INFO1111 Exam Textbook Prep

Chapter 1

INFO1111 – 2025 S1
Week 01: Professionalism, Careers and Lifelong Learning

---

1 Professionalism and the Modern Computing Profession

1.1 The Real Currency of a Computing Career

Jose M. Aguilar captures the central theme of this unit:

> "If you think you are worth what you know, you are very wrong. Your knowledge today does not have much value beyond a couple of years. Your value is what you can learn and how easily you can adapt to the changes this profession brings so often."

In other words, professional value grows from adaptability, not from any fixed body of facts. Modern computing changes so rapidly that even the freshest graduate must expect today's technical toolkit to date quickly. Consequently, cultivating professional habits—curiosity, self-directed learning, critical reflection—matters as much as mastering any individual language, framework or device.

1.2 An Initial Puzzle: Reasoning About Constraints

To underline the importance of problem-solving over memorised knowledge, consider the classic **domino tiling puzzle** introduced in the lecture.

Problem statement
Can you cover an "almost-4 × 4" grid—and, by extension, an "almost-8 × 8" grid—with dominoes if one square is missing from each?

Key observations

1. A complete 8 × 8 chessboard contains 64 squares: 32 dark, 32 light.

2. Remove two opposite corners and only 62 squares remain, but both removed squares share the same colour.

3. Each domino must cover exactly one dark and one light square. Therefore, after removing two squares of the same colour the board contains an **imbalance**—32 squares of one colour but only 30 of the other.

4. Because every legal domino placement preserves colour balance (1 dark + 1 light), no sequence of dominoes can restore the lost parity. Hence the tiling is **impossible**.

The lesson: lasting professional skill lies in recognising underlying constraints—here, colour parity—rather than merely attempting exhaustive search or brute-force coding.

---

2 What Computing Graduates Actually Do

2.1 Typical Professional Activities

Although newcomers often equate "computing" with coding, day-to-day professional life is broad and interdisciplinary. Graduate roles routinely involve:

- requirements elicitation and business analysis

- system and data modelling

- iterative design and prototyping
- implementation (coding)
- unit, integration and system testing
- debugging and performance tuning
- deployment and support, including hardware or cloud infrastructure
- ongoing data analysis for insight or optimisation

Understanding how these activities interlock is crucial; employers seldom hire pure coders. Communication, collaboration and critical thinking are therefore as prized as technical fluency.

2.2 Time Spent Writing Code—A Reality Check

Survey data shared in class shows that graduates often devote **less than half** their working hours to writing code. Significant time is consumed by design meetings, documentation, user liaison, code reviews, and learning unfamiliar technologies. Preparing for this mix of responsibilities is one objective of INFO1111.

2.3 Interpreting Job Advertisements

A quick search on Australian sites such as Seek or CareerOne reveals terms like:

```
Role:          Full-Stack Java Developer
Skills:        Java, JavaScript, CSS, HTML5
               Hibernate, RESTful services
               SCRUM / Agile
Tools:         Eclipse, JIRA, GitHub
Extras:        Excellent English, 2 years experience,
               Bachelor of Computer Science
```

Reading such adverts teaches two things:

1. Employers assemble **tech stacks**—curated sets of interoperable technologies—to solve problems.
2. They also demand soft skills: teamwork, agility, and continuous learning.

2.4 Tech Stacks in Practice

A tech stack is an integrated palette of technologies used across a project or organisation. Classic example: **LAMP** (Linux, Apache, MySQL, PHP/Perl/Python). Modern variants appear on sites such as StackShare or Mixpanel and may include React, Node.js, Postgres, Docker, Kubernetes, Kafka and so on. Mastery of one stack demonstrates depth; the capacity to transfer principles across stacks demonstrates adaptability.

---

3 Navigating Technological Change

3.1 Roles That Disappeared and Emerged

Between 2000 and 2024 several roles faded—Fortran programmer, Flash developer, Windows XP administrator—while new ones flourished: user-experience designer, cloud developer, data miner, Internet-of-Things engineer. Lists will change again long before you graduate.

3.2 Landmarks of the Last Two Decades

Rapid innovation is the industry's norm. Consider the timeline:

- Facebook (2004), YouTube (2005), Google Maps (2005)
- Twitter (2006), Netflix streaming (2007)
- iPhone and 4G (2007–08)
- First Android phone (2008)
- Bitcoin (2009)
- iPad (2010), Apple Pay (2014), Apple Watch (2015)
- TikTok (2016), ChatGPT (2022)

Each product spawned new ecosystems, skills and businesses.

3.3 The Futility of Certain Predictions

History is littered with confident yet misplaced forecasts:

- "I think there is a world market for maybe five computers."
  —Thomas Watson, 1943
- "There is no reason anyone would want a computer in their home."
  —Ken Olsen, 1977
- "Spam will be a thing of the past in two years' time."
  —Bill Gates, 2004

The lesson is humility: anticipate change, but accept that its direction will surprise you. Designing a career for adaptability is safer than betting on any single technology.

---

4 Lifelong Learning as a Core Competency

4.1 Continuous Growth or Professional Stagnation

Bob Dylan's lyric—"not busy being born is busy dying"—reminds us that careers plateau when learning stops. INFO1111 therefore stresses *how to keep learning* after formal teaching ends.

4.2 Weekly Self-Learning Exercises

Each week you will investigate a concise "computing concept" independently, then discuss discoveries in the following lecture. The opening task is *Drunkard's Walk*—a random-walk model that illustrates unpredictable yet statistically describable motion. Later this week you also self-study **Markdown**, a lightweight document-formatting language heavily used for README files, issue tracking and academic writing. Limit yourself to two or three hours; the aim is to practise efficient, self-directed exploration, not to achieve mastery overnight.

4.3 Metacognitive Strategies

Effective autonomous learning involves:

1. Reading learning outcomes deliberately and mapping tasks to them.
2. Periodically self-checking progress ("Can I explain this concept to a peer?").
3. Managing time: beginning assignments early, pacing effort, submitting early where possible.
4. Building networks—peers, tutors, industry mentors—to exchange knowledge and receive feedback.

5 INFO1111: Structure and Expectations

5.1 Unit Goals

INFO1111 explores:

- what it means to work as a computing professional
- available specialisations and career pathways
- technical and non-technical skills required for success
- strategies for staying current, and common pitfalls in professional practice

5.2 A Mastery-Based Assessment Model

The unit uses *no numeric marks*. Instead, it offers six tasks grouped into three learning areas:

1. Knowledge – Foundation and Advanced
2. Skills – Foundation and Advanced
3. Self-Learning – Foundation and Advanced

Each submission receives one of three ratings: **Poor**, **OK**, **Strong**.
Process:

- You begin with the Foundation task in each area.
- Achieving **Strong** promotes you to the corresponding Advanced task.
- Multiple attempts are allowed; improvement, not perfection, is rewarded.

Final grades depend solely on the level you reach by semester's end:

- OK in Knowledge + OK in Skills = 50 (Pass)
- Progressing to Strong and then Advanced levels yields higher grades (Credit, Distinction, High Distinction).
- Late work incurs daily penalties that may reduce the final grade, so timely submission remains crucial even in a non-numerical scheme.

5.3 Practical Expectations

1. Complete each online lecture before Tuesday of its week and raise questions for the Wednesday review session.
2. Attend the tutorial allocated in your timetable (Thursday or Friday). Group work requires consistent presence.
3. Devote an additional 6–9 hours weekly for readings, self-learning and assessment preparation.
4. Monitor Canvas and Ed forums at least weekly for announcements, clarifications and peer discussion.
5. Communicate early with staff or group partners if difficulties arise; special consideration processes exist for genuine disruptions.

6 Summary and Forward Look

Professional success in computing rests on three pillars:

1. Deep yet portable technical know-how (e.g. coding, testing, architecture).
2. Non-technical competences (analysis, communication, teamwork).
3. A commitment to lifelong learning, enabling you to pivot as technologies, roles and industries evolve.

INFO1111 sets the stage by combining professional context, weekly self-learning challenges and a mastery-oriented assessment structure. Your immediate tasks are:

- Form tutorial groups and familiarise yourself with collaborative expectations.
- Begin the Markdown self-learning exercise, limiting yourself to a focused two-to-three-hour exploration.
- Reflect on your current learning strategies—time management, self-assessment, help-seeking—and note one improvement to implement this week.

Adaptability starts now.

Chapter 2

INFO1111 – 2025 S1
Week 02: Professionalism in Practice — Teams, Diversity and Command-Line Thinking

---

1 Professionalism Extends Beyond Code

Edsger Dijkstra famously argued that computer science is **"no more about computers than astronomy is about telescopes."** Felipe Jara updated the idea: **"Computer science is no more about coding than running a restaurant is about ingredients."**
Both remarks remind us that professional success depends on a constellation of capabilities—technical, organisational, and interpersonal—rather than on programming skills alone.

1.1 A Restaurant Analogy

Running a first-rate restaurant demands expertise in:

- Ingredient quality and supply chains
- Kitchen equipment and maintenance
- Recipes and cooking technique
- Service, ambience, finance, marketing and staffing

Analogous layers appear in software development:

Restaurant Layer	Computing Equivalent
Flour, rice, beef (technology)	Java, Python
Bread, sauce (frameworks)	Django, Angular
Oven, knife (tools)	Eclipse, JUnit, Git
Recipes (techniques)	Agile, DevOps
Cooking ability (skills)	Coding, teamwork, communication
Business skills (professionalism)	Ethics, processes, finance

Well-chosen ingredients are essential but never sufficient. Likewise, high-quality code matters, yet professionalism integrates ethics, business realities, and human factors into a coherent whole.

