and to what extent these will be completed. Some anticipated features may not be deliverable.

- In a longer project, trust is developed in a team by team members delivering good work in a timely manner. With a severely shortened project, there is little time for this. It will be more difficult to build trust in your team.

To mitigate these risks, teams can try to adopt the following practices.

- Organise a kick-off meeting as soon as possible after team allocations are released. Even though you will not start working on the project any time soon, it will be difficult to convene the team if you do not do this. At this meeting, you will agree when you will start the project and schedule and date, time, and place for your team's project initiation meeting. That will be the start of the project. Chances are that multiple team members do not engage with requests to organise a kick-off meeting. If so, proceed nonetheless, record your meeting minutes (and their absence) on Team Feedback, and record what you have agreed.
- It is a good idea for someone to send a reminder of the project initiation meeting about a week before it is due to take place. Irrespective of whether a reminder is sent, everyone should be attending the project initiation meeting.
- From the outset and at all times during the project, limit your ambitions related to the scope of the project. Attempting to implement too much functionality is going to affect the quality of your project and this, in turn, may cap your eventual mark (irrespective of how much functionality you produced).
- Start quality control from the outset. Testing, and code inspections/reviews cannot be delayed.

7.3.4 *General considerations*

FAQ: Will choosing a particular schedule affect my mark or how I will be assessed?

Assessment The schedule you choose only affects the development period of the project. The length of a team's development period is a factor in assessing major mark corrections for a team (see Section 7.8.3). Otherwise, all teams are marked in exactly the same way.

A team that develops their small group project on a longer schedule will have more time to produce work, quality control the software being produced, and respond to emerging risks. Therefore, it stands to reason that, on balance, teams adopting a longer schedule will tend attain higher team mark.[4] However, the expectations imposed on individual students in teams following a shorter

[4] We have never employed schedule-based allocation before, so this is purely speculative at this point.

schedule tend to be considerably less demanding. If you cannot meet the expectations of a particular schedule, your individual mark will be affected in three ways. Firstly, you will lose one or more weeks of development, leading to a major downward correction of your individual mark. Secondly, you will find it difficult to catch up with the progress of your team. Thirdly, your track record of poor productivity will cause your team mates to lose trust in you, and it will take some time to rebuild that trust.

Choosing a schedule in the registration form is to make a commitment to your future team. To maximise your marks, you should the longest schedule where you will still be able to meet the associated commitments.

FAQ: Can we choose when the project starts and ends, provided the duration of the schedule is adhered to?

Schedule start dates It is essential that the whole team works on the project at the same time! It is not ok for members of the same team to work at different periods of the Semester. That would prevent collaborative quality control. Such an approach would also leave a subgroup of the team with final responsibility for completing the project towards the end.

The end date of the project is the same for all teams: i.e. the project submission deadline. The project must take place during a continuous period that ends in the project deadline. If your team follows the four week schedule, the project must start about four weeks before the deadline and continue up until the deadline. If your team follows the three/two week schedule, the project must start about two or three weeks before the deadline and continue up until the deadline.

Teams should not start development earlier than the project requires. This would exclude team members who signed up to a shorter schedule in the expectation that that would give them time to prepare to engage with the project by a particular date. Of course, a team can start development sooner than required if all team members *unanimously* agree to do so. Please bear in mind that it is advisable to have one or two team coordination meetings before the scheduled start of the project. Only development should not start until the project start date.

If a team is awarded an extension as mitigation for an MCF, the extension should not be used to delay the start time of the project.

FAQ: What is the "normal" schedule?

Default schedule You should pick the schedule that is right for your situation, just prior to reading week. Ideally, everyone would follow the six week schedule. In practice, a substantial proportion of the class is unable to meet the obligations associated with that schedule for a variety of reasons. Some students are behind on the independent learning activities associated with this module. Some students are too busy working on coursework for other modules to engage with the small group project. If you pick an overly demanding schedule, you will probably join a team that produce a better result overall. However, individually you are likely to end up with

a worse mark and you will end up frustrating and alienating fellow students.

FAQ: I am no longer able to meet the expectations of the schedule I have chosen. Can I change schedule or team?

Changing schedules Before the registration deadline, you can submit the registration form repeatedly. After the submission deadline, the information included in your final submission will be used to allocate you to a team. Everyone in the team must follow the same schedule. In other words, you cannot unilaterally decide to deviate from your chosen schedule because of new circumstances. Team changes will not be possible either.

If you are affected by extenuating circumstances beyond your control, please used the College's mitigating circumstances process to address this. For more information, please see Section 2.4.

7.4 Assignment

The small group project assignment will be released when the project starts. You will be tasked with developing an information or content management system with Django. In essence, such a system consists of a database, application logic implementing CRUD operations, and a web based User Interface (UI) to interact with the CRUD operations.

FURTHER READING: The assignment will be released on KEATS when the project starts. Navigate to *Small group project* → *Handbook* to find it.

The assignment will be described in the form of *objectives* and *priorities*, *not* as *requirements*. Objectives describe what a client or a business needs – what is of value to them. Requirements describe the features the software needs to provide. As the assignment provides objectives, part of the task is to convert the objectives into requirements.

EXPECTATION: Teams are *not* expected to try and meet all objectives. The project scope should be a conscious decision by the team based on the team's effective size, the team's ability and productivity, and expectations other that software functionality that the team should meet. (see Team marking scheme for more details).

Teams decide the scope of what they will produce. Team may decide to implement more or less functionality depending on the team's specific circumstances, and standard of work they can deliver. A range of other aspects of the team's work are also assessed, including design, code quality, version control, deployment, and certain aspects of project and team management. Certain improvements in the application's functionality will not increase your team's marks unless those other criteria are addressed.

7.5 Technology and tools

7.5.1 Technology constraints

FAQ: What languages, frameworks, and technologies are allowed in the small group project?

This project must be developed entirely with Python, Django, Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and Bootstrap. You can install any Python/Django packages, as long as these can be installed through PIP as needed. These must be recorded, with the versions you use, in the `requirements.txt` file. Even though the tools you are using here do rely on other languages/technologies, your team must not produce code in any other languages or technologies, including Javascript/Typescript.

Using Bootstrap components that rely on Javascript are fine, but you should produce your own Javascript in the small group project.

If you have experience with web development, you will know that this makes some tasks a little harder. Some problems are solved more easily and effectively with front-end languages. However, these technology constraints aim to ensure most of what you need for the project is learned through the independent study component. As no team member can be excluded from the project due to technology choices, it makes the project more inclusive. Therefore, these technology constraints will be enforced.

Some *starter code* will be released with the small group project assignment. This starter code provides some initial pages, a User model and corresponding database migration, features to log in, log out, and constraint access based on log in/out status, database seeder code, and extensive test code. You must extend your application from this starter code. The starter code does not contribute towards the assessment. For example, the automated tests included with the starter code are not counted towards testing marking criteria and only the additional tests you add are considered.

7.5.2 Tools

FAQ: Which development tools should be used in the small group project?

In the small group project, the team must use the following tools:

- Team Feedback: The team must use Team Feedback as outlined in Section 3.6. Pay particular attention to ensure that meeting attendance records are accurate, minutes are recorded in detail, collaborative coding sessions are recorded by the committer, and code contribution statistics are correct. At the end of the project, Team Feedback is used to administer the peer assessment exercise.
- Git and GitHub version control: your code must be produced and shared via a single git repository. This repository must be shared via Team Feedback.
- Trello: your team must maintain development tasks via a single Trello Kanban board. This board must be shared via Team Feedback.
- The code must be derived from the starting code. Make sure that all build automation tools work specified in the original `README.md` file. Do not introduce new requirements for creating the database, seeding the database, running the code in development, running the tests, and produce code coverage statistics.

7.6 Project management approach

FAQ: How should the small group project be project managed?

The Small Group Project Handbook outlines a detailed project management approach, including meeting agendas and meeting

templates. This approach is based heavily on the project management approach we discuss in the lectures. Teams are expected to follow this approach carefully. To ensure that that teams follow this approach, some observable features of the approach are considered in the assessment criteria. However, the main reason why teams should adopt the specified approach is to promote team productive and manage risk.

7.7 Deliverables

FAQ: What are the deliverables for the small group project?

The team's deliverables at the end of the project consists of three parts:

FURTHER READING: On KEATS, navigate to *Small group project* → *Handbook* to find a detailed description of the project deliverables.

1. A KEATS submission: On KEATS, you must submit a single ZIP file containing everything needed to install, configure, run, test, and evaluate the application. This submission must include a `README.md` file, the source code, the database migrations, the test code, the `requirements.txt` file, and a self-assessment report named `self-assessment.pdf`. To produce the self-assessment report, teams will be given a LaTeX document to fill out and compile. Take care *not* to include the git repository, the virtual environment, or the sqlite database file.
2. A deployed system: The web application must run on a production server accessible through a publicly available URL. The URL must be provided in the README.md file. The deployed system's database must be seeded with a substantial amount of data. The system must be accessible with username/email and password combinations provided with the assignment.
3. Team management data on Team Feedback: Team Feedback must have access to your team's Git repository and Trello board. All team meetings must have been minuted within 24 hours of each meeting taking place.

7.8 Assessment

7.8.1 Team marking criteria

FAQ: What are the criteria used to mark the team in the small group project, and how does the team's effective size affect they way these criteria are interpreted?

The team deliverables of the small group project are primarily assessed on a range of criteria. The most important are functionality, design, code, and testing. However, there are other criteria as well that assess specific aspects of your submission. The functionality criterion refer to the scope of the software: the range of features that were implemented, how well they focus on project objectives, how consistent their UI, how polished their implementation is, and the overall ambition and technical achievement of the team's work. Functionality is assessed in relation to the teams effective size. The design and code criterion refer to the quality of your source code and the overall design of your system. The expectations are based on Martin's books on "clean code" [Martin, 2009] and software

architecture [Martin, 2017]. The testing criterion assesses the coverage, comprehensiveness, and depth of your automated test suites.

7.8.2 Team marking scheme

FAQ: How is the team mark for the small group project decided?

The team marking scheme consists of a set of marking criteria, organised in 20 percentage point bands. They are labelled, from lowest band to highest: I to V. Each marking criterion is labelled to be either criticalor necessary. The criteria of any band above 20% are only considered if *all* the criticalcriteria of the bands below it are met. In other words, if a submission fails to meet a criticalcriterion in a particular band, marking will be constrained by that band. Therefore, it is important to consider all marking criteria, especially those in lower bands. A necessarycriterion should be met as failing to meet them will affect the team's mark. However, failing to meet a necessarycriterion will not prevent the team from meeting higher band criteria. All band IV and V criteria are critical. Criterion VI.0 requires that all necessarycriteria are met. Therefore, teams that fail to meet a necessarycriterion cannot attain band V.