1.2 SFIA — A Structured View of Professional Skill

The **Skills Framework for the Information Age (SFIA)** catalogues the wide range of competencies demanded across the digital sector—from strategy and architecture through development, deployment, support and security. Familiarity with SFIA helps practitioners identify personal gaps and plan career growth systematically.

---

2 Working in Teams

Human-computer interfacing absorbs much attention in computing, but effective **human-human interfacing** is just as critical. Teams create nearly all non-trivial software.

2.1 What Counts as a Team?

Definitions highlight complementary skills and shared purpose:

1. "A group of people with a full set of complementary skills required to complete a task, job, or project." — BusinessDictionary

2. "An intact social system, complete with boundaries, interdependence for some shared purpose, and differentiated member roles." — Hackman et al.

3. "Individuals working together to achieve a goal… generating synergy." — Wikipedia

2.2 Types, Topologies and Roles

- **Team types**: development, operations, quality assurance, support, security.
- **Topologies**: stream-aligned, enabling, platform, …
- **Roles**: analyst, programmer, architect, tester, …

Detailed discussion of agile team structures appears in the recommended reading:
https://relevant.software/blog/what-agile-software-development-team-structure-looks-like/

2.3 Common Misconceptions

Research reviewed in class challenges several widespread beliefs:

- Harmonious teams do **not** always outperform those that argue; productive conflict can sharpen ideas.
- Leader style alone rarely explains team dynamics.
- Bigger teams do **not** automatically perform better.
- Teams that stay intact can **deteriorate**; 73 % of airline incidents studied occurred on crews' first day together.

2.4 Hackman's Model of Successful Teams

Attributes of consistently effective teams:
1. They satisfy internal and external clients.
2. They build capability for future work.
3. Members find meaning and satisfaction.

Five enabling factors:
- A real, bounded team
- Compelling direction (often framed as SMART goals)
- Enabling structure (right size, skill balance, clear internal process)

- Supportive context (information, rewards, development)
- Expert coaching (mentoring, timely feedback)

2.5 Teams in Professional IT Practice

Modern software work is:

- **Multi-disciplinary** — business analysts, creatives, engineers
- **Multi-faceted** — analysts, architects, coders, testers
- **Collaborative** — pair programming, code reviews, daily stand-ups

Approaches range from traditional plan-and-document to agile methods such as **SCRUM** and **Extreme Programming (XP)**.

2.6 Teams in the Classroom

Student teams differ from commercial ones:

- Limited consequences and partially shared fate
- Shorter schedules and mismatched personal goals

Evidence from past cohorts shows that **equal contribution, full discussion and mutual support** correlate with both higher product quality and member satisfaction. Conversely, logistics problems, task misallocation, poor coordination or free-loading depress performance and raise stress.

Practical strategies:

- Draft a **team constitution** at the first meeting—ground rules, communication channels, decision procedures.
- Surface cultural differences openly and agree on respectful norms.
- Address negative behaviours (aggression, blocking, freeloading) early.

2.7 Diversity and Bias

Diversity means "the inclusion of different types of people in a group or organisation" (Merriam-Webster). Every team is diverse on dimensions such as culture, language, gender identity, professional background or learning style. Awareness of **implicit bias**—automatic, unconscious associations—helps professionals build inclusive environments. Explore your own implicit associations at https://www.projectimplicit.net/ and watch the brief explainer at https://www.youtube.com/watch?v=dVp9Z5k0dEE.

---

3 Why Software Projects Are Hard

Industry data presented in class paint a sobering picture:

- Only ~30 % of software projects are deemed **successful** (deliver the right functionality on time and on budget).
- Roughly 50 % suffer significant overruns or feature cuts.
- About 20 % are abandoned altogether.
- In 2020 alone, **US $300 billion** in projects were scrapped and **US $750 billion** exceeded schedule or budget.

Comparable failure rates in civil engineering would be unthinkable. The complexity of intangible requirements, rapid change and human coordination challenges explain much of the difficulty—hence the premium on professionalism and robust team practice.

---

4 Tools of the Trade: Command-Line Interfaces and Scripts

4.1 The Software Stack in Perspective

```
Applications (Graphical Interfaces)
↑
Command-Line Interface (CLI)
↑
Operating System
↑
Hardware
```

The CLI bridges human intent and operating-system services without the overhead of a graphical layer. Mastery of command-line work is a hallmark of efficient developers and system administrators.

4.2 Shells You Should Recognise

- `bash` – default on most Linux and macOS systems
- `zsh` – feature-rich alternative, now default on new macOS installs
- `PowerShell` – object-oriented shell for Windows, also available cross-platform

4.3 Core Navigation and File Commands

Purpose	Unix/macOS (`bash`, `zsh`)	Windows (`PowerShell`/CMD)
Current path	`pwd`	`pwd` or `cd`
Change dir	`cd`	`cd`
Make dir	`mkdir`	`mkdir`
Remove dir	`rmdir`	`rmdir`
List files	`ls`	`dir` or `ls`
Display file	`cat`	`type` or `Get-Content`
Delete file	`rm`	`del` or `Remove-Item`

Learning these commands fluently accelerates virtually every development task.

4.4 Scripting for Automation

A **script** chains multiple CLI commands into a single executable file. Typical uses:

- Batch data conversion
- Automated testing and deployment (continuous integration)
- System maintenance (log rotation, backups)

Even short scripts can remove hours of repetitive manual effort and guarantee reproducibility.

---

5 Self-Learning Focus for Week 02: Big O Notation

Last week's self-learning exercise explored the **Drunkard's Walk**, a stochastic process illustrating random motion. This week you will investigate **Big O notation**—the mathematical language we use to describe how an algorithm's resource requirements grow with input size. Reserve two or three focused hours, then discuss your findings in the next tutorial.

---

6 Immediate Actions

1. **Form your project team on Canvas** (up to four members, five only with tutor approval).

2. **Attend Tutorial Week 02** prepared to:
   - Review assignment requirements
   - Practise group decision-making techniques
   - Exchange early insights on Big O notation

3. **Set up your development environment**: choose a shell (`bash`, `zsh` or `PowerShell`), verify basic commands, and write a first script that automates a small task you personally perform.

---

7 Key Takeaways

- Professionalism merges technical mastery with business insight, ethical awareness and strong communication.
- Effective teamwork is deliberate: clear goals, complementary skills, structured interaction and openness to conflict.
- Diversity enriches creative problem-solving; recognising bias is a professional responsibility.
- Command-line fluency and scripting remain foundational skills, independent of any particular modern framework.
- Big O notation is a core analytical tool—you will need it throughout your degree and career.

Advance confidently: professionalism grows by integrating these perspectives into everyday practice.

Chapter 3

INFO1111 – 2025 S1
Week 03: Communicating as a Computing Professional — From Email Etiquette to LaTeX Typesetting

---

1 Where We Stand: Essential Ideas from Week 02

By now you should recognise that technical expertise is only one strand in professional success.
Three Week 02 elements continue to surface throughout this chapter:

1. Professional identity — you decide whether you are the "kitchen helper" or the "restaurant owner".

2. Command-line fluency — a single pipeline such as

   ```
   cat names.txt | grep "David" | head -4 | sort
   ```

   reminds us that clear, compact commands save hours of manual effort.

3. Computational complexity — Big O notation (`O(1)`, `O(log n)`, `O(n)`, `O(n²)`) is the yard-stick for comparing algorithms.

Hold these ideas; they influence both the language you choose and the way you explain solutions to others.

---

2 Self-Learning Focus for Week 03: Regular Expressions

This week you will explore **regular expressions (RegExp)**—symbolic patterns that describe sets of strings. Unix tools such as `sed` depend on them.

Example 1 – Basic substitution

```
sed "s|aaaa|bbbb|g" file1 > file2
```

- searches `file1` for every occurrence of `aaaa`,
- replaces it with `bbbb`,
- writes the result into `file2`.

Example 2 – A denser pattern

```
sed "s|\(^T[io]m\)\W|\1my |g" names.txt > newnames.txt
```

Break-down:
- `^T[io]m` matches "Tim" or "Tom" only at the start of a line,
- `\W` requires a non-word character after the name,
- `\1` recalls the matched name,
- `my` inserts the string "my " immediately after it.

Try predicting the output, then verify on real data. Record questions for tutorial discussion; you will be asked to justify each part of the expression.

---

3 Why Communication Skills Matter in Computing

Code rarely speaks for itself. Designers, developers, analysts, testers and managers exchange **requirements, assumptions, constraints and decisions** on a daily basis. Misunderstandings cost time, goodwill and often money. Therefore, knowing *how* to communicate becomes as valuable as knowing *what* to communicate.

---

4 Foundations: What Is Communication?

Merriam-Webster defines communication as "a process by which information is exchanged between individuals through a common system of symbols, signs, or behaviour."
Key phrases to internalise:

- **information is exchanged** – not merely transmitted; the receiver must obtain meaning.
- **common system of symbols, signs or behaviour** – the parties must share at least some conventions. This is the domain of **semiotics**, the study of signs and symbols.

A quick puzzle from the lecture underscores the point:

```
Dpohsbuvmujpot. Bsf zpv uif gjstu up xpsl ju pvu?
```

Shift every character one place backward in the alphabet to reveal the hidden sentence. The exercise proves that, without the right symbol system, messages remain opaque.

---

5 Classical Models of Communication

5.1 Shannon's Transmission Model

Source → Encoder → *channel* → Decoder → Destination

Claude Shannon's model treats communication as the flow of encoded bits through a noisy channel. Think of a file transfer over TCP/IP or, in everyday terms, a team member sending JSON over a REST endpoint. The model highlights:

- noise — packet loss, line noise, typos, background distractions,
- redundancy and error detection — CRCs, parity bits, checksums.

5.2 Interaction Model

Sender Receiver

Both parties alternately encode and decode messages within a *context*. An agile daily stand-up is a textbook example: each speaker both conveys status and responds to colleagues in rapid cycles.

5.3 Transaction Model

Here communication is simultaneous and multi-directional. All participants are senders and receivers, messages overlap, and **feedback** is immediate. Pair programming at a single workstation embodies this model; verbal explanations, keystrokes and facial expressions converge in real time to solve a problem.

---

6 Effective Communication—Responsibility, Goals and Context

1. **Responsibility** — Effective communication is *joint*, yet the *sender* carries primary responsibility for making the message intelligible.

2. **Goal clarity** — Before you speak or write, decide what change you want:
   - recipients **know** something,
   - recipients **do** something,
   - recipients **feel** or **believe** something.

3. **Context awareness** — Tailor your approach to the recipients' starting point: existing knowledge, expectations, values, even preferred style. Use familiar reference points, avoid unexplained jargon and choose tone deliberately.

---

7 Choosing and Shaping the Form

7.1 Written Instructions vs Visual Map

Consider two ways to guide someone from the New Law Building to a destination across campus:

1. A numbered list of walking directions (61 m west, 120 m south, …).

2. A campus map with the route highlighted.

Both carry the same information, yet the map often communicates faster because it leverages *spatial cognition* and a different symbol system (graphics instead of prose).

7.2 Dimensions That Differentiate Communication Forms

- **Medium** — written document, oral talk, annotated code, live demo.
- **Scale** — one-page summary vs 50-page report; lightning talk vs half-day workshop.
- **Target** — boss, colleague, client, broad public.
- **Audience size** — one-to-one, small group, broadcast.
- **Interaction pattern** — continuous dialogue, one-way briefing, Q&A at the end.
- **Purpose** — convey facts, guide action, alter attitudes.

Skill lies in matching form to goal and context.

---

8 Everyday Professional Scenarios

8.1 Email to a Lecturer — Which Version Works Best?

Three sample messages appeared on the slide:

A.

```
Hiya,
I wanna get help bc my team wont work
IDK. ¯_( )_/¯
thx bigD
```

B.

```
Hi David,
I'm having problems with my group members who just aren't responding to any emails over the last few days.
Can you suggest how I should handle this?
Thanks, Dan
```

C.

```
Dear Professor Lowe,
Over the last week I have sent emails to my INFO1111 CC34-2 group members about the assignment on Fri, Sat
Kind regards, Daniel Smith (SID: 123456789)
```

Message B typically achieves the best balance: it is respectful, concise and requests advice rather than demanding specific punitive action. Message A fails on tone and clarity; Message C, while detailed, risks sounding accusatory and prescriptive.

8.2 Common IT Workplace Communications

- Designer specifies inputs/outputs for a new module.
- Consultant reports on business impact of emerging technology.
- Team leader presents project plan to client representatives.
- Developer requests funding for a tool and relevant training.
- Architect documents overall system design for future maintainers.
- Local expert teaches proper use of a tool to colleagues.

Each task differs in medium, audience, purpose and expected feedback.

---

9 Written Communication Tools: Introduction to LaTeX

9.1 What Are TeX and LaTeX?

- **TeX** — a low-level typesetting program created by Donald Knuth.
- **LaTeX** — macros built on TeX that provide higher-level structure (sections, figures, bibliographies).

Both operate on plain text with **markup tags** that describe rather than *display* structure; a compiler then produces PDF or other output.

Knuth's aims were:

1. allow anybody to produce high-quality books with minimal effort,
2. guarantee identical results on all machines at any time.

The development of TeX itself is a case study in disciplined software engineering: meticulous version control, public change logs and even monetary rewards (e.g. US $2.56, or 0x100 in hexadecimal) for bug reports.

9.2 Why Use LaTeX in INFO1111?

- Forces the writer to think about *content and structure* instead of superficial formatting.
- Excels at equations, cross-references and bibliographies.
- Illustrates computing concepts such as **compilation** and **separation of concerns**.
- Free, cross-platform and widely used in academia.

---

10 A First LaTeX Document—Line by Line

```
\documentclass[12pt]{article}
\usepackage{amsmath}          % add maths support

\title{First example}
```

```
\author{David}
\date{\today}

\begin{document}
\maketitle                    % generates the title block

\section{First section}
This is the \textbf{first section} in my \LaTeX\ document.
I can include maths directly in the text like $a^2 + b^2 = c^2$.
Or I can display it separately (see equation~\ref{aaa}).

\begin{equation}
\label{aaa}
\gamma^2 + \theta^2 = \omega^2
\end{equation}

That was so easy.
\end{document}
```

Observe:

- The *preamble* (`\documentclass`, `\usepackage`, metadata) separates concerns from the *content*.
- `$…$` denotes inline maths; `equation` provides a numbered, cross-referenced display equation.
- Commands begin with backslash; comments begin with `%`.

---

11 The LaTeX Toolchain

1. `pdflatex main.tex`
   - reads the source file, generates an auxiliary `.aux` file (cross-reference data) and a PDF.

2. If bibliography commands are present:
   - `bibtex main` uses the `.aux` file to build `.bbl` (formatted references).

3. Run `pdflatex` twice more so that citations, cross-references and table of contents resolve.

Diagrammatically (simplified):

```
main.tex      pdflatex      main.aux     bibtex     main.bbl

           second/third pdflatex passes
```

Running the compiler only once often leaves "??" in place of references—an error visible in the lecture's "Process (wrong)" slide.

---

12 Summary and Immediate Actions

- **Regular expressions** expand your command-line power; master them through experimentation with `sed`.
- Effective communication hinges on clearly defined goals, audience analysis and suitable choice of medium and style.
- Transmission, interaction and transaction models each illuminate different real-world scenarios—from network protocols to pair programming.
- LaTeX exemplifies *separation of content and presentation*; learning it now will accelerate every technical report you write during your degree.

Action checklist for the coming week:

1. Finish the RegExp self-learning exercise and bring at least two questions to the tutorial.

2. Draft a LaTeX document that contains one section, one numbered equation and one citation (use a `.bib` file). Compile it successfully with `pdflatex` and `bibtex`.

3. Review any current team issues; compose an email in the style of Message B if you need guidance from teaching staff.

Professionalism grows through precise thinking, deliberate communication and the effective use of tools—habitual practices that start now.

Chapter 4

INFO1111 – 2025 S1
Week 04: Technology Stacks, APIs and Automated Builds with Make

---

1 Where We Stand: Essential Ideas from Week 03

Week 03 left three practical reminders that underpin today's material:

1. **Receiver-centred communication**
   Always predict how the receiver will interpret your message. Context determines meaning; omission of context breeds error.

2. **LaTeX tooling on the command line**
   You should now be able to edit a `.tex` file and compile it directly with

   ```
   pdflatex main.tex
   ```

   Success proves you can drive a compiler from the shell—an ability you will reuse when Makefiles appear later in this chapter.

3. **Regular expressions as industrial-strength wildcards**
   The exemplar command

   ```
   sed "s|\(^T[io]m\)\W|\1my |g" names.txt > newnames.txt
   ```

   showed that a single pattern can filter, capture and rewrite text in one pass.
   For additional algorithmic context review the Big-O article:
   https://developerinsider.co/big-o-notation-explained-with-examples/

---

2 Self-Learning Focus for Week 04: Greedy Algorithms

A **greedy algorithm** makes the locally optimal choice at each step, hoping to find a global optimum.
Tasks for your self-learning log:

• Write a crisp definition in your own words.
• Locate (or create) one counter-example where greedy choice is *sub-optimal*—for instance, certain coin-change sets or the travelling-salesperson problem on asymmetric graphs.
Be prepared to defend why the greedy heuristic fails.

---

3 Technology Stacks

3.1 Definition

"A tech stack is the combination of technologies a company uses to build and run an application or project. Sometimes called a 'solutions stack,' a tech stack typically consists of programming languages, frameworks, a database,

front-end tools, back-end tools, and applications connected via APIs."
(Source: https://www.heap.io/topics/what-is-a-tech-stack)

In plain terms, a tech stack is the *inter-related kit* that turns ideas into running systems.

3.2 Layered Roles Inside a Stack

A workable mental model is to picture vertically-arranged layers, each fulfilling a role:

- **Operating system** – the foundation that schedules processes, manages file systems and exposes system calls.
- **Server layer** – may include
  – Data-management servers,
  – Application servers that hold back-end logic.
- **Back-end functionality** – business rules, authentication, analytics.
- **Front-end interfaces** – web, mobile or desktop GUIs that users touch.
- **Communications** – HTTP, WebSockets, gRPC or message queues carrying requests and events.
- **Load balancing and scaling** – distributes traffic and resources.
- **Monitoring & observability** – collects metrics, logs and traces so humans see inside the black box.