Each marking criterion is assigned a certain amount of marks, adding up to 100%. If a criterion is met, the associated marks are awarded. If a submission fails to meet a criterion, no marks are assigned for that criterion.

FAQ: What criteria must be met in order to attain a team mark in the 0–20% range (band I) in the small group project?

Band I: basic project requirements (0–20%) This band contains the essential requirements for a basic submission that includes minimal working software and everything the markers need to assess the team's work.

I.1 *Functionality/critical* (6 marks): The team has delivered a Django web application that capable of displaying pages with content (static or dynamic). There are enough such pages relative to the team's effective size. The application also delivers some content dynamically retrieved from a database. The scope of this work is sufficient in scale for the given team size.

I.2 *Design/critical* (4 marks): The source code includes working models, views, and templates.

I.3 *Version control/critical* (4 marks): The marking team has access to the project's development history through a git repository shared via Team Feedback. It consists of mostly small commits throughout.
Hint: Break down each development task you are assigned into smaller coding sub-tasks that you can tackle in a short period of time (e.g. 15–30 minutes). Commit each solution to a subtask right away with an informative message. Always push your local repository immediately after committing.

I.4 *Project management/necessary* (3 marks): Team Feedback contains an accurate record of the team's meetings. This

should include a record of at least one meeting per week during the team's development period, with an accurate attendance record.
Hint: Assign one person in the team the responsibility to record each meeting on a laptop during the team's meeting. Always record attendance accurately.

I.5 *Delivery/necessary* (3 marks): The team's submission meets the directory structure and file/directory naming requirements to the letter. All required content is included. The `README.md` file and the self assessment document have been filled out in full.
Hint: Prepare your submission collectively as a team. Read the instructions carefully. Have one person check the work of another! Make sure the code can still be installed, seeded, tested, and run from the submission file.

FAQ: What criteria must be met in order to attain a team mark in the 20–40% range (band II) in the small group project?

Band II: requirements for an adequate team submission (20–40%) The criteria of this band are assessed *only if all critical* requirements of band I are met in full. In combination with the band I, the criteria of this band reflect everything the team must achieve to attain a basic pass of the project.

II.1 *Functionality/critical* (6 marks): The application enables users to create, read, update, and delete new data, using forms as necessary. Such features are combined into tools that are useful for some of the intended end-users. The scope and scale of the application is adequate for the team's effective size.

II.2 *Design/critical* (3 marks): The software uses Django forms to generate a forms (not raw HTML).

II.3 *Code/critical* (3 marks): The code is free from significant defects or major bugs.

II.4 *Testing/critical* (3 marks): The source code includes some new automated tests, at least two per effective team member.

II.5 *Delivery/necessary* (5 marks): The submission is evaluated with a specific set of build automation commands (specified with the assignment) to install a virtual environment, install required packages, setup and seed/unseed the database, run the tests, generate a code coverage report, and run the development server. The markers will log into the server with predefined user access credentials. These must be available to the markers exactly as specified so that they can assess the team's work *without* (!) having to review the README.md file, analyse the code, or perform minor debugging.
Hint: Prepare your submission collectively as a team. Double or triple check: have one person check the work of another!

pe met in
k in the
the small

Band III: requirements for a fair team submission (40–60%) The criteria of this band are assessed *only if all critical* requirements of the lower bands (bands I and II) are met in full. In combination with the preceding bands, the criteria of this band reflect everything the team must achieve to attain a fair grade for the project.

III.1 *Functionality/critical* (3 marks): The application possesses a set of working features that is moderately ambitious, given the team's effective size.

III.2 *Management/critical* (3 marks): The application's feature set is largely focussed on supporting a cohesive set of project objectives. Isolated features that do not contribute to a fully supported/implemented objective are largely avoided.
Hint: Achieving this criterion requires careful task allocation. Define tasks that meet the INVEST criteria and only ever assign tasks that add value now.

III.3 *Design/critical* (2 marks): The software employs the Django framework effectively, making good use of models, views, templates, forms, and other components. To meet this criterion, the aforementioned components must be present in the source code and used for the purpose Django intended.

III.4 *Code/critical* (3 marks): The code is reasonably clean. Variables, functions, methods, and classes mostly have suitable names. The code layout is mostly consistent. Whitespace is used mostly consistently to separate functions, methods, classes, and other components. Excessive/unnecessary whitespace is mostly avoided. The code base mostly uses the same indentation symbol.
Hint: Small lapses of judgement will be tolerated in this band, but make an effort. Unless everyone's code is inspected by someone else before merging it with the main, it is impossible to enforce a basic level of code cleanliness.

III.5 *Testing/critical* (3 marks): The software comes with a test suite with good statement and branch coverage. Failed tests are largely avoided. Normally, each individual module should achieve at least 70% statement coverage, and overall, 90% of tests should pass.
Hint: Each developer should write a test suite for their own source code as and when they write that source code. Do not postpone writing tests. Before merging a branch, inspect the test coverage of new source code. Never merge a branch that causes tests to fail into the main branch until the problem is resolved. Never ever delegate test writing to give a previously disengaged team member in order to give them something to do: this is a recipe for failing this criterion.

III.6 *Deployment/critical* (2 marks): All the application's features work in both the development and the deployed version of

the software. The features are accessible via the user credentials specified in the assignment. There are no features that have not been deployed. The deployed version of the software is seeded with a substantial database, containing a sufficient volume records to perceive the application at scale.
Hint: Start deploying a version of the application early and keep it updated. Use a database seeder to seed the database as this will avoid problems. Require every member of the team to test the deployed version immediately. You cannot blame failing this criterion on another member of the team: the whole team is jointly responsible for meeting it.

III.7 *Code (comments)/necessary* (2 marks): The source code documents all (public) classes, methods, and functions that may be called from other modules. All noise comments, including "To Do" comments and commented-out code, have been removed.
Hint: Employ code inspections. Require comments to be "clean" before merging a branch with the main.

III.8 *Code (file structure)/necessary* (2 marks): The source code is organised into a sensible file structure that meets Django and Python conventions. Excessively large files and directories containing too many items at the same level are large avoided. For the purposes of marking, the maximum number of lines in a source code file is set to 400, and the maximum number of subdirectories and file in a directory is set to 30.
Hint: Employ code inspections. Follow the naming/structure conventions used in the teaching materials. Aim to keep file and directory sizes well below the limits.

FAQ: What criteria must be met in order to attain a team mark in the 60–80% range (band IV) in the small group project?

Band IV: requirements for a (very) good team submission (60–80%) The criteria of this band are assessed *only if all critical* requirements of the lower bands (bands I–III) are met in full. In combination with the preceding bands, the criteria of this band reflect everything the team must achieve to attain a good or very good grade for the project.

IV.0 *Delivery/critical* (0 marks): All necessary criteria mentioned above are met. No marks are assigned to this criterion. However, if a submission fails to meet a necessarycriterion, the team's mark is capped at this band.

IV.1 *Functionality/critical* (3 marks): The application can be used to achieve an ambitious range of objectives. The application's feature set is largely focussed on supporting a cohesive set of project objectives. Isolated features that do not contribute to a fully supported/implemented objective are largely avoided.

IV.2 *Functionality/critical* (3 marks): Features are fully developed, offering an intuitive and flexible interface to end users. Expectations include (but are not limited to) the following. Dates/times are entered in an intuitive format consistent with UK conventions. Individual records can be identified or selected without using data not intended to be used by users (e.g. primary keys, unless these have a special meaning). Lists come with a range of facilities to navigate them, including pagination, ordering, and searching.
Hint: When developing a new feature, start by ignoring this criterion and produce a basic working version. Then, gradually add new tasks to the backlog to incrementally refine existing features. Do not forget that the new code associated with this criterion requires cleanup and refactoring! This takes time.

IV.3 *Management/critical* (2 marks): The application's user interface is consistent throughout. Information screens, forms, and lists have the same look and feel. The language/terminology used throughout the user interface is consistent. Similar features have the same functionality set. Where different screens possess equivalent controls and information, these can be found in the same place with the same look and feel.
Hint: The level of coordination required to achieve this marking criterion goes substantially beyond that of the previous band. The team will need discuss, agree, and document the organisation, look, and feel of the application and specific types of screens, possibly at multiple stages in the project. User interface inspections will need to be incorporated into the team's task review processes before a task can be deemed completed. This coordination tends to require time and, therefore, a longer schedule.

IV.4 *Design/critical* (3 marks): Views are critical and highly connected modules in a Django applications. The bodies of view functions and methods are small and restricted to control logic. Repetition of policies required by multiple views is prevented through effective reuse of code.
Hint: This criterion requires systematic

IV.5 *Code (Source code)/critical* (3 marks): The source code meets high code cleanliness standards. Variables, functions, methods, and classes *consistently* have clear and descriptive names. Code layout is consistent *throughout*. There are no long functions or methods: no method of function body consists of more than 25 lines in marking. No function or method body has more than 2 levels of nesting. The code is mostly DRY as significant code repetition has been avoided.
Hint: To meet this criterion, a team's quality control standards need to be substantially more rigorous than those required to attain band D/C criteria. Document your code inspections. Auditing code

inspections (i.e. reviewing whether inspectors spot all problems) can be helpful here. Identifying repetitive code normally requires code reviews.

IV.6 *Code (Templates)/critical* (2 marks): High code cleanliness standards apply to templates. Repetitive code in templates is avoided through effective use of template inheritance and template partials. The templates contain no manual styling: CSS classes are used instead as necessary.
Hint: Identifying repetitive code will require a systematic code review. Issues such as local style attributes can be identified in an adequately organised inspection.

IV.7 *Testing/critical* (4 marks): The software comes with a test suite with impeccable statement and branch coverage. All tests pass.
Hint: Before merging a branch, inspect the test coverage of new source code. Set stringent conditions on test coverage. Do not merge a branch if it causes some tests to fail. Consider adopting GitHub Actions to run the test suite.

FAQ: What criteria must be met in order to attain a team mark in the 80–100% range (band V) in the small group project?

Band V: requirements for an exceptional team submission (80–100%)
The criteria of this band are assessed *only if all* requirements of the lower bands (bands I–IV) are met in full. In combination with the preceding bands, the criteria of this band reflect everything the team must achieve to attain an exceptional grade for the project.

V.1 *Functionality/critical* (5 marks): The team delivered an application that is very ambitious in scope and exceptionally polished, given the team's effective size.

V.2 *Management/critical* (3 marks): The team's time management has been excellent. The final three days of the project were free from significant development activity (as manifested by code activity statistics). All development and almost all refactoring took place before this period. This has allowed the team to focus the final days of the project on quality assurance of the submission.