The reference chart on https://www.heap.io/topics/what-is-a-tech-stack illustrates this separation.

3.3 How the Layers Interact

Integration relies on:

1. **Configuration** – YAML, JSON or environment variables bind addresses, credentials and feature flags.
2. **Code** – libraries and frameworks expose calls in typed or dynamic languages.
3. **Network sockets** – every API request, database query or message-queue publish travels through TCP/UDP endpoints.

Skilled engineers understand *where* boundaries lie and *how* contracts are enforced across them.

3.4 Horizontal vs Vertical Aggregation in "Business" Stacks

The Chief Martec diagram of matrixed tech stacks (https://chiefmartec.com/2022/01/matrixed-tech-stacks-with-both-horizontal-and-vertical-aggregation-platforms/) shows that enterprises often curate:

- **Horizontal platforms** – analytics or identity layers reused by many products.
- **Vertical slices** – complete solutions tailored for marketing, finance or supply chain.

Recognising these patterns helps you align technical decisions with organisational strategy.

3.5 Tracking Real-World Stacks

Several public catalogues reveal what leading companies deploy:

- Heap's definition and examples – https://heap.io/topics/what-is-a-tech-stack
- FullScale "Top 5 Tech Stacks" – https://fullscale.io/blog/top-5-tech-stacks/
- Crowd-sourced StackShare – https://stackshare.io/
- Mixpanel case studies – https://mixpanel.com/blog/tech-stack-examples/

Study them to relate lecture concepts to production systems.

---

4 APIs — The Glue of Modern Software

4.1 Definition and Significance

API stands for **Application Programming Interface**. An API specifies *how* software components interact: endpoints, parameters, expected responses and error semantics.

4.2 Google Maps API as Working Example

The lecture used Google Maps (reference: https://www.w3schools.com/graphics/google_maps_reference.asp) to illustrate progressive integration diagrams.

1. **Single-tier view**

   ```
   Web Browser   Your code   Database Server   Maps
   ```

2. **Adding an explicit API server**

   ```
   Web Browser   Web Server   API Server   DB Code   Maps
   ```

3. **Full third-party involvement**

   ```
   Web Browser   Web Server   Google API   Google Maps
                   Your internal DBMS Code
   ```

Each arrow represents an HTTP or WebSocket call; each box can be scaled, monitored and versioned independently.

---

5 Automating Builds with Makefiles

5.1 Why Manual Builds Fail

Real applications are assembled from many source files, assets and generated artefacts. Dependencies form a directed graph; rebuilding everything every time wastes hours and invites mistakes. We need declarative automation.

5.2 Core Vocabulary

A **Makefile** (the default name recognised by `make`) lists:

- **Target** – a file you want to produce or refresh.
- **Dependencies** – files the target relies on.
- **Process (recipe)** – shell commands that create or update the target.

5.3 Minimal Working Example

```
test1: test2
    echo "Creating test1"
    touch test1

test2:
    echo "Creating test2"
    touch test2
```

`make test1` checks timestamps: if `test2` is newer (or absent) it executes the recipe for `test2` first, then for `test1`.

5.4 Building a LaTeX Report — Complete Makefile

```
filename = INFO1111_Group_Project_CC99-01
bibfile  = main.bib

pdf: $(filename).pdf
    echo
    echo "PDF files now up to date"

$(filename).pdf: $(filename).tex $(bibfile)
    pdflatex $(filename)
    bibtex   $(filename) || true
    pdflatex $(filename)
    pdflatex $(filename)
```

```
    echo
    echo "$(filename) has been updated"

$(filename).bib:
$(filename).tex:

clean:
    rm -f $(filename).{ps,pdf,log,aux,out,dvi,bbl,blg,toc}
```

Key points:

1. **Variables** (`filename`, `bibfile`) prevent duplication and ease refactor.
2. The `pdf` *meta-target* depends on the concrete PDF file, then prints a success message.
3. Multiple `pdflatex` passes resolve cross-references; `|| true` after `bibtex` suppresses non-fatal errors so the script continues.
4. `clean` provides a safety net—restore a pristine workspace with one command.

5.5 How `make` Decides What to Rebuild

`make` compares the timestamp of each target against its dependencies. If any dependency is newer (or missing), the associated recipe runs. Otherwise, the target is considered *up-to-date*, saving compute time and reducing human error.

---

6 Summary and Immediate Actions

1. Master the definition and layered nature of **tech stacks**; recognise roles from OS up to monitoring.
2. Visualise **API** interactions—start from the simple browser-to-code picture and progress to multi-service ecosystems.
3. Re-implement the lecture's Makefile in your own directory; modify `filename` to compile a personal `.tex` report.
4. Complete the **greedy algorithm** self-learning exercise: record a sub-optimal example and reason about its failure.

Proficiency in modern computing demands not only writing code but understanding the *infrastructure, interfaces and automation* that carry code into production. The tools introduced this week—stack thinking, API design and Makefile builds—equip you to assemble reliable, maintainable solutions on real-world projects.

Chapter 5

INFO1111 – 2025 S1
Week 05: Finding Information, Collaboration and Git

---

1 Linking Back to Week 04

Week 04 closed with two practical threads that flow straight into the present material:

1. **Makefiles** – you proved you could automate the creation of initial `.tex` and `.bib` files with

   ```
   make setup
   ```

   That success highlighted why reproducibility matters—an idea we will revisit through *version control*.

2. **Greedy algorithms** – an algorithmic reminder that "doing the best-looking thing now" is not always correct.

Keep those capabilities in mind as we expand to *information discovery* and *multi-person engineering workflows*.

---

2 Self-Learning Focus for Week 05: Race Conditions

A **race condition** occurs when the correctness of a computation depends on the relative timing of independent events.
Situations where it matters:

- Concurrent reads/writes to shared memory.
- Two build scripts trying to overwrite the same file.
- Network services that assume a specific order of messages.

Locate one real or hypothetical case where a race condition corrupts results, and record which synchronisation mechanism (lock, atomic operation, transactional memory) would remove the hazard.

---

3 Finding Information

3.1 Sources and Resources

- **Sources** are the materials from which ideas and data are drawn.
- **Resources** are the access paths to those materials.

Immediately available resources include:

– The University of Sydney Library portal
– Main catalogue: https://library.sydney.edu.au/
– Research guide: https://library.sydney.edu.au/research/index.html?accordion=findinfo
– Databases index: https://library.sydney.edu.au/databases/
– Google Scholar: https://scholar.google.com.au/
– ACM Digital Library: https://dl.acm.org/
– Any other reputable online repository.

3.2 Judging Quality: The REVIEW Checklist

Use the six REVIEW criteria each time you decide whether to trust a document.

R — Relevance
- Does the content address your report's specific question?

E — Expertise of Author
1. What are the author's qualifications?
2. Are they writing within their recognised field?
3. Do other authors cite them?

V — Viewpoint of Author or Organisation
1. Could professional or commercial affiliations bias the work?
2. Was the piece sponsored by a party that benefits from its message?
3. Is the goal to inform, persuade, or entertain?

I — Intended Audience
- General public, practitioners, or scholars?

E — Evidence
1. Are claims backed by primary data, secondary citation, expert opinion, or merely anecdote?
2. Are references accurate?
3. Has the article been peer-reviewed?

W — When Published
1. Publication date.
2. Have major developments occurred since then?

A quick example: the Quora answer declaring that *"a data scientist should learn git"* may be relevant and timely but offers limited peer-reviewed evidence—treat it as an informed opinion, not definitive authority.

3.3 Referencing and Academic Integrity

Referencing acknowledges prior work and lets readers verify your claims. According to the University Library guide,

"Referencing is an essential part of academic writing … to point the reader to the original documents so that they can determine independently whether the attributed sources support the author's argument as written."

Three key principles follow:

1. **Cite direct quotations, paraphrases, and ideas**, even if they come from your own earlier writing.

2. **Never** submit another person's work, recycle your own without citation, fabricate data, or assist others in misconduct.

3. **Separate your contribution from the literature** with clear language and consistent citation style.

---

4 Collaboration

4.1 What Collaboration Means

Dictionaries converge on a simple definition: *"to work jointly with others, especially in an intellectual endeavour."*
For students collaboration surfaces in group projects, shared attribution, and efficient use of limited time.
For professionals the stakes rise—security, cost of failure, audit trails, and legal accountability demand **traceability** of every change.

4.2 Industry Expectations

Modern workplaces default to shared online systems, for example:

- Task boards such as Asana.
- Team discussion forums.
- Central or distributed **version-controlled repositories** that guarantee

– Security of artifacts,
– Full history for accountability,
– Conflict detection when edits collide.

Version control is therefore not optional; it is the backbone of safe collaboration.

---

5 Version Control Fundamentals

5.1 Why Version Control?

Without a formal system, teams resort to filenames like

```
Report_final_v7_reallyFinal_NEWest.docx
```

Consequences include:

- Inability to pinpoint when (or by whom) a bug entered the code.
- Manual, error-prone rollback.
- Frequent clashes when parallel edits touch the same lines.

Effective version control supplies:

- A strict history of every change.
- Branches to isolate experimental work.
- Automated merging and conflict flags.

5.2 Historical Milestones

1970 s Conceptual origins
1982 RCS
1986 CVS
2004 Subversion (SVN)
2005 Git | Mercurial

Front-ends such as Bitbucket and GitHub popularised these tools. Google Trends confirms Git's dominance.

---

6 Git: The Dominant Distributed Version Control System

6.1 Core Model

1. **Snapshot database** – each commit stores what the project looked like at one point in time.
2. **Working files** – where you edit.
3. **Staging area** – where you queue changes with `git add`.
4. **Repository (local)** – receives commits.
5. **Branches** – independent lines of history that can later merge.
6. **Remote repositories** – additional copies on servers, enabling push and pull between machines.

A simplified life-cycle:

```
edit → git add → git commit → git push
                                  ↑
                        git pull (others' work)
```

6.2 Remote Storage Providers

While Git needs no central server, most teams use one:

• GitHub • Bitbucket • GitLab • AWS Commit • Azure DevOps
INFO1111 projects live on https://github.sydney.edu.au

6.3 Canonical Workflow: Step-by-Step

The lecture's HTML example illustrates every major command.

```
# 1  Start a repository and record the first file
git init
edit paper.html
git add paper.html
git commit -m "First paper"

# 2  Evolve the document
edit paper.html        # add red styling
git add paper.html
git commit -m "Added styling"

# 3  Inspect differences
```

```
git diff
edit paper.html        # multi-colour paragraphs
git add paper.html
git commit -m "Multicolour styling"

# 4  Branch from the earlier commit to try a CSS file
git checkout -b exper <commitB>
edit paper.css
git add paper.*
git commit -m "Separate CSS"

# 5  Merge the experiment back into main
git checkout master
git merge exper        # resolve any conflicts
edit paper.html        # manual fixes
git add paper.html
git commit -m "Merged version"
git branch -d exper

# 6  Connect to a remote and push everything
git remote add origin https://github.sydney.edu.au/INFO1111-2025/GitDemo.git
git push -u origin --all
```

Key observations:

- `git status` and `git log` keep you informed at every step.
- Conflicts surface during `git merge`; they are resolved by editing the marked sections, staging, and committing.
- `git push -u origin --all` establishes tracking so future `git push` / `git pull` suffice.

6.4 Visualising Operations

The Wikimedia diagram (Daniel Kinzler, *Git operations.svg*) maps each command onto arrows:
commit branch merge push/pull. Keep that mental picture handy; it explains the flow of history.

---

7 Project Management Context

Effective collaboration extends beyond code storage.

- **Task allocation and transparency** – To-Do lists, boards, and issue trackers.
- **Methodologies** – Lean, Agile, Scrum, Kanban, Waterfall: each prescribes how tasks move from backlog to done.
- **Kanban Boards on GitHub** – a practical illustration of workflow embedded directly inside the repository host.

---

8 Immediate Practice

1. Clone your lab repository from GitHub Sydney to your laptop, make a trivial edit, commit, and push.
2. Invite a peer to pull, branch, and propose a conflicting change; resolve it together.
3. Evaluate one article this week using the REVIEW checklist—note gaps in evidence or currency.
4. Draft an annotated bibliography entry in your preferred citation style and confirm that every quotation and paraphrase is fully attributed.

---

9 Summary

• Information literacy begins with **identifying reputable sources**, assessing them via the REVIEW criteria, and **referencing** scrupulously.
• **Collaboration** demands transparency, traceability, and accountability—properties delivered by **version control**.
• **Git** models project history as snapshots, supports branching and merging, and syncs work across remotes such as GitHub.
Master these foundations and you will move comfortably from solo exercises to large, multi-developer systems—precisely what the professional world expects.

Chapter 6

INFO1111 – 2025 S1
Week 06 A: Systems Thinking and Problem Solving

---

1 Linking Back to Week 05

Race conditions closed last week. A single shared bank account illustrated the hazard:

```
Initial balance           200
Person 1 reads balance    200
Person 2 reads balance    200
Person 1 adds 100         300  (writes)
Person 2 adds 200         400  (writes)
Final balance             400  (should be 500)
```

A €5 µs timing difference was enough to lose $100, and similar problems surface in hardware as well as software. This vignette foreshadows two Week 06 themes:

1. **Systems behave in ways we cannot predict by looking at one component in isolation.**

2. **Effective problem solvers must first understand interactions before they propose fixes.**

The self-learning thread for the coming week is **Hill Climbing**, a search heuristic you will explore on your own.

---

2 A Mind-set for Professionals

Albert Einstein reputedly allocated "55 minutes to understanding the problem and 5 minutes to solutions."
George E. P. Box warned that "the only way to know how a complex system will behave after you modify it, is to modify it and see …"
Together they capture the professional stance: think deeply about the *system* that hosts your solution and test ruthlessly.

---

3 Systems: When Simple Parts Create Complex Wholes

3.1 Emergent Patterns

• *Turing patterns* arise when two simple chemical reactions have different diffusion rates, producing leopard spots and seashell stripes.
• Reaction–diffusion equations, cellular automata, flocking birds and ant colonies all confirm that **simple rules can yield intricate, beautiful, and sometimes chaotic global behaviour**.

(Recommended visual: concentric waves of colour generated by a two-variable reaction–diffusion simulation.)

3.2 Key Questions for Software Engineers

1. Can we predict whole-system behaviour from the behaviour of its parts?
2. If the parts are extremely simple, does prediction become easier or harder?
3. Can we design a set of simple interacting components that guarantee the behaviour we want?

---

4 Systems Thinking

4.1 Definition

At its broadest, **systems thinking** connects a proposed solution, the technical system that implements it, and the society in which that system operates.
More narrowly it studies each component *and its relationships* to see how they influence the functioning of the whole.

4.2 Why It Differs from Traditional Analysis

Traditional analysis decomposes a problem into parts and studies each part in isolation. Systems thinking keeps decomposition but adds explicit focus on:

- The *structure* that links parts.
- Feedback loops—balancing or reinforcing.
- Delays that blur cause and effect.
- Emergent behaviour that no engineer consciously designed.

4.3 Complexity and Dynamics

Systems are usually nonlinear, history-dependent and dynamic.
A design that works today may destabilise tomorrow as feedback loops amplify small changes.

---

5 Consequences of Ignoring the System

5.1 The Tragedy of Borneo's Roofs

DDT spraying killed malaria-carrying mosquitoes—and also gecko predators, which allowed caterpillars to devour thatch roofs.
The *intended* improvement (public health) produced an *unintended* side-effect (structural damage) because engineers treated mosquitoes as an isolated problem.

5.2 Therac-25: A Software Race Condition Turned Lethal

The Therac-25 medical linear accelerator suffered an unsynchronised interface/control task. Operators who keyed commands "too quickly" triggered a race, delivering fatal radiation overdoses.
System lesson: **the user interface, scheduler, safety interlocks and human workflow formed a single interacting system**. Testing individual modules did not expose the failure mode.

5.3 General Principles

- A perceived isolated issue may actually be a node in an unseen network of causes.
- Long feedback delays hide causal chains until the damage is already done.
- What worsens a situation can sometimes be leveraged to create change once the loop is recognised.

---

6 Problem-Solving Fundamentals

6.1 First Understand the Real Problem

Elevator complaints capture the point. Engineers considered faster motors or extra shafts—massive costs—until someone installed full-length mirrors near the doors. Guests occupied themselves, complaints vanished, and **the real problem proved to be boredom, not elevator speed**.

6.2 Puzzles as Micro-laboratories

Matchstick equations, donkey re-arrangements and "FIND THE NEXT NUMBER" puzzles sharpen skills:

- The *surface* question looks numeric, geometric or logical.
- The *hidden* question tests your ability to reframe the problem (e.g., count letters rather than digits).

A methodical solver:

1. Lists all given constraints.
2. Checks whether any constraint is implicit or assumed.
3. Experiments with low-cost trials to expose the underlying structure.

6.3 Requirements in Professional Contexts

Poor requirements come in many forms:

incorrect • incomplete • vague • conflicting • missing • design-dictating

Compare:

- "The system shall have good usability" (vague).
- "The system shall respond very fast" (vague).
- "The system shall respond in less than 2 s" (measurable).
- "The system shall be error-free" (unrealistic).

Capturing clear, verifiable requirements is itself a problem-solving task.

---

7 Approaches and Techniques

7.1 Divergent Thinking

Tools such as *Six Thinking Hats*, mind-mapping, word association, role-playing, "what if" exaggeration and SWOT analysis generate many options quickly.

Exercise: In 30 s list as many new uses for 50 paper clips as you can. Quantity breeds originality.

7.2 Convergent Thinking

Decomposition—"divide and conquer"—breaks a large problem into smaller, loosely coupled, highly cohesive components. Choices include:

- Functional decomposition (what the system does).
- Procedural decomposition (sequence of steps).
- Domain decomposition (business sub-areas).
- Role decomposition (responsible stakeholders).

The aim is *minimal coupling* and *maximal cohesion*.

7.3 Levels and Languages

Architects sketch systems at the **architecture level**; programmers express solutions at the **algorithm/unit level**. Stay fluent at both scales—you must translate strategic goals into tactical code and back again.