V.3 *Design/critical* (3 marks): The design achieves high cohesion and low coupling throughout. Classes have limited responsibility, ideally a single responsibility. Functions and methods do one thing only.

V.4 *Code (Source code)/critical* (2 marks): The source code meets exemplary code cleanliness standards. Naming is consistent throughout the application. All names make meaningful distinctions. Function and method are extremely short with no more than 15 lines of code and 1 level of nesting. The code includes no repetition.

V.5 *Code (Test code)/critical* (2 marks): High code cleanliness standards extend to the test code. Test code uses clear, descriptive, and consistent names. Test code repetition is minimal. The bodies of test functions/methods are limited to 25 lines and 2 levels of nesting.

V.6 *Testing/critical* (5 marks): Spot inspections of test code shows that test suites have been carefully designed to ensure good coverage of input and output partitions, as well as potential causes of errors.
Hint: This criterion is not concerned with code coverage, but with the range of test cases. To meet this criterion, code inspection must examine test code as well as source code.

Please bear in mind that, at level 5 (Year 2) of the course, marking criteria are more stringent than at level 4 (Year 1). Attaining band V criteria is intended to be challenging. While not impossible, it is far from the norm!

7.8.3 Major mark correction

FAQ: Under what conditions would my individual small group project mark be a severe reduction on the team's mark?

As explained on page 15, it is not possible for the markers to make a precise, objective, and accurate assessments of the extent to which each individual in a team meets the individual expectations. In the small group project – a project where features are implemented incrementally/iteratively, and vertically – team members should be engaged in similar coding tasks. The mark scheme seeks to encourage all team members to attend all team meetings, commit to some work at each meeting, complete their task between meetings, and deliver that work at the meeting succeeding the one where they committed to the task.

To check to what extent that commitment was met, a relatively rudimentary assessment will be made based on a set of metrics:

EXPECTATION: The code contribution statistics are derived from the team's shared repository. Every student is responsible to ensure that their workstation is set up correctly to ensure that their commits are associated with the correct GitHub account. When coding, every student is responsible to ensure that they commit their work and push their work to the shared repository. Use Team Feedback's features to correct attribution errors (see *Team Feedback* > *Help* > *Git troubleshooting* for more help.

1. The number of weeks in which the number of committed line changes exceeds a minimum threshold. This (admittedly crude) metric seeks to assess that an effort has been made to produce work.

2. The number of weeks in which the number of line changes committed to the main branch exceeds a minimum threshold. After commits are merged with the main branch, the line changes in those commits count as changes committed to the main branch, and the commits are attributed to the week the commit was made originally. This crude metric assesses the amount of work the team could actually use, on account of it being merged with the main.

3. The number of meetings attended in full (i.e. where the team member was present on-time, or 5 minutes late).

EXPECTATION: Line counts are not a good way to measure code contribution. However, the thresholds are set low. It is assumed that everyone builds/changes the models, views, templates, and test code for the features they are responsible for (as

Table 7.1 summarises how metrics 1 and 1 are used to compute a major mark correction based on code contribution. In this table, the number n corresponds to the number of weeks that at least one member of the team made a full contribution to the code. The thresholds for the number of committed line changes and the number of line changes committed to the main branch, referred to in metrics 1 and 1 respectively are defined in the Small Group Project handbook.

Table 7.1: Major individual mark correction for code contribution in teams with a development period of n weeks. Weeks are defined as the Monday-Sunday intervals between the start and end of the project.

#Weeks of minimum code contribution	#Weeks of used minimum code contribution	Mark correction
$n-1$ or n	3 or more	no effect
	2	-10%
	1	-20%
	0	-40%
$n-2$	3 or more	-10%
	2	-20%
	1	-30%
	0	-50%
$n-3$	3 or more	-25%
	2	-35%
	1	-45%
	0	-65%
$n-4$	2	-50%
	1	-60%
	0	-80%
$n-5$	1	-70%
	0	-90%
0	0	-95%

Table 7.2 summarises how metric 3 is used to compute a major mark correction based on meeting attendance. Please note that team members ought to attend all meetings: 80% is *not* "enough".

Table 7.2: Major individual mark correction for meeting attendance.

% of meetings where team member is "present" or "5 minutes late"	Mark correction
80% or above	no effect
60% – 79%	-20%
40% – 59%	-50%
20% – 39%	-80%
less than 20%	-100%

The largest absolute value of the two major correction is applied as the major mark correction for individual marking purposes. Most students normally do not get a major mark correction from the team mark. Where major mark corrections are applied, the peer

assessments are reviewed manually to review whether there is a reason not to apply the major correction.

7.8.4 Minor mark redistribution

FAQ: If my individual mark is not severely reduced, do I receive the same mark as my team members in the small group project?

Only students whose mark is unaffected by a major mark correction are considered for minor mark redistribution. This section outlines briefly how the policy discussed on page 14 is applied to the small group project.

After the small group project, you will participate in a peer assessment exercise. The minor mark adjustment is primarily based on the peer assessments. Only peer assessments about students whose mark is not subjected to a major mark correction are considered.

The peer assessment exercise includes a set of multiple-choice questions for each of your peers. Scores are associated with each of the answers. A weighted sum of these scores is calculated to produce a score for each peer assessment by a reviewer about a reviewee. The weights favour engagement and contribution, but no question is weightless. The peer assessment scores for each reviewer are normalised so that each reviewer within a team produces the same average peer assessment scores across the reviewees considered for minor mark redistribution. The peer assessment score for a student is the average of the normalised peer assessment scores they received.

Table 7.3: Summary of the minor mark redistribution policy in the small group project

Peer assessment score	Code contribution statistics	Mark redistribution
Above average by a substantial margin	Above average	+3, +4, or +5 depending on the peer assessment score
Below average by a substantial margin	Below average	-3, -4, or -5 depending on the peer assessment score
Any other combination		between -2 and -2 (inclusive), as necessary to ensure the sum of the adjustments is zero

Table 7.3 summarises how minor corrections are decided. In essence, team members with scores substantially above or below average, and with code contributions that above or below average respectively, are adjusted by a mark of at least 3 and at most 5. The amount depends on the difference between the peer assessment score and the average peer assessment score. All other individual are adjusted up or down by at most 2 to ensure that the sum of all corrections is 0.

The peer assessments manually reviewed to check for anomalies,

such as cases where individuals are dishonest and try to game the system.

7.8.5 Peer assessments

FAQ: How are the peer assessments marked in the small group project?

The peer assessments you write about your team mates are marked directly, focussing on the open-ended feedback text that you write. This contributes towards 5% of the small group project mark. The following assessment criteria are used to produce this mark:

0% The feedback text contains no meaningful write-up.

20% A peer assessment was submitted for almost every team member. The feedback text of each submitted peer assessment contains a single sentence explaining the scores for that peer assessment.

40% A peer assessment was submitted for every team member. Each peer assessment includes at least a 2–3 sentence explanation for the peer assessment scores.

60% A peer assessment was submitted for every team member. Each peer assessment includes at least a 2–3 sentence explanation for the peer assessment scores, recognising strengths and weaknesses, as well as some valid constructive feedback that, when acted on, would help the reviewee in future projects. The text is largely free from grammatical errors.

80% A peer assessment was submitted for every team member. Each peer assessment includes a thorough explanation for the peer assessment scores, recognising strengths and weaknesses, as well as valid constructive feedback that, when acted on, would help the reviewee in future projects. The peer assessment is written thoughtfully, and internally cohesive. The constructive feedback logically follows from the explanation. The text is free from grammatical errors.

100% An unusually extensive peer assessment was submitted for every team member. Each peer assessment includes a thorough explanation for the peer assessment scores, recognising strengths and weaknesses, as well as valid constructive feedback that, when acted on, would help the reviewee in future projects. The peer assessment is written thoughtfully, and internally cohesive. The constructive feedback logically follows from the explanation. The ideas are rooted in project management theory, and sometimes insightful. The text is free from grammatical errors.

In some teams, you may have a team member who failed to engage with the team at all: i.e. they did not attend any meetings or produce any work. Such a person may remain in your team if they signed up to participate in the small group project and did

not defer early on. Obviously, there is little to write about in such a situation. The feedback to write for such a peer can be limited to a single sentence explaining the lack of engagement without affecting your peer assessment mark. If a person's engagement was very limited, but they attended some meetings or produced some work, focus on your interactions with them.

8
Major group project

8.1 Group allocation

FAQ: How are teams formed in the major group project?

In the major group project, you will be working in teams of approximately 8 (± 2) people. There will be a selection of projects and teams can choose their own technology stack. As it is easier to do this in a team of people with shared project and technology preferences, students are strongly encouraged to form their own teams. Students who are unable to find a team can register to be allocated to one, in a similar manner as for the small group project. If you would like to work with some people, but are unable to create a group with 6–10 members, you can create a smaller project team to be extended through allocation. Students in teams that came together with a shared understanding of the kind of project they want to do, and how they wish to work tend to have the best experience.

To find a major group project team, it helps to start networking as early as possible: from the start of the academic year. With the team formation deadline still far away, you have the time to get to know fellow students, what their interests, talents, goals, and values are. The lectures, small group tutorials, and the small group project all offer opportunities to meet and talk to fellow students. While you want to work with people who share similar objectives, values, and project type and technology preferences, make sure to diversify your team. Team tend to benefit from having people with different backgrounds, personalities, and talents.

To register a student-formed team, one person has to create the team on Team Feedback. As team owner, they can then send invitations to fellow students. After receiving an invitation, make sure to accept the invitation.

If the team you form has 6 to 10 members, no changes will be made through subsequent allocation. Outside that range, adjustments will made to the team: teams of 5 people or less will have additional people allocated and teams of 11 members or more are split into two. If you are not invited to a team or you do not accept an invitation to join a team, but wish to participate in the major group project, you must submit a form to request to be allocated to a team. As in the small group project, there are two rounds of allocation.

8.2 Assignment

FAQ: What choice of project assignment will we have in the major group project?

Unlike the small group project, there will be multiple major group project assignments for teams to choose from. There are three types of assignment, though they come with different eligibility requirements, as follows:

- *Self-proposed project*. Teams (not individuals!) can propose their own project. To do so, you must have a complete team and submit a project proposal by the relevant deadline. The entire team must agree to undertake this project. The proposal must be approved.
- *Client-proposed project*. These projects are proposed by a client who requires a software system to be developed to specific requirements. If you undertake one of these projects, you will have several (at least three) meetings with the client. Each client project will have limited capacity, depending on the client's availability to engage. To apply for a client project, you must form a complete team, agree (collectively as a team) to bid for one or more client projects, and submit your agreed client-project preferences by the relevant deadline. You can only undertake a client project if you are assigned one following the bidding process.
- *Academic-proposed project*. These are projects proposed by academics in the department. The objectives tend to be relatively open-ended, allowing the team to design their own project. There are no constraints on this type of project. If your team is not eligible to undertake a self-proposed project or a client-proposed project, you must choose (collectively as a team) any one of the academic proposed projects.