---

8 The Market Value of Problem Solving

Average Australian salaries (payscale.com, glassdoor.com.au):

- Software Tester $62 k
- Programmer $69 k
- Software Engineer $75 k
- Business Analyst $82 k
- IT Project Manager $103 – 128 k

The more your role centres on defining and solving the *right* problems, the more organisations will pay.

---

9 Week 06 Self-Learning Focus: Hill Climbing

Hill Climbing is an iterative optimisation strategy.
Starting from an initial solution you repeatedly move to a neighbour with a better score until no neighbour scores higher. The method is:

1. Simple to implement.
2. Effective on smooth landscapes.
3. Prone to get stuck in local maxima.

You will explore its strengths and weaknesses in the independent exercise.

---

10 Summary

- Complex behaviour emerges from simple interacting parts; therefore prediction requires a *system* view, not just component tests.
- Systems thinking foregrounds relationships, feedback and time delays; ignoring them leads to expensive or dangerous mistakes.
- Problem solving begins with correct problem definition, continues with creative divergence, and finishes with structured convergence.
- Clear requirements, low coupling, high cohesion and multi-level fluency translate these ideas into robust software.

Master these habits now and you will write code—and design organisations—that thrive in the face of complexity.

Chapter 7

INFO1111 – 2025 S1
Week 06 B: Advanced Systems Thinking and Complexity

---

1 From Simple Task Lists to Whole-System Structure

1.1 Work Breakdown Structures (WBS)

In Week 05 you learned how decomposition breaks a project into manageable pieces.
A Work Breakdown Structure formalises that idea:

Definition A WBS is a deliverable-oriented grouping of all work required for a project. It defines total scope and underpins schedule, cost, resource and change management.

Creating a WBS is itself an act of *systems* thinking. You decide where boundaries lie, which deliverables interact and which do not. Four practical approaches exist:

1. Analogy Begin with the WBS of a similar project and tailor it.

2. Top-down Start with the final product and subdivide.

3. Bottom-up List specific tasks first, then roll them up.

4. Mind-mapping Capture ideas non-linearly, then re-organise into a hierarchy.

A sample web-development WBS might group work by product features (GUI, database, authentication), functional layers (presentation, logic, data) or any other organising principle that reflects the system you are building.

1.2 Why a Purely Task-Centred View Is Not Enough

A WBS clarifies *what* must be done, but not *how* the resulting artefacts will interact once delivered. For that we need **design architecture**—a description of all entities in a system and their relationships—and the strategic insight to create and capture value in *large, complex, evolving* designs.

---

2 Key Concepts for Designing Complex Systems

2.1 Design Architecture and Strategy

Carliss Y. Baldwin frames architecture as a deliberate partitioning of the whole into *modules*:

- Within a module → high internal dependency.
- Across modules → near independence.

The architect's job is to locate interfaces so that:

1. Work can be assigned efficiently.

2. Future change concentrates inside modules, not across them.

3. Stakeholders can appropriate the value they create.

2.2 The Design Precedence Matrix

A *precedence matrix* (or dependence structure matrix, DSM) records which component must be designed before which. Analysing the matrix reveals feedback loops, parallelisable clusters and bottlenecks—insights impossible to obtain from a flat task list.

Stuart Kauffman's NK-landscape metaphor re-interprets that matrix as a rugged fitness surface. Each dependency (K) couples design variables; more coupling raises landscape ruggedness and makes optimisation harder.

---

3 Principles of Systems Thinking

Systems scientists distil several recurring ideas:

1. Openness Every system exchanges matter, energy or information with its environment. You must draw a *system boundary* that distinguishes controllable from uncontrollable variables.

2. Purposefulness Actors behave in line with goals and values; understanding *why* they act is as important as *how* they act.

3. Adaptiveness Systems respond, learn and evolve.

4. Emergent Properties The whole exhibits behaviours that no part possesses in isolation (e.g., traffic jams, market crashes).

5. Multidimensionality Economic, technical, social and political dimensions coexist and may conflict.

6. Counter-intuitiveness Well-intentioned actions can trigger the opposite of their intended effect.

7. Inflection Points Small parameter shifts can flip the entire regime of behaviour.

These principles guide both diagnosis and design. They warn you to examine feedback loops, time delays and unintended consequences before committing to code, policy or hardware.

---

4 Dynamics, Patterns and Unintended Consequences

4.1 The Pesticide Paradox

Daniel Aronson's agricultural example is classic. Applying pesticide kills crop-eating insects (*short-term benefit*), yet over time removes food for predators, letting the pest population rebound even higher. Interventions that ignore ecological feedback simply move the problem in time.

4.2 Predator–Prey Simulation

A software demo of the Lotka–Volterra equations or a cellular-automaton predator–prey world shows cyclic dynamics: predator numbers lag prey numbers, sometimes collapsing both. Such simulators make *interdependence* visible and measurable.

4.3 Pattern Formation in Cellular Automata

Bar-Yam's classroom exercise begins with binary grids such as

```
1 0 1 1 0 0 0 1 …
```

and applies a simple local rule iteratively. Different initial seeds produce stripes, checkerboards or chaotic speckles, illustrating that:

- There are 2 possible initial structures.
- Local interaction rules alone can generate global order—or disorder.

4.4 A Model of Panic

Replace "0" with "calm" and "1" with "panicky". A person changes state if a threshold number of neighbours are panicky. Repeating the rule across an auditorium shows how panic can cascade or die out depending on initial clustering. *Emergence* and *threshold effects* are no longer abstract—they are counted row by row:

```
Step 0   0s:526   1s:498
Step 1   0s:557   1s:467
…
Step 8   0s:614   1s:410
```

The same rule, applied to a different initial seating pattern, can leave most people calm. Structure matters.

---

5 Modularity as a Weapon Against Complexity

5.1 Definition

Modularity is the degree to which a design is partitioned into components that are *highly dependent internally and nearly independent externally.*

5.2 IBM System/360 – Proof by Industrial Scale

Between 1962 and 1967 IBM's System/360 became the first fully modular computer family. Standardised Solid Logic Technology (SLT) circuits, common I/O control structures and interchangeable processors let IBM ship a *product line* rather than separate models. Market response validated the **option value** of modularity: customers could start small and upgrade pieces later.

Yet even IBM hit limits: operating systems and compilers stubbornly resisted modularisation. The lesson is two-fold:

1. Hardware can often be modularised earlier than system software.

2. When software lacks clear interfaces, vendors may struggle to remain the sole supplier—an early hint of the open-source movement that would follow.

A partial list of the 35 System/360 modules—processors, memories, tapes, performance tools, FORTRAN/COBOL compilers—shows how work was allocated across geographically dispersed teams.

5.3 Measuring Modularity with DSM Tools

Coordination Cost (sum of dependency weights) and Change Cost (percentage of components affected by a typical change) quantify architectural health. A comparative study found:

- Mozilla (post open-source launch) Coordination Cost 30.5 million, Change Cost 17.4 %
- Linux of similar size Coordination Cost 15.8 million, Change Cost 6.7 %

Lower numbers signal cleaner modular separation, reduced ripple effects and easier evolution.

---

6 Tenets of the Systems Thinking Approach

The canonical list consolidates what you have seen in examples and principles:

- Interdependence Isolated elements never form a system.
- Holism Analyse parts, but evaluate wholes.
- Goal Seeking Interactions should converge toward a purpose.
- Inputs / Outputs Closed systems have fixed inputs; open systems accept new stimuli.
- Transformation Systems *do* something—convert inputs into outputs.
- Entropy Without energy or information inflows, order decays.
- Regulation Feedback keeps behaviour within acceptable bounds.
- Hierarchy Systems nest within systems; leverage this to manage complexity.
- Differentiation Sub-systems specialise.
- Equifinality Different paths, same outcome.
- Multifinality Same starting point, different outcomes.

You will meet each tenet repeatedly—in operating systems scheduling, network protocols, business process engineering and organisational design.

---

7 Connecting the Dots

By now you should see why Week 06 spans two chapters.
Part A highlighted *why* you must think in systems; Part B equips you with the *conceptual tools*:

1. WBS decomposition establishes a clear map of deliverables but must be complemented by architectural views.

2. Design precedences, DSMs and NK-landscapes expose interaction patterns invisible in task lists.

3. Core principles—openness, emergence, multidimensionality—explain paradoxes such as stronger pesticides increasing pest damage.

4. Simulation (predator–prey, panic) turns qualitative notions into quantitative insight.

5. Modularity, measured not preached, remains the most practical antidote to coordination overload.

---

8 Self-Test Questions

1. Explain how *counter-intuitiveness* and *threshold effects* combine in the auditorium panic model.

2. A legacy codebase has a coordination cost twice that of a peer project. What architectural interventions could halve it?

3. Sketch a WBS for developing a predator–prey mobile app, then identify which WBS elements belong in separate modules.

4. Describe equifinality in the context of achieving fault tolerance in distributed systems.

Reflect on these before moving to Week 07; they demand the same holistic reasoning the INFO1111 exam will require.

---

9 Summary

Complexity is not an aberration—it is the default state of modern IT systems. Professional engineers:

- Identify and respect system boundaries.
- Seek modular architectures but verify them with metrics.
- Use simulation and DSM analysis to uncover hidden dependencies.
- Recognise that interventions reverberate through ecological, social and technical layers.

Master these habits and you will not merely *build* systems; you will *shape* them to evolve gracefully in a world that never stands still.

Chapter 8

INFO1111 – 2025 S1
Week 07 A: Intellectual Property and Professional Ethics

> "The most important thing about intellectual property vs. creative expression is that copyright law was created not to stifle creativity but to encourage creativity."
> — Shepard Fairey

> "Intellectual property has the shelf life of a banana."
> — Bill Gates

1 Why This Chapter Matters

Week 06 taught you to reason about complex technical systems.
Week 07 shifts the lens from *how systems work* to *how society expects professionals to behave* when they create, exploit or deploy those systems. Two strands dominate:

1. Intellectual Property (IP) – the rules that determine *who owns* ideas expressed in software and allied artefacts, and *how that ownership can be shared or enforced*.

2. Professional Ethics – the principles and frameworks that help you decide *what is right or wrong* when technical possibilities, business incentives and human values collide.

A clear grasp of both strands is essential for any open-ended INFO1111 exam question that asks you to justify decisions, evaluate trade-offs or propose responsible action.

2 Intellectual Property

2.1 What Counts as Intellectual Property?

The Free On-line Dictionary of Computing defines IP as
"the ownership of ideas and control over the tangible or virtual representation of those ideas. Use of another person's intellectual property may or may not involve royalty payments or permission, but should always include proper credit to the source."

Ownership gives two powers:
- the right to *benefit* from the idea (for example, earn licence fees), and
- the right to *decide* how others may use it.

2.2 A Brief History

14 Century Letters *patent* (publicly issued documents) granted specific rights.
1624 Statute of Monopolies curbed abuse of royal patents and preserved protection only for genuinely *novel* inventions.
1662 Licensing of the Press Act gave a printers' guild control over publication—an early form of content censorship.
1710 Statute of Anne (Copyright Act) transferred authority from printers to *authors* for 14 years (renewable once), explicitly "for the Encouragement of Learning."

The modern message is unchanged: society grants time-limited monopolies so that people will invest effort in new work.

2.3 Who Owns the Code You Write?

Ownership is often *context specific*:

- Portrait paintings The person who commissions the work is the first owner, although the artist can sometimes limit other uses.
- Commissioned photographs The photographer normally owns copyright, unless the photos are "private or domestic" in which case the commissioner does.
- Films Copyright sits with the *maker*—the person who organised the production—yet live performers might also qualify as makers.
- Commissioned films Ownership shifts to the commissioning party.

Software, under Australian law, is classed as a **literary work**.
- If created *in the course of employment*, the employer owns it.
- Otherwise the individual developer(s) do, unless a separate agreement says differently.

Think about assignments: what does the University's student agreement say about code you submit?

2.4 Four Legal Mechanisms for Protecting Ideas

1. **Trade Secrets**
   - You hide crucial information and make "reasonable efforts" (confidentiality policies, physical or electronic security).
   - Only executables, not source code, are released.
   - Protection lasts as long as secrecy lasts, but it collapses if someone independently discovers the idea.
2. **Copyright**
   - Arises automatically.
   - In the US and Canada a work must be *fixed in a tangible medium*; France and Australia do not require fixation.
   - Exclusive rights: reproduce, publish, adapt and communicate the software.
   - Notable exceptions: fair dealing for research, criticism, review, parody, satire; flexible dealing in teaching; classroom demonstrations; back-up copies.

Application Programming Interfaces (APIs) test the boundary.
Could you write a new mapping system but expose exactly the same interface as the Google Maps API? Courts continue to debate the answer (e.g. *Oracle America v. Google*, multibillion-dollar stakes).

3. **Patents**
   - Protect a *technical solution* that is new, inventive and useful.
   - Algorithms in the abstract are not patentable, yet an algorithm embodied in a device may qualify.
   - The inventor gains a time-limited monopoly but must publish enough detail for IP Australia to make the patent public.
4. **Trademarks**
   - Safeguard distinctive brands—words, logos, sounds, even smells.
   - Must be actively used or can lapse.
   - Registration is optional but strengthens protection.
   - A company name or domain name is *not* automatically a trademark and vice-versa.

2.5 Licensing: Turning Ownership into Value

Once ownership is clear you can grant permission via a licence.

- Proprietary software End-User Licence Agreements (EULAs) stress that the product is "licensed, not sold."
- Free and Open Source Software (FOSS) Still copyrighted but the licence may allow users to copy, modify, distribute and even sub-licence, subject to conditions.

Licences allocate rights *and* liabilities; choose them deliberately.

2.6 Commercialisation in Practice

2.6.1 A Classroom Scenario You invent a phone app that analyses vibrations to diagnose car suspension problems. A fellow student builds a prototype for an honours project; a freelancer turns it into a polished product; the app goes viral; the NRMA offers $12 million.
Questions:
- What should you do?
- How should the money be divided between inventor, student, supervisor and freelancer?

The answer depends on who owned which IP at each stage and what agreements, if any, were in place.

2.6.2 Why the Stakes Are High Market capitalisations in 2021: Apple $2 200 B, Microsoft $1 930 B, Amazon $1 680 B, etc.
IP is a major—sometimes the only—asset for technology companies.

2.6.3 Funding and Start-up Vocabulary Angel investor, seed funding, incubator, accelerator, due diligence, NDA, technology transfer, IPO, R&D tax incentive—learn these terms; they frame every commercial discussion.

2.6.4 Business Model Patterns Digital enterprises commonly adopt one (or combine several) of the following models:

Brokerage Marketplace, auction, transaction broker, virtual marketplace
Advertising Portal, classifieds, advertising networks, audience measurement
Merchant Virtual, click-and-mortar, bit vendor
Manufacturer, direct sales, affiliate, pay-per-click, revenue-sharing
Community Open content, social networking
Subscription User registration, metered usage, metered subscriptions
Infomediary, content services, networking, utility, incentive marketing

Examples: Amazon (virtual marketplace), eBay (auction), PayPal (transaction broker), Wikipedia (open content), NetFlix (content services). Even McDonald's earns much of its profit as a *real-estate* business rather than a food business.

2.7 Managing IP in Distributed Development

• Proprietary/commercial code may rely on copyright and obfuscation.
• Shareware, freeware and open-source models distinguish **free as in beer** (zero price) from **free as in speech** (few constraints).
• Licences such as Creative Commons, BSD, GNU and others specify what users may distribute, modify or sub-licence.

Before releasing code, consider Bill Gates' famous 1976 "Open Letter to Hobbyists" (software deserves payment) and the counter-movement that created the modern open-source ecosystem.

---

3 Professional Ethics

> "We will always have to temper significant advances with ethics."
> — Ben Carson

3.1 From Personal Morals to Social Rules

• *Morals* are your internal compass.
• *Ethics* are externally accepted rules of conduct.
Telling the truth is a moral ideal; disclosing confidential medical data may still be unethical if it violates professional duties.

3.2 Conceptual Frameworks for Ethical Reasoning

Ethical pluralism acknowledges that no single theory covers every situation. Three broad families appear in computing practice.

1. **Teleological (Consequentialist) Ethics**
   – Evaluate the *state of the world* that results.
   – *Egoism* values consequences for the actor.
   – *Utilitarianism* seeks "the greatest good for the greatest number," balancing intense harm to a few against mild harm to many.
2. **Deontological Ethics (Duty-based)**
   – Judge actions by whether they honour duties rather than by their outcomes.
   – W.D. Ross listed seven prima facie duties: fidelity, reparation, gratitude, justice, beneficence, non-maleficence and self-improvement. Duties often clash—ethical reasoning weighs their relative importance.
3. **Contractarianism**
   – Right actions are those people would accept under a fair social contract.

3.3 Professional Codes of Conduct

• **Australian Computer Society (ACS)**
1. Primacy of public interest
2. Enhancement of quality of life
3. Honesty
4. Competence
5. Professional development
6. Professionalism

• **ACM / IEEE-CS (2018 revision)**
Selected imperatives:
– Contribute to society and human well-being
– Avoid harm
– Be honest and trustworthy
– Be fair and avoid discrimination
– Respect intellectual property

– Respect privacy and confidentiality
– Work only within areas of competence
– Design secure, robust systems
– Put the public good first, especially for infrastructure systems

Codes supply guidance, priorities and sometimes disciplinary procedures.

3.4 Ethics in Corporate Strategy

Corporate Social Responsibility (CSR) extends "profit, people, planet" with a fourth bottom line: *progress, purpose or principles*. Your own decisions—where to work, what to buy—signal how much ethical stance matters. Would you join a tobacco firm if the salary doubled?

---

4 Case Studies and Scenarios

The following real or hypothetical situations test both IP law and ethical frameworks. They are fertile ground for exam questions that ask you to *analyse*, not merely *describe*.