8.2.1 Self-proposed project

FAQ: I have an idea for a self-proposed project. Can I undertake this project instead of one proposed by staff or clients?

If you have a good idea for a project that you, and the team you have formed, can complete within 10 weeks, then you can do a *self-proposed project*. The deadlines to do this are tight, so you need to start early. You will need to assemble a complete team, write a proposal together, and submit it by the relevant deadline. Every member of the team must have read the proposal and agreed to the project. There are some further considerations:

- The proposal must be for a new, self-contained software development project. If any code or part of the system that you will be building already exist, you must declare what that is and share the code. It is critical that the team is assessed on what it produces during the major group project period.
- You must agree clear "intellectual property" arrangements for the software that will be developed in this project. Do not assume that that is something that can be resolved later, as this can be very complicated. I strongly recommend that the team agrees

to release the software under a free[1]. This removes most, if not all, complications. In light of this, it is a good idea to work on something you would enjoy doing, but not necessarily something you would like to commercialise.

[1] "Free" as in "free speech", not as in "free beer".

- I will review your project and return some feedback shortly after the deadline. Where I have concerns, I will try to encourage you to proceed and suggest solutions if I can. However, if your project involves work that is illegal, unethical, or in violation of College regulation, it will be rejected. Projects that require ethics approval (e.g. a project that involves collecting new, personal data) are discouraged because the ethics approval process will delay the project too long.

Unless your self-proposed project has a particular client in mind, I recommend that the outcome of the project is released under a free software license. Other arrangements, such as sharing intellectual property rights amongst the team or assigning it to one member of the team, risk causing conflict within the team during or after the project. If your team does not wish to release the project under a free software license, the team should agree intellectual property arrangements for the project prior to starting the work. I recommend you seek the advice of an experienced solicitor, as I am not qualified to provide legal advice.

8.2.2 Client-proposed project

FAQ: What are client-proposed projects?

We will offer one or more *client-proposed projects*. These are project proposed by a (normally external) client with specific objectives, requirements, constraints, and context. The clients' ideas will have been scrutinised before offering these projects. However, their ideas will not necessarily have been fully worked out.

- If you take on a client project, your team will have to work with the client to develop a suitable solution. Client ideas are usually not fully worked out. Some clients may have a limited appreciation of what is technically possible or how difficult different aspects of the work are. If so, your team will have to advise them and negotiate how to proceed with the project. Consequently, the requirements engineering for client-proposed projects tends to be substantially more complex compared to other types of project.

- The "intellectual property" arrangements for client-proposed projects vary. Typically, the software is either owned fully by the client or it is released under a free software license.

- Capacity on client-proposed projects is limited. If you are interested in one or more client-proposed projects, you must form a complete team, agree what projects you might like to do, and submit that team's preferences by the relevant deadline. Projects will be allocated to team's based on the submitted preferences.

We cannot guarantee that all teams will be able to get a client-proposed project of their choice.

8.2.3 *Self-proposed client project*

FAQ: Can I self-propose a project with an external client?

If you know a potential client and wish to consider working with them, I would ask you to discuss this with me as soon as possible. If they have a viable project, this project can be employed as a major group project, either as a self-proposed project or a client proposed project. In either case, your team will have first right of refusal to undertake this particular project. If the project is handled as a self-proposed project, your team would be the only one undertaking this project. If it is handled as a client-proposed project, multiple teams would be allowed each to produce a solution independently. As with other self-proposed or client-proposed projects, you must form a team prior to the relevant deadlines and ensure that the entire team is on board with the project. In addition, I wish to contact the prospective client, so that I can discuss expectations with them.

8.2.4 *Academic-proposed project*

FAQ: What are academic-proposed projects?

Academic-proposed projects are the default type of project. These are projects proposed by academics to produce an application or system to meet certain broad objectives. Like the small group projects, the scope of these projects can be tailored somewhat to suit your team. They will not come with "client meetings" or highly detailed requirements. Instead, your team will need to design a product around the specified objectives that suits a particular end-user need. This may require some independent research.

There are no capacity constraints on these projects. If your team does not have a self-proposed or client-proposed project, your team must choose one of these. All teams that have lecturer-allocated members in them must do an academic-proposed project of their choice. The outcome of the project is released under a free software license.

8.3 *Technology constraints*

FAQ: What languages, frameworks, and technologies are allowed in the major group project?

In the major group project, teams can choose the programming languages, frameworks, libraries, services, and tools they deem most suitable for the task at hand, provided the following constraints are met.

- They must be free (as in "free beer", not necessarily as in "free speech"). This means that teams must not pay for anything they use.
- Team must use git with either GitHub.com's or King's College London's GitHub Enterprise service. External clients may request deviations from this rule in advance of the project start and

an exception is made to this rule *only* for client-proposed projects where conventional approaches to using git are not possible.

- Employing these tools must involve substantial coding and software design work. For example, creating websites with a WYSIWYG design tool is not permitted.

When choosing a technology stack, it is important to be aware of all the implications of your choice. Consider the following questions.

- *How will you meet the code cleanliness and design requirements for the project?* Certain frameworks and libraries promote sound design and code cleanliness, while others do not. Make sure that the team is comfortable with the technology you plan to produce well-designed software and clean code.

- *How will you use this technology to produce an automated test suite?* The marking criteria impose requirements on automated testing. A good project will require you to produce a comprehensive test suite as well as a code coverage report. Prior to committing to a particular technology, the team should identify the tools they will use to achieve this and ensure the team can learn to use them effectively.

- *How will you deploy your work?* Some teams fail to consider this question before they are firmly committed to a technology and end up tied to a system that is very difficult to deploy. It is essential to deploy your application. Therefore, it is advisable to try deploying some software before committing to certain technologies.

- *How much effort is required to ensure the team becomes productive with a technology?* Some teams choose a technology because one or two team member's strongly advocate for it. However, this is a decision that affects everyone engaged with the work, especially the people on whom a technology choice imposes study requirements. Do not pick a technology that team members will struggle to learn.

- *Is everyone able to install the tools needed for development?* Some technologies come with requirements that some team members are unable to meet. For example, it is obviously a bad idea to build an application with Apple's XCode if a member of the team does not have a Mac.

The requirements listed above tend to be easier to meet if teams build a web application. Modern technology stacks to build web applications tend to come with good testing tools. In deployment, teams have considerable control over the configuration of the (virtual) system the software runs on. Web applications are inherently cross-platform. The scope of a web-based information system is

typically very scalable, enabling teams to adjust what they produce as circumstances change. When building a web-application, make sure your system does not rely on paid-for services, such as payment handling services, and paid-for cloud computing infrastructure, such as Amazon Web Services.

If your team needs to or wishes to build a mobile application, consider carefully whether this needs to be a native application. Often, a web application with a front-end developed for mobile browsers can be nearly indistinguishable from a native application. However, the web version will be easier to test and deploy. You will not gain marks for choosing a technology stack that complicates your work. Instead, as an engineer, you ought to choose the best tools for the job under the constraints you have been given.

Occasionally, a team chooses to build a computer game as their major group project. While this can be a very interesting project to do, such a project comes with considerable risks and challenging. It can be difficult to meet the testing requirements with certain modern game engines. Moreover, a minimum viable product (MVP) for a game can be quite substantial, leaving only limited scope to scale down the application. Often, 10 weeks is a very short time for an inexperienced team to produce a significant game. Finally, a typical game requires multimedia resources and the production of such resources is not recognised in the marking scheme. Thus, game development only a viable option for teams that are prepared to manage and face the risk involved.

Some teams seek to incorporate AI features into the systems that they build. While such features can contribute to the project objectives, it is important to note that (i) this project is a software engineering project (not an AI project), and (ii) the members of the team will have only had limited exposure to AI. Therefore, care must be taken that such features are sufficiently narrow in scope and ambition, so that they are feasible. Moreover, certain sub-symbolic AI techniques require computational resources beyond what a typical consumer computer provides. Do not expect high performance computing infrastructure to be available for your project, unless their need has been identified and resourced prior to the start of the project. You may discuss this with the module organiser if you are unsure whether your ideas are achievable. Do take care to ensure that the AI code produced meets the software engineering standards set out in the marking criteria.

8.4 Project management approach

FAQ: How should the major group project be project managed?

Teams can use whichever project management approaches suit their team and project best. There is no requirement to follow the lean and agile approaches required in the small group project. Teams do not have to maintain a Kanban board. However, the project management approaches of the small group project have been designed to avoid or mitigate many of the risks involved in student group

projects. If your team decides to deviate from this approach, consider the implications carefully.

To ensure it is possible to maintain at least basic levels of *accountability*, teams must meet following expectations (irrespective of the project management approach used):

- Teams must have *at least* one whole team meeting per week. Everyone must attend every whole team meeting. Team members must work on the tasks agreed with the team at team meetings, and only such tasks. Team members not attending meeting, expecting colleagues to tell them what to do or choosing their own work, will not normally be considered full members of the team. Team members must not decide what they work on without consultation of and agreement with the team. Without this, coordination of work between team members becomes impossible.

- Teams must use git version control with either GitHub.com's or King's College London's GitHub Enterprise service to version control their code base, their report document, and any prototype coding activities (including machine learning work). Use separate remote repositories on GitHub for each item your project requires[2]. Register all shared repositories on Team Feedback. Without this, it is impossible to obtain a view of individual engagement with the project.

- Every member of the team must make demonstrable contributions to the project in every week of the project duration. These contributions should normally be visible in the team's repository. Without weekly delivery of contributions, it is difficult for a team to predict what a member can and will contribute. This makes planning unnecessarily difficult. The major marking scheme is based on this expectation: if a peer assessment raises significant concerns, and there is no evidence of weekly contributions, then the scheme is applied.

- Everyone must contribute significantly and meaningfully *both* to the report and the software. Work on the report and the software targets different learning outcomes, and it is important that all team members attain both. Moreover, the report must reflect the reflections of participants of software development project. To achieve this, the people writing the report must be the same people that completed the project.

[2] Team Feedback encourages creating as few repositories as necessary. While you should not create more repositories than necessary, each distinct component of your project can have its own repository.

EXPECTATION: Everyone must contribute substantially to both code and report. Do *not* create specialist roles where certain individuals contribute to only one and not the other. Contributions should be observable in the report and code repositories.

8.5 Deliverables

The team's deliverables at the end of the project consists of four parts:

FAQ: What are the deliverables for the major group project?

1. *Software and report*: On KEATS, you must submit a single ZIP file containing the following items:

FURTHER READING: On KEATS, navigate to *Major group project* → *Handbook* to find a detailed description of the project deliverables.

- *The software*: This includes everything needed to install, configure, run, test, and evaluate the application or system. If the software is a system consisting of multiple components (e.g. a web application and a mobile application that interact with one another), the zip package must include a directory for each component (with everything needed to install, configure, run, test, and evaluate the component of the system). For components containing a database, a database seeder/unseeder script should be provided.
- *The project report*: This is a moderately substantial report that reflects on the project management and software engineering decisions that were made in the development of the software, explain their rationale, and relate it to the theory you were taught in the course. The project report should be jointly authored by all team members. To produce this document, teams will be given a LaTeX document with detailed instructions on what the report must contain.
- A single *developer's manual*: Ensure that this is a file named `developers-manual.pdf` and that it resides at the root of the application. This file must contain detailed instructions how to install, configure, run, and test the software, and each of its components. For components containing a database, explain how to generate data for the database. It is good practice to employ *virtual environments* and build automation as much as possible. Beware that your submission may rely on specific versions of libraries and frameworks, so make sure your specifications are sufficiently detailed.
- A `README.md` file: This file must include the title of the project and the name of the software, the names of all authors of the software, a list of all significant parts of the source code written by others that you employed directly or relied on heavily when writing this software and the locations of this source material. Think of this as the "reference list" for your source code, and the location where the software or software component is deployed and sufficient information to access it. The latter includes access credentials for the different types of user who may employ the software.

2. *Screencast*: On KEATS, you must submit a single video file containing a screencast demonstrating the software. This is a video showing all (significant) features of the software. It is important that the demonstration includes each software component and the typical interactions of each type of operator (user) with the system. Minor features, such as I/O validation, do not need to be included throughout the video. The screencast should be submitted as a single video file (even if the system contains multiple software components). Your screencast should come with clear explanations of what you are demonstrating. The screencast should be narrated, as that tends to be the most efficient

way to incorporate the explanations. The video should be saved MP4/MPEG-4 format. If you are unable to do this, common alternative video formats are acceptable. However, do not use AVI as that tends to create a very large file! You can submit a ZIP file provided it contains only a single video file. Bear in mind that video tend to be highly compressed already: zip algorithms do not tend to reduce the file size significantly.

3. A deployed system: Teams must make a deployed version of the application available[3] Ideally, the deployed system is available from within a web browser on a URL. Most web applications and mobile applications can be deployed that way. Generally speaking, a deployed system is one that is readily available for execution without the need to set up a comprehensive development environment. This is quite a vague statement, but it is impossible to be more precise as your options for deployment will vary considerably from project to project.

4. Team management data on Team Feedback: Team Feedback must have access to your team's Git repositories. All team meetings must have been minuted within 24 hours of each meeting taking place.

[3] Unless you undertake a project with an external client who, for valid commercial reasons, does not allow this. This is normally known in advance of project selection.

8.6 Assessment

8.6.1 Team marking scheme

The team marking scheme for the major group project employs the same approach as that of the small group project. The mark scheme is divided into 10 percentage point intervals, from 10% to 100%. To achieve a particular mark, say 60%, the team's work must meet all the requirements associated with that mark *and all the requirements that precede it*. If a submission does not meet a particular criterion (say at the 60% point), the team will not attain the mark associated with that criterion (i.e. their mark will be less than 60%).

Markers will not consider achievements at a given mark point unless all expectations of all criteria at the lower mark point have been achieved. If a team achieves some of the criteria at a particular mark point, they will receive some marks for this, proportional to the number of criteria attained at the higher level). However, the mark will be capped below the grade point for which the submission does not attain all expectations. Therefore, it is crucial that your team understands all marking criteria, and gets the basics right before trying to attain higher level expectations!

Band I: 0–20%

I.1.1 *Delivery/necessary* (1 marks): The submission is structured as required. All files and directories, including the top-level directory, meet the required naming conventions.

FAQ: What criteria must be met in order to attain a team mark in the 0–20% range (band I) in the major group project?

Hint: To meet this requirement, make sure you read the specification of the deliverables carefully. Prevent errors by maintaining the project structure using the required specifications from the very start. Leave sufficient time to double or triple check the submission before uploading.

I.1.2 *Delivery/necessary* (2 marks): All required content is included in the submission as specified in the checklist of deliverables[4].
Hint: Keep track of the list of deliverable and what you have completed in a shared document.

[4] The checklist of deliverables will be provided via KEATS under the Major Group Project topic.

I.2.1 *Design/critical* (1 marks): The source code includes functional data structures or domain models that represent core entities and support application logic.

I.2.2 *Design/critical* (1 marks): The source code includes functional components or handlers that respond to user interactions or requests.

I.2.3 *Design/critical* (1 marks): The source code includes presentation-layer templates or components that render dynamic content based on application data.

I.2.4 *Design/necessary* (1 marks): The submission includes a diagram modelling the structure of the system that has been produced. A UML structural diagram (a class diagram, a component diagram, or a deployment diagram) is expected. Where the team is unable or unwilling to employ UML, the diagram should be accompanied by a legend describing the components.

I.4.1 *Functionality/critical* (1 marks): The application delivers some form of content to the end user. This content may be static content or views includes static content or views that are accessible without any dynamic data processing.

I.4.2 *Functionality/critical* (2 marks): The application includes dynamic interfaces or views that display data retrieved from a persistent storage layer (e.g. a database or a file) or retrieved from an online data source.

I.4.3 *Functionality/critical* (3 marks): The application includes a number of usable features corresponding to at least the equivalent to two one-person-week user stories per effective team member.
Hint: Use the full development period. This will ensure that this requirement is met.

I.6.1 *Version Control/critical* (1 marks): The project's git repository is accessible via Team Feedback.
Hint: Make the shared repository accessible as soon as possible.

I.6.2 *Version Control/critical* (1 marks): The repository shows a history of mostly small commits throughout development.
Hint: Expect team members to make regular commits when working on tasks and push these immediately to the shared repository. Use this information to assess when team members are actively working on the project. If you enforce this expectation, the repository will naturally meet this requirement.

I.6.3 *Version Control/necessary* (2 marks): Commits include informative messages that adhere to the requirements for commit messages set out in 5.5.4 of the Module Handbook.
Hint: Review commit messages regularly, or when merging work at the very latest,

I.8.1 */necessary* (1 marks): Team Feedback contains a record of meetings (at least one per week while the team is working on the project). Each meeting is recorded on Team Feedback within less than one week after the meeting took place. Each meeting record includes an accurate attendance list.
Hint: Assign a team member the responsibility to take meeting minutes. Take attendance and record a first draft of the minutes during the minutes rather than postpone this until a later date.

I.8.2 */necessary* (2 marks): Each meeting record meets the expectations set out in Section 5.5.2, including an agenda, a clear outline of key decisions made at the meeting, and log of actions arising from the meeting (other than software development task allocations assigned via a project management tool, such as Trello).

FAQ: What criteria must be met in order to attain a team mark in the 20–40% range (band II) in the major group project?

Band II: 20–40% The criteria of this band are assessed *only if all critical* requirements of band I are met in full.

II.1.1 *Delivery/necessary* (1 marks): The submission includes automated setup scripts or configuration files to initialise the development environment and install dependencies. Unless asked otherwise, you will be using Nix.
Hint: Further information on using Nix or related tools will be released in January 2026.

II.1.2 *Delivery/necessary* (1 marks): There are predefined user access credentials specified in the `README.md` file that the examiners can access the application.
Hint: Do not implement two-factor authentication or CAPTCHA unless required to do so by a client in a client-proposed project. Such features make accessing your application more time-consuming for markers and should not be necessary for a demonstration application.

II.1.3 *Delivery/necessary* (1 marks): The submission supports database setup and seeding/unseeding via automation commands.

II.1.4 *Delivery/necessary* (1 marks): The submission supports running tests and generating a coverage report via automation commands. The submission includes the original coverage report(s) as produced by the team's chosen code coverage tools.
Hint: The availability and easy of use of automated testing and code coverage tools must be a consideration in the decision as to what technology stack to use in the project. Do not commit a technology stack unless this is resolved.

II.1.5 *Delivery/necessary* (1 marks): The submission supports running the application via automation commands.

II.2 *Design/critical* (3 marks): The design diagrams are free from major errors and reflect the application developed.

II.3 *Code (Source)/critical* (3 marks): The code is free from significant defects or major bugs.
Hint: Test your code regularly and before each merge.

II.4.1 *Functionality/critical* (1 marks): The application allows users to input and submit new data through appropriate user interfaces or interaction mechanisms.

II.4.2 *Functionality/critical* (1 marks): The application provides functionality to retrieve and display existing data to users.

II.4.3 *Functionality/critical* (1 marks): The application allows users to modify existing records or content through appropriate interfaces.

II.4.4 *Functionality/critical* (1 marks): The application includes functionality for users to remove or deactivate data as needed.

II.4.5 *Functionality/critical* (1 marks): The application integrates data management features into cohesive workflows that support user goals.

II.4.6 *Functionality/critical* (1 marks): The scope and scale of the application is adequate for the team's effective size.

II.5 *Testing/critical* (3 marks): There are at least two automated tests per effective team member.
Hint: This criterion should be easy to meet if the team produce tests alongside source code.

FAQ: What criteria must be met in order to attain a team mark in the 40–60% range (band III) in the major group project?

Band III: 40–60% The criteria of this band are assessed *only if all critical* requirements of bands I and II are met in full.

III.1.1 *Delivery/necessary* (1 marks): The submission includes clear reused software references or AI generated code in README.md.

III.1.2 *Delivery/necessary* (1 marks): The application can be installed and run without manual fixes beyond documented steps.

III.2 *Design/critical* (3 marks): The software design separates concerns through the use of appropriate modules, such as class-based components or well-structured functions. Where a framework is used, the framework structural conventions are broadly adhered to. Where no framework is used, the team introduced a basic, sensible organisation to software.

III.3.1 *Code (Source)/critical* (2 marks): The code mostly follows consistent naming conventions.

III.3.2 *Code (Source)/critical* (2 marks): The code contains no egregious code smells that an average team should be able to pick up via superficial reviews (e.g., very deeply nested code, very long functions, egregious examples of code repetition).

III.4.1 *Functionality/critical* (2 marks): The application supports multiple related features integrated into a cohesive workflow.

III.4.2 *Functionality/critical* (2 marks): Features are fully functional and tested manually (no broken paths).

III.4.3 *Functionality/critical* (2 marks): The application handles basic error cases gracefully (e.g., invalid input).