1. **Googalzon Consultants** – You can finish a $6 M mining-software project in two months for $1 M. The boss tells you to conceal the shortcut to keep staff employed and revenue flowing. Utilitarianism might justify the deception if jobs and cash benefit many; deontology emphasises honesty and client fidelity.
2. **Facebook Mood Experiment** – By tweaking users' newsfeeds, researchers provoked emotional contagion. Issues: consent, harm, beneficence, data-driven manipulation.
3. **Toysmart Bankruptcy** – An online toy retailer promised never to share customer data, then tried to sell the database in bankruptcy proceedings. The FTC intervened; Disney eventually destroyed the data. Raises questions of contractarian trust.
4. **Google WarDriving** – Street View cars logged Wi-Fi SSIDs, MAC addresses and unencrypted packets. Google blamed a rogue engineer, but documentation showed managerial approval. Are publicly broadcast packets "free to capture"?
5. **High-School IT Administrator** – The principal asks you to break into a teacher's computer suspected of containing pornography. Under what policy or legal authority can you comply?
6. **Cleaning-Company Phone Tracking** – Management wants to install a location-tracking app covertly. Consider employee consent and proportionality.
7. **Kim Dotcom and MegaUpload** – Cloud storage or conduit for mass copyright infringement? What responsibility do developers have for illegal user behaviour?
8. **Start-up Analyses Proprietary Code** – A contractor reverse-engineers a competitor's algorithm. Is that misappropriation?
9. **Garry McKinnon Hack** – Entering US military networks in search of UFO evidence: curiosity, whistle-blowing or crime?
10. **Left-Pad Chain Accident** – A bug in your open-source padding function propagates through libraries and kills motorists when a traffic-control system crashes. Concepts: negligence, strict liability, professional malpractice.
11. **Therac-25 Radiation Overdoses** – Six massive overdoses, three deaths. Contributing factors: no independent code review, poor failure-mode analysis, cryptic error codes. Who is liable—the programmers, the company, the regulators?
12. **Volkswagen Diesel Emissions** – Software activated emission controls only in test labs. CEO claimed "a couple of software engineers." Intentional wrongdoing versus systemic corporate pressure.
13. **Facebook Group Outing & Web Accessibility** – Automatic group additions disclosed sexual orientation; a blind user sued Coles for an inaccessible web site. Both illustrate inadvertent harm and discrimination through technical design.

5 Liability and Competence

Legal theories invoked when software fails or causes harm:

- Criminal offences
- Intentional torts
- Strict liability
- Negligence
- Fraud or negligent misrepresentation
- Deceptive trade practice
- Breach of contract
- Malpractice—measured against a *professional* standard, not just a hobbyist's.

The ACM code reminds you to "acquire and maintain professional competence" and "accept appropriate professional review." Before you declare yourself "a good programmer," ask: Who judges? By what benchmark? Links in the slides point to self-assessment resources.

---

6 Self-Learning Spotlight: The Halting Problem

Week 07's self-learning exercise introduces Alan Turing's **Halting Problem**: determining, for arbitrary code and input, whether execution will eventually stop. It is *undecidable*—no general algorithm exists. The result underpins many limits on static analysis and warns that perfect pre-deployment verification may be impossible; ethical responsibility therefore extends to runtime monitoring and fail-safe design.

---

7 Summary

1. Intellectual property law seeks to **encourage creativity** by granting time-limited, enforceable rights.
2. Ownership of software hinges on context—employment, commissioning agreements and national statutes.
3. Trade secrets, copyrights, patents and trademarks offer different balances of disclosure, protection and cost.
4. Licences convert ownership into economic or social value but also allocate risk.
5. Ethical reasoning frameworks—teleological, deontological, contractarian—provide structured ways to justify actions.
6. Professional codes translate high-level values into day-to-day guidance.
7. Real-world case studies reveal how technical decisions intersect with law, business and human welfare.
8. The Halting Problem reminds us that perfect foresight is unattainable; ethics fills the gap between what we *can* calculate and what we *must* decide.

---

8 Self-Test Questions

1. A start-up asks you to replicate the Google Maps API so existing apps will work unchanged on a rival map service. Which IP mechanism(s) are you potentially infringing, and how would you analyse the situation using utilitarian and deontological frameworks?

2. In the Googalzon Consultants scenario, list the prima facie duties at stake and explain how each influences your decision.

3. Compare trade secret protection and patent protection for an algorithm that predicts engine failure from vibration data. What are the commercial and ethical trade-offs?

4. The left-pad chain accident raises questions of liability. Trace the flow of responsibility through library author, integrator and system operator, citing at least three legal theories.

5. How does the Halting Problem limit automatic detection of harmful code paths in systems like Therac-25, and what ethical obligations follow from that limit?

Reflect on your answers; the open-ended INFO1111 exam will expect you to integrate legal facts, ethical reasoning and system-level thinking.

Chapter 9

INFO1111 – 2025 S1
Week 07 B: Ethical Reasoning Beyond the Letter of the Law

1 What Ethics Adds to Professional Practice

A system of ethics is a system of moral principles—principles of right and wrong and principles of conduct that govern the behaviour of an individual or a group. Technical competence is only part of professional responsibility. You must also judge whether particular actions *ought* to be taken, even when they are legally permissible, or whether actions *must* be resisted, even when they are legally required.

2 Ethics, Morality and Law—Three Distinct Concepts

1. Morality: your personal convictions about good and evil.

2. Ethics: socially endorsed rules of conduct that tell professionals what is acceptable.

3. Law: the set of enforceable rules created by governments.

The three frequently overlap but are not identical. A practice may be:

- Legal and moral – e.g. designing a system to be safe.
- Legal yet immoral – e.g. Australian "human zoos" that displayed Aboriginal people in the 1940s.
- Illegal yet moral – e.g. parking in a no-parking zone to help an injured person.
- Illegal and immoral – e.g. killing an innocent person.

These four categories are easily visualised in a 2×2 matrix:

	Moral	Immoral
Legal	Safe design	Aboriginal human zoo (1940s)
Illegal	Aid injured	Killing an innocent

The matrix warns against assuming that "legal" automatically means "ethical," or that "illegal" automatically means "unethical."

3 From Question to Justified Decision—The Course Expectation

Every INFO1111 ethics task, including open-ended exam questions, asks you to:

1. Read and understand a scenario.

2. Identify the ethical dilemma.

3. Reach a decision (yes, no, or neutral).

4. Explain *why* you chose that position.

5. Relate your reasoning to at least one ethical framework.

Marks reward the clarity and relevance of your reasoning, not *which* decision you pick.

3.1 Practical Study Hints

Students often ask, "Do we have to memorise everything?" Memorisation helps, but ethical questions are primarily *interpretive*. Focus on:

- Recognising conflicting duties or values in the scenario.
- Mapping them onto frameworks (utilitarianism, deontology, virtue ethics, contractarianism, etc.).
- Articulating trade-offs succinctly.

Videos supplied in class are illustrative; this chapter provides the same core content in written form for faster revision.

4 Professional Codes that Guide Computing Work

Two authoritative sources set explicit expectations for computing professionals:

- Association for Computing Machinery (ACM) Code of Ethics: https://www.acm.org/code-of-ethics
- Australian Computer Society (ACS) Code of Professional Conduct: https://www.acs.org.au/memberships/professional-ethics-conduct-and-complaints.html

Both emphasise duties already introduced in Chapter 8—primacy of public interest, avoidance of harm, honesty, competence and respect for intellectual property—but you should now view them through the lens of the legal–moral matrix above. A course of action that satisfies company policy or local law can still violate ACS or ACM imperatives.

5 Case Study: Open-Source Code, Deadly Consequences

Scenario
"I wrote a game app in my spare time and released the source code openly. Someone reused that code in a traffic-control system. A latent bug caused the system to fail, leading to car crashes and three deaths. Do I bear any ethical responsibility?"

5.1 Analyse the Dilemma

Stakeholders
- Original developer (you)
- Integrator who adopted the code
- Traffic-system operator and the public

Key Facts
- Code was intentionally made freely available.
- The integrator chose it for a safety-critical application.
- A defect directly contributed to loss of life.

5.2 Applying Ethical Frameworks

1. Utilitarianism
   – Ask which action now maximises overall welfare. Options include issuing a patch, assisting investigations, or accepting liability to deter negligence.

2. Deontological (duty-based)
   – Did you have a duty of care to ensure the code was safe for *any* foreseeable use? Some argue that releasing under an "as-is, no warranty" licence fulfils the duty; others counter that foreseeability imposes higher

obligations.

3. Professional Codes
   – ACM: "Avoid harm." Even if legal disclaimers exist, you may be ethically obliged to help remediate.
   – ACS: "Primacy of public interest." Public safety outranks convenience or legal defence.

5.3 Possible Conclusions

Legal liability might rest primarily with the system integrator, yet many would judge that the original author still has an ethical responsibility to:

- Cooperate fully in post-incident analysis.
- Disclose known limitations or latent defects promptly.
- Strengthen documentation or testing for future users.

A well-argued answer can defend *or* reject such duties, provided the reasoning is explicit and grounded in a recognised framework.

6 Bringing It All Together for Exam Success

1. Distinguish clearly between what the law requires and what professional ethics demand.

2. Use the legal–moral matrix to categorise actions quickly.

3. When faced with a scenario, follow the five-step expectation checklist.

4. Cite at least one clause from the ACM or ACS codes to reinforce your argument.

5. Remember: your score depends on the depth and structure of your analysis, not on choosing a particular moral stance.

Self-Check Questions

1. Give an example (other than the ones above) of something that is legal but, in your view, immoral. Explain why it falls in that quadrant.

2. In the open-source traffic-control case, outline how a strict deontologist could completely absolve the original developer of responsibility.

3. Suppose the integrator had modified the code and introduced the bug themselves. How would that change your utilitarian analysis?

Reflect on these questions until you can defend each answer from more than one ethical perspective—exactly the skill the INFO1111 Advanced Knowledge Exam will probe.