III.5.1 *Testing/critical* (2 marks): The project includes unit tests for critical features.

III.5.2 *Testing/critical* (2 marks): The test suite achieves coverage greater than 50%.
Hint: Use a code coverage tool.

III.8 *Project management/necessary* (1 marks): The team maintains updated task tracking (e.g., backlog, progress board). This is made available either through a Trello board connected to Team Feedback, or other evidence that needs to be submitted in the `appendixes.zip` file, so that examiners have access.

Band IV: 60–80% The criteria of this band are assessed *only if all critical* requirements of bands I, II, and III are met in full.

FAQ: What criteria must be met in order to attain a team mark in the 60–80% range (band IV) in the major group project?

IV.1 *Delivery/critical* (0 marks): All necessarycriteria from bands I, II, and III were met (precondition for Band V eligibility).

IV.2.1 *Design/critical* (1 marks): Code adheres to separation of concerns and avoids bloated or multi-purpose handlers.

IV.2.2 *Design/critical* (2 marks): Shared logic or rules (e.g., access control, validation) are abstracted and reused across components to avoid duplication.

IV.3.1 *Code (Source)/critical* (1 marks): Variables, functions, and classes have clear, descriptive names.

IV.3.2 *Code (Source)/critical* (1 marks): Consistent code layout throughout the project.

IV.3.3 *Code (Source)/critical* (1 marks): No function/method exceeds 25 lines. No function/method has more than 2 levels of nesting.

IV.3.5 *Code (Source)/critical* (1 marks): The code base is mostly DRY (significant repetition avoided).

IV.3.6 *Code (Templates)/critical* (1 marks): Templates are structured to minimise repetition through modular and reusable components. (e.g.: Common layout elements such as headers, footers, navigation are abstracted into separate files or components).

IV.3.7 *Code (Templates)/critical* (1 marks): Presentation is separated from structure and logic (e.g.: no inline styling, CSS classes used instead; design systems are used).

IV.4.1 *Functionality/critical* (2 marks): The application supports an ambitious range of objectives.

IV.4.2 *Functionality/critical* (1 marks): The feature set is cohesive, avoiding isolated features that don't contribute to objectives.

IV.4.3 *Functionality/critical* (1 marks): Features are fully developed and offer an intuitive, polished interface. Expedient implementation at the expense of usability is generally avoided. For example, dates/times follow UK conventions and are intuitive. Records can be identified without exposing internal IDs (e.g., primary keys). Lists include pagination, ordering, and searching.

IV.4.4 *Functionality/critical* (1 marks): Features provide flexibility for end users (e.g., multiple ways to interact, customisation, admin parameters, preferences etc.).

IV.5.1 *Testing/critical* (1 marks): The project includes a comprehensive test suite.

IV.5.2 *Testing/critical* (2 marks): The test suite achieves impeccable statement and branch coverage.

IV.5.4 *Testing/critical* (1 marks): All tests pass and there is evidence of manual tests.

IV.7 *Management/critical* (2 marks): The UI is consistent across screens (look and feel). Terminology and language are consistent throughout the interface. Equivalent controls appear in the same place with the same look and feel.

Band V: 80–100% The criteria of this band are assessed *only if all* requirements (critical*and* necessary) of bands I, II, III, and IV are met in full.

FAQ: What criteria must be met in order to attain a team mark in the 80–100% range (band V) in the major group project?

V.2.1 *Design/critical* (1 marks): The design achieves high cohesion across components.

V.2.2 *Design/critical* (1 marks): The design achieves low coupling across components.

V.2.3 *Design/critical* (1 marks): Classes have limited responsibility (ideally single responsibility). Functions and methods do one thing only.

V.3.1 *Code (Source)/critical* (1 marks): Naming is consistent and meaningful throughout the application.

V.3.2 *Code (Source)/critical* (1 marks): Functions and methods are extremely short (15 lines or less). Functions and methods have no more than 1 level of nesting.

V.3.4 *Code (Source)/critical* (1 marks): The code includes no repetition (fully DRY).

V.3.5 *Code (Test)/critical* (1 marks): Test code uses clear, descriptive, and consistent names.

V.3.6 *Code (Test)/critical* (1 marks): Test code repetition is minimal.

V.3.7 *Code (Test)/critical* (1 marks): Test function/method bodies contain 25 lines or less with no more than 2 levels of nesting.

V.4.1 *Functionality/critical* (3 marks): The application is very ambitious in scope for the team's effective size.

V.4.2 *Functionality/critical* (3 marks): The application is exceptionally polished in terms of usability and completeness.

V.5.1 *Testing/critical* (2 marks): Test suites are carefully designed to cover a comprehensive range of input and output partitions.

V.5.2 *Testing/critical* (1 marks): Test suites address a wide range of potential error causes.

V.7 *Management/critical* (2 marks): The team's time management was excellent throughout the project. The final three days of the project were free from significant development activity (based on commit stats).

8.6.2 *Major mark correction*

FAQ: Under what conditions would my individual major group project mark be a severe reduction on the team's mark?

The major (individual) mark correction scheme of the major group is similar to that of the small group project, with some small differences. As for the small group project, there is a correction scheme based on contribution statistics derived from GitHub and a correction scheme based on meeting attendance. The major individual mark correction (if there is any), is the highest correction produced by both schemes.

Due to the potential variety of projects in the major group project, it can be difficult to assess whether team members contributed every week to the project. Team members should ensure their work is shared and tracked via a remote repository hosted by GitHub. As this is not always consistently done, *no major mark correction based on contribution concerns is applied unless there is at least one peer assessment that raises significant concerns*. If one or more peer assessments about a team member are very poor, the contributions of that team member recorded on the registered team repositories are reviewed. If they fall short of the expectation of weekly contributions throughout the team's project period, a correction is applied. Table 8.1 summarises the mark scheme based on contributions.

The major individual mark correction scheme for attendance is summarised in Table 8.2. It is identical to that of the small group project.

8.6.3 *Minor mark redistribution*

FAQ: If my individual mark is not severely reduced, do I receive the same mark as my team members in the major group project?

The minor mark redistribution scheme for the major group project operates in exactly the same manner as that of the small group project. This scheme is explained on page 7.8.4.

8.6.4 *Peer assessments*

FAQ: How are the peer assessments marked in the small group project?

The peer assessments you write about your team mates are marked directly, focussing on the open-ended feedback text that you write. This contributes towards 5% of the small group project mark. The peer assessment marking scheme in the major group project is similar to that of the small group project, except that the scheme is slightly stricter. The following assessment criteria are used to produce this mark:

0% The feedback text contains no meaningful write-up.

20% A peer assessment was submitted for almost every team member. The feedback text of each submitted peer assessment contains at least a single sentence explaining the scores for that peer assessment.

40% A peer assessment was submitted for every team member. Each peer assessment includes at least a 3–4 sentence explanation for the peer assessment scores.

60% A peer assessment was submitted for every team member. Each peer assessment includes at least a 3–4 sentence explanation for the peer assessment scores, recognising strengths and weaknesses, as well as some valid constructive feedback that, when acted on, would help the reviewee in future projects. The text is largely free from grammatical errors.

80% A peer assessment was submitted for every team member. Each peer assessment includes a thorough explanation for the peer assessment scores, recognising strengths and weaknesses, as well as valid constructive feedback that, when acted on, would help the reviewee in future projects. The peer assessment is written thoughtfully, and internally cohesive. The constructive feedback logically follows from the explanation. Some ideas are rooted in project management theory. Some are insightful. The text is free from grammatical errors.

100% An unusually extensive peer assessment was submitted for every team member. Each peer assessment includes a thorough explanation for the peer assessment scores, recognising strengths and weaknesses, as well as valid constructive feedback that, when acted on, would help the reviewee in future projects. The peer assessment is written thoughtfully, and internally cohesive. The constructive feedback logically follows from the explanation. The ideas are firmly rooted in project management theory and insightful throughout. The text is free from grammatical errors.

In some teams, you may have a team member who failed to engage with the team at all: i.e. they did not attend any meetings or produce any work. Such a person may remain in your team if they signed up to participate in the small group project and did not defer early on. Obviously, there is little to write about in such a situation. The feedback to write for such a peer can be limited to a single sentence explaining the lack of engagement without affecting your peer assessment mark. If a person's engagement was very limited, but they attended some meetings or produced some work, focus on your interactions with them.

Table 8.1: Major individual mark correction for contribution (code and report) in teams with a development period of n weeks. Weeks are defined as the Monday-Sunday intervals between the start and end of the project.

#Weeks of minimum contribution	#Weeks of used minimum contribution	Mark correction
$n-2$ or more	3 or more	no effect
	2	-10%
	1	-20%
	0	-40%
$n-3$	4 or more	no effect
	3	-10%
	2	-20%
	1	-30%
	0	-50%
$n-4$	4 or more	-10%
	3	-20%
	2	-30%
	1	-40%
	0	-60%
$n-5$	4 or more	-20%
	3	-30%
	2	-40%
	1	-50%
	0	-70%
$n-6$	4	-30%
	3	-40%
	2	-50%
	1	-60%
	0	-80%
$n-7$	3	-50%
	2	-60%
	1	-70%
	0	-90%
$n-8$	2	-70%
	1	-80%
	0	-90%
$n-9$	1	-80%
	0	-95%
0	0	-100%

Table 8.2: Major individual mark correction for meeting attendance.

% of meetings where team member is "present" or "5 minutes late"	Mark correction
80% or above	no effect
60% – 79%	-20%
40% – 59%	-50%
20% – 39%	-80%
less than 20%	-100%

A
Team charter checklist

FAQ: What issues should be covered when drafting a team charter?

Each person tends to have their own habits, work practices, and preferences. These differences between people can cause challenges and tension within the team. To avoid this, it is good practice to agree a set of rules of engagement or work practices in the form of a team charter. To develop a team charter, the team must agree on a common set of practices and rules of engagement, as well as approaches to deal with deviations from the agreement, disagreements, and conflicts. These are then written down in a document that can be referred to if concerns, disagreements, or conflicts arise. By anticipating potential challenges and agreeing approaches to deal with these, such challenges can be dealt with more easily.

1. Communication
    - Which tools should be used for day-to-day communication? (e.g. Slack, WhatsApp, Teams, email, etc.)
    - What is the expected response time for messages?
    - How will be handle urgent v non-urgent messages?
    - We are required to organise regular team meetings (same time and place each week)
        - How many time slots will we reserve per week? (e.g. one, *two*, *more*)
        - How will we meet? (e.g. *in-person*, Teams, Zoom, etc.)
        - Which time slots will be used every week?
        - Who is responsible for convening/scheduling extra meetings when we need them?
2. Roles and responsibilities
    - How will we assign roles? (e.g. project manager, team meeting chairperson, team meeting secretary/minute taker, client/stakeholder liaison, product owner, etc.)
    - Will roles be fixed or rotated?
    - Who is responsible for deadlines, progress tracking, and reporting?
    - (if relevant) Who is responsible for communication/engagement with the client?

- What are team member responsibilities in the days before the deadline to ensure that the deliverables are submitted on time?

3. Collaboration tools
   - Which GitHub service will we use? (GitHub.com or GitHub.kcl.ac.uk)
   - Where will we store and share documents? (e.g. GitHub Wiki, OneDrive, Google Drive, etc.)
   - What project management tool will we use? (e.g. Trello, Jira, Notion, etc.)
4. Decision making
   - How will we make decisions? (e.g. majority vote, *consensus*, role based)
   - How will we resolve disagreements?
   - Who has final say if there is a deadlock?
5. Code development practices
   - What coding standards and guides will we follow, and to what extent? (e.g. specify a subset of the code guidelines in the module handbook)
   - How often should code be committed/pushed to version control? (e.g. at least once per day of code, push after every commit)
   - What process do we use to review a branch before merging with the main?
   - How will we ensure code quality? (e.g. pair programming, code review before merging, weekly/fortnightly code inspections)
   - How will we track tasks? (e.g. Kanban board, meeting minutes, etc.)
   - What is the expected level of automated testing?
6. Accountability
   - How will we track individual contributions? (e.g. GitHub, Team Feedback)
   - How will we deal with missed deadlines?
   - How will we deal with work that fails to meet agreed standards?
   - How will we deal with poor participation/engagement?
   - Under what circumstances will the team reassign a task?
   - What do we do with task assignment and incomplete tasks if a team member fails to attend a team meeting?

- If someone cannot contribute due to circumstances beyond their control, they should submit a **mcf!** (**mcf!**). How will you deal with such circumstances internally?
- How do we ensure that collaborative coding sessions are recorded correctly by the committer?

7. Conflict resolution
   - How will we raise concerns with the group?
   - What steps will we take if there is a disagreement in the team?
   - Under what circumstances should issues be escalated (e.g. to an advisor or the lecturer)?
8. Availability and commitments
   - What are the time constraints of each member?
   - How many hours per week should each member expect to commit to the project?
   - When should team members NOT be contacted? (e.g. weekdays 5pm-9am, weekends)
   - How will we handle reading week and deadlines for other coursework?
9. Team culture and values
   - How will we ensure respect and inclusivity in discussion?
   - How will we celebrate milestones and achievements?
   - What behaviours are unacceptable? (e.g. ignoring messages, missing meetings without notice, etc.)
   - How will the team respect and accommodate personal identities and needs? What issues affect this particular team? (e.g. pronouns, culturally sensitive issues, disabilities, etc.)
10. Review: When and how will the team accommodate the need for adjustments and changes to the charter?

B
Code inspection checklist

The following checklist provides a set of practical guidelines for writing clean, maintainable code in your software engineering group project. These principles are summarised from Robert C. Martin's *Clean Code* textbook [Martin, 2009]. You can use this checklist in your team's code inspections. Alternatively, select a subset of these principles for code inspections and leave a broader range for code reviews.

FAQ: Against what criteria should code cleanliness be assessed?

Naming

- Use meaningful and descriptive names for variables, functions, classes, and files. Names must make meaningful distinctions.
- Avoid abbreviations unless they are widely understood (e.g., `id`', '`url`').
- Use consistent naming conventions (e.g.: `snake_case` for variables and functions and `PascalCase` for classes).
- Choose pronounceable names to ease verbal communication within the team.
- Avoid misleading names that suggest incorrect behaviour or data type.

Functions

- Functions should be small (ideally 5 – 15 lines).
- Functions should do one thing, and do it well.
- Function names should clearly state their purpose and side effects.
- Avoid side effects unless they are intentional and clearly documented.
- Avoid deeply nested functions.
- Prefer fewer function parameters; avoid more than 3 where possible.
- Use default arguments or object parameters when appropriate.

Code structure

- Organise code into logical, cohesive modules.
- Do not repeat yourself. Write DRY code.
- Keep related functions and data close together.
- Use consistent indentation and formatting throughout the codebase.
- Limit the length of source files. Split large files into smaller ones when needed.
- Place higher-level concepts above lower-level details in source files.

Comments

- Document public classes, methods, and functions, ideally in a format suitable for automated documentation generation tools.
- Write comments only when the code cannot be made self-explanatory.
- Avoid redundant comments that restate what the code already expresses.
- Use comments in the body of the code to explain *why* something is done, not *what* is done.
- Keep comments up to date. Delete outdated or incorrect comments.
- Remove noise, such as "TODO" comments and commented out code.

Formatting

- Use consistent spacing, indentation, and bracket placement.
- Use blank lines to separate logically distinct sections of code.
- Group related code together and separate unrelated code.
- Keep line lengths reasonable (typically under 100 characters).

Error handling and control flow

- Use exceptions rather than error codes where possible.
- Avoid deeply nested code; return early when conditions are not met.
- Handle all expected error conditions gracefully and clearly.

Testing Considerations

- Write code that is easy to test (e.g., avoid global state).
- Design small, independent units that can be tested in isolation.

- Keep test code clean and readable, following the same standards as production code.

Practices

- Remove dead code and unused variables promptly.
- Refactor code continuously to improve clarity and structure.
- Use tools such as linters and formatters to enforce coding standards.
- Perform regular code reviews in addition to code inspections to maintain code quality.

Note: Clean code is not just about writing code that works. It is about writing code that can be easily understood, maintained, and extended by your team and future developers.

C
UNIX Command Line Quick Reference

Conventions. Words in UPPERCASE (e.g., `FILE`, `DIR`, `PID`, `USER`) are placeholders.

C.1 Files and Directories

- `pwd` – print the current working directory.
- `ls` – list files in the current directory.
- `ls -l` – list files in long format (with permissions, owner, size, date).
- `ls -a` – show hidden files as well.
- `cd DIR` – change to directory DIR.
- `cd ..` – go up one directory.
- `touch FILE` – create an empty file or update its timestamp.
- `cp SOURCE DEST` – copy file.
- `cp -r DIR1 DIR2` – copy a directory and its contents.
- `mv SOURCE DEST` – move or rename a file or directory.
- `rm FILE` – remove a file.
- `rm -r DIR` – remove a directory and its contents (destructive).
- `cat FILE` – display file contents.
- `less FILE` – view file contents one page at a time.

C.2 Managing Access Rights

- `ls -l` – shows file permissions in the first column (r=read, w=write, x=execute).
- `chmod MODE FILE` – change permissions. Example: `chmod 644 FILE` sets read/write for owner, read-only for others. Example: `chmod +x FILE` adds execute permission.

- `chown USER FILE` – change ownership of FILE to USER (requires privileges).
- `chgrp GROUP FILE` – change group ownership of FILE.

C.3 Managing Processes

- `ps` – list running processes for the current shell.
- `ps aux` – list all processes with details.
- `top` – interactive view of processes and resource usage.
- `kill PID` – terminate the process with process ID PID.
- `kill -9 PID` – forcefully terminate PID.

C.4 Searching with grep

- `grep PATTERN FILE` – search for PATTERN in FILE.
- `grep -i PATTERN FILE` – case-insensitive search.
- `grep -r PATTERN DIR` – search recursively in a directory.
- `grep -n PATTERN FILE` – show matching line numbers.
- `grep -v PATTERN FILE` – show lines that do not match.
- `grep -E "PAT1|PAT2" FILE` – search for multiple patterns using extended regex.

C.5 Redirection and Pipes

- `COMMAND > FILE` – redirect standard output to FILE (overwrite).
- `COMMAND » FILE` – append standard output to FILE.
- `COMMAND < FILE` – use FILE as standard input.
- `COMMAND 2> FILE` – redirect errors (stderr) to FILE.
- `COMMAND1 | COMMAND2` – send the output of COMMAND1 as input to COMMAND2 (pipe).
- `COMMAND1 | grep PATTERN` – filter output of COMMAND1 for lines matching PATTERN.
- `COMMAND1 | less` – view long output one page at a time.

These commands form the backbone of everyday UNIX usage. For details, see the manual pages with `man COMMAND`.

D
Git Command Quick Reference

Conventions. Words in UPPERCASE (e.g., `FILE`, `BRANCH`, `URL`) are placeholders.

D.1 Repository Setup

- `git init` – create a new repository in the current directory.
- `git clone URL` – clone a remote repository.

D.2 Inspecting State

- `git status` – show changed, staged, and untracked files.
- `git log` – show commit history.
- `git diff` – show unstaged changes.
- `git diff -staged` – show staged changes.

D.3 Staging and Committing

- `git add -A` – stage all modified and untracked files.
- `git commit -m "message"` – commit staged changes.

D.4 Branches

- `git branch` – list branches.
- `git branch BRANCH` – create a branch.
- `git checkout BRANCH` – switch to a branch.
- `git merge BRANCH` – merge into current branch.

D.5 Working with Remotes

- `git remote -v` – list remotes.
- `git pull` – fetch and merge remote updates.

- `git push` – upload local commits.
- `git push -u origin BRANCH` – push the local branch named BRANCH to the remote repository called `origin`, and set that remote branch as the default "upstream" for the local branch. This means that in the future you can simply run `git push` or `git pull` without specifying the remote or branch name, because Git will remember the association.

Bibliography

F. Brooks. *The mythical man-month: essays on Software Engineering*. Addison-Wesley, 1975.

T. DeMarco and T. Lister. *Peopleware. Productive Projects and Teams*. Addison-Wesley, third edition, 2013.

B. Hooker and R. Moir. *The waste detectives. Methods and techniques to improve flow, increase value and boost profitability in a large-scale transformation*. Independently published, first edition, 2022.

K. Kogon and S. Blakemore. *Project management for the unofficial project manager*. Franklin-Covey, updated edition, 2024.

R. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Addison-Wesley, 2009.

R. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design: A Craftsman's Guide to Software Structure and Design*. Addison-Wesley, 2017.

Frequently asked questions

Code contribution

- How is the code contribution data on Team Feedback used? (p. 32)
- Team Feedback is missing some of my code contribution data. Is that a problem? (p. 32)
- I suspect a member of my team has faked code contribution data. What can be done about this? (p. 34)
- How do I get credit for code produced through pair or mob programming and someone else made the commits? (p. 34)

Code quality

- In what ways is code a communication tool? (p. 62)
- Does clean code matter? (p. 67)
- Against what criteria should code cleanliness be assessed? (p. 133)

Code quality: Code structure

- Does it matter what file/package you put code in? (p. 71)
- Is code repetition acceptable? (p. 71)
- Does the order of functions/data matter? (p. 72)
- Does formatting matter, assuming the code compiles correctly? (p. 72)
- How long can a source code file be? (p. 72)
- How many entries can a directory contain? (p. 72)
- What order should functions, methods, and classes appear in within a source code file? (p. 73)

Code quality: Comments

- What kinds of comments should our code contain? (p. 73)
- Is there such a thing as excessive commenting? (p. 73)
- Can I use comments to explain how my code works? (p. 74)
- Can I use a comment to explain how a particularly difficult line of works? (p. 74)
- Can comments become outdated? (p. 74)
- Can we use TODO comments or comment out code? (p. 74)

Code quality: Error handling

- Which is better: throwing exceptions or returning error codes? (p. 76)
- Does nesting structures that perform error handling (e.g. try-except blocks) count towards nesting limits? (p. 77)
- Can we silence errors? (p. 77)

Code quality: Formatting

- What rules should we follow with regards to spacing, indentation, and bracket placement? (p. 75)
- How should we use blank lines in source code? (p. 75)
- How should units in a source code file be organised? (p. 76)
- Is there a maximum line length (p. 76)

Code quality: Functions

- How long can a function be? (p. 69)
- How much work can a function do? (p. 69)
- How should functions be named? (p. 69)
- Can functions have side effects? (p. 70)
- How much nesting in a function is ok? (p. 70)
- How many parameters can a function have? (p. 70)
- Is it good for a function to have default arguments? (p. 71)

Code quality: Naming

- What is a "good" name? (p. 67)
- Can we create our own abbreviations, so that our variable names are shorter? (p. 68)
- Do we need to coordinate the way we name things? (p. 68)
- Do I need to be able to say a name out loud? (p. 68)
- Can variable names be misleading? (p. 68)

Code quality: Testing

- The code I write is really hard to test with automated tests. Is that a problem? (p. 77)
- How can we make code more testable with automated unit tests? (p. 78)
- Does test code need to be clean? (p. 78)

Communication

- Do team members need to talk to each others? (p. 57)

Design

- What is software design? Where the handbook mentions software design, what are you talking about? (p. 79)
- What is the main issue we should think about when designing software? (p. 79)
- How do we decide what goes into a particular module and what does not? (p. 80)
- Should we design a solution that considers our future plans, or is it ok to keep our design as simple as possible? (p. 80)
- How should interactions between modules be organised? (p. 81)

Extenuating circumstances

- I have difficulties engaging with the project due to circumstances outside my control. What can I do? (p. 16)
- What happens if I need to redo a group project? (p. 17)

General questions

- What is the software engineering group project module? (p. 9)
- What are the aims and objectives of the Software Engineering Group Project module? (p. 11)
- What is the overall structure of this module? (p. 12)
- Can we use generative AI in this project? (p. 18)
- What is the expected workload for this module? (p. 19)
- What are the deadlines for this module? (p. 20)
- What should I be doing in the first week of teaching? (p. 39)
- What is the "independent study" component of the module? (p. 40)
- What is the roles of large and small group tutorials? Do they cover different material from the independent study component? (p. 41)
- What will I learn in the small group tutorials? (p. 42)
- What will I learn in the large group tutorials? (p. 42)
- What is expected of me in the group projects? (p. 43)
- How should give feedback to/write peer assessment for my teammmates? (p. 44)
- Does software quality matter? (p. 67)

Handbook

- Why does this module have a handbook? (p. 9)
- How should the module handbook be used? (p. 9)

Kanban board

- Do we have to register a Trello board on Team Feedback? (p. 35)

Leadership

- How do I get people in my team to do what they should be doing? (p. 48)
- How do I develop leadership or informal authority? (p. 49)
- What is the role of a project manager? (p. 50)

Major group project

- How are teams formed in the major group project? (p. 111)
- What choice of project assignment will we have in the major group project? (p. 112)
- I have an idea for a self-proposed project. Can I undertake this project instead of one proposed by staff or clients? (p. 112)
- What are client-proposed projects? (p. 113)
- Can I self-propose a project with an external client? (p. 114)
- What are academic-proposed projects? (p. 114)
- What languages, frameworks, and technologies are allowed in the major group project? (p. 114)
- How should the major group project be project managed? (p. 116)
- What are the deliverables for the major group project? (p. 117)
- What criteria must be met in order to attain a team mark in the 0–20% range (band I) in the major group project? (p. 119)
- What criteria must be met in order to attain a team mark in the 20–40% range (band II) in the major group project? (p. 121)
- What criteria must be met in order to attain a team mark in the 40–60% range (band III) in the major group project? (p. 122)
- What criteria must be met in order to attain a team mark in the 60–80% range (band IV) in the major group project? (p. 123)
- What criteria must be met in order to attain a team mark in the 80–100% range (band V) in the major group project? (p. 125)
- Under what conditions would my individual major group project mark be a severe reduction on the team's mark? (p. 126)
- If my individual mark is not severely reduced, do I receive the same mark as my team members in the major group project? (p. 126)

Marking

- What is the assessment structure for the module? (p. 13)
- How does my individual mark relate to what the team produces? (p. 14)
- How can I find out my mark and feedback on a group project? (p. 35)

Peer assessments

- How do the peer assessments affect me or my mark? (p. 13)
- How can I participate in the peer assessment exercise? (p. 35)

Project management

- Why is project management so important in this module? (p. 47)
- What makes project management difficult? (p. 47)
- What is the overall structure of a typical project? (p. 51)
- What are the main decisions to be made at the start of a project? (p. 51)
- What does project planning involve? (p. 52)
- We have a plan. What are the challenges in getting the plan executed? (p. 53)
- Our plan is not working. What do we do? (p. 55)
- What are the main decisions to be made at the end of a project? (p. 56)
- What is the role of communication in project management? (p. 57)
- How do you communicate project plans? (p. 60)
- How do we avoid wasting time and resources in our project? (p. 63)
- What issues should be covered when drafting a team charter? (p. 129)

Small group project

- How are students allocated to teams in the small group project? (p. 85)
- Why do you allocate students in the small group project based on student commitment, gender, and ethnicity? (p. 86)
- What knowledge and skills must team members have acquired before starting the small group project? (p. 87)
- Why are there small group project? (p. 88)
- What are the pros and cons of choosing the six week schedule? (p. 89)
- What are the pros and cons of choosing the four week schedule? (p. 91)

- What are the pros and cons of choosing the three/two week schedule? (p. 93)
- Will choosing a particular schedule affect my mark or how I will be assessed? (p. 94)
- Can we choose when the project starts and ends, provided the duration of the schedule is adhered to? (p. 95)
- What is the "normal" schedule? (p. 95)
- I am no longer able to meet the expectations of the schedule I have chosen. Can I change schedule or team? (p. 96)
- What languages, frameworks, and technologies are allowed in the small group project? (p. 96)
- Which development tools should be used in the small group project? (p. 97)
- How should the small group project be project managed? (p. 97)
- What are the deliverables for the small group project? (p. 98)
- What are the criteria used to mark the team in the small group project, and how does the team's effective size affect they way these criteria are interpreted? (p. 98)
- How is the team mark for the small group project decided? (p. 99)
- What criteria must be met in order to attain a team mark in the 0–20% range (band I) in the small group project? (p. 99)
- What criteria must be met in order to attain a team mark in the 20–40% range (band II) in the small group project? (p. 100)
- What criteria must be met in order to attain a team mark in the 40–60% range (band III) in the small group project? (p. 101)
- What criteria must be met in order to attain a team mark in the 60–80% range (band IV) in the small group project? (p. 102)
- What criteria must be met in order to attain a team mark in the 80–100% range (band V) in the small group project? (p. 104)
- Under what conditions would my individual small group project mark be a severe reduction on the team's mark? (p. 105)
- If my individual mark is not severely reduced, do I receive the same mark as my team members in the small group project? (p. 107)
- How are the peer assessments marked in the small group project? (p. 108)
- How are the peer assessments marked in the small group project? (p. 126)

Team feedback

- How can I get access to Team Feedback? (p. 28)
- Do I need to register my gender identity, ethnicity, and disability? (p. 28)
- Do we have to register shared GitHub repositories on Team Feedback? (p. 31)

Team meetings

- Do we have to schedule our meetings via Team Feedback? (p. 29)
- How can I record a team meeting on Team Feedback? (p. 29)
- Who must record the team's meeting on Team Feedback? (p. 29)
- When should a team meeting be recorded? (p. 29)
- Which meetings do we have to record via Team Feedback? (p. 30)
- Should a team meeting be in person, online, or hybrid? (p. 30)
- How important are Team Feedback meeting attendance records? (p. 30)
- Do meetings require an agenda? (p. 31)
- How important are Team Feedback meeting minutes? (p. 31)
- How do we organise team meetings? (p. 58)

Team member issues

- The number of participating people in my team is smaller than in other teams. How will that affect us? (p. 15)

Testing

- What is the role of software testing? (p. 81)
- What is the role of manual testing? (p. 81)
- What is the role of automated testing? (p. 82)
- Can we substitute automated testing by more extensive manual testing? (p. 82)
- Why should we evaluate test quality? (p. 82)
- How do code coverage tools help evaluate automated test suites? (p. 82)

- Is high code coverage enough to have a good automated test suite? (p. 83)
- Who should be writing automated tests? (p. 83)
- When should we write automated tests? (p. 84)
- What strategies can we adopt to ensure we produce good test suites? (p. 84)

Tools

- What is the role of software tools in this module? (p. 21)
- What software tools do I need in this module? (p. 21)
- Do we have to use/know how to use a UNIX command-line interface (CLI) or Terminal? (p. 22)
- What code editor or integrated development environment (IDE) should we use? Is using an integrated development environment (IDE) allowed? (p. 23)
- What version of Python and Django do I need (p. 24)
- What version control tools do we use in this module? (p. 24)
- Which version of GitHub should I use? (p. 24)
- What is Team Feedback? (p. 27)
- What tool should we use to produce the major group project report? Do we have to use LaTeX? (p. 36)
- Which productivity tools do we require? (p. 37)

Version control

- How do I configure git to assign the correct email address to my commits? (p. 25)
- How should the team be using version control tools? What is allowed and what is prohibited? (p. 26)
- In what ways are version control tools communication tools? (p. 61)

List of acronyms

CLI Command-Line Interface

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheets

DRY Don't Repeat Yourself

FAQ Frequently Asked Question

HTML Hypertext Markup Language

IDE Integrated Development Environment

KIP King's Inclusion Plan

MCF Mitigating Circumstances Form

SEG Software Engineering Group Project

UI User Interface

WYSIWYG What You See Is What You Get

Index