Software Engineering 2025–26

Lecture 1: Engineering Mindset & The Cost of Failure

Or: Why do we need this course?

Prof. Robert Harle

Department of Computer Science and Technology
University of Cambridge

Easter Term

Lecturer Introduction & Course Objectives

Welcome to Software Engineering!

Objectives for this term:

- Transition from writing scripts to engineering systems.
- Understand how to manage code complexity at scale.
- Learn industry-standard processes for team collaboration.
- Recognize the ethical and safety implications of software design.

Syllabus Overview

Over the next 6 lectures, we will cover the software lifecycle:

1. **Lecture 1:** Engineering Mindset & The Cost of Failure
2. **Lecture 2:** Requirements and Specifications
3. **Lecture 3:** Engineering Process
4. **Lecture 4:** Quality Assurance & Reliable Delivery
5. **Lecture 5:** Software Evolution
6. **Lecture 6:** Software Engineering in the AI Era

Assessment & Practicals

How you will be assessed:

- **Paper 2 (Written Exam):** Theoretical concepts, process understanding, and case study analysis.
- **IB Group Project (Practicals):** You will apply these engineering principles next year.
  - Teamwork is mandatory.
  - Version control (Git) is mandatory.
  - You will be graded on process, not just the final working code.

What is the "Software Crisis"?

1968 NATO Software Engineering Conference (Garmisch, Germany)

The realization that software was fundamentally different from hardware:

- Projects routinely ran over budget and over time.
- Systems were delivered with massive inefficiencies.
- Code was unmaintainable and broke frequently.
- **The pivot:** The term "Software Engineering" was coined to demand a rigorous, systematic approach to code creation.

Programming vs. Engineering

Programming

- Single developer
- Small scale ($< 1,000$ lines)
- Short lifespan (days/weeks)
- Primary metric: "Does it work today?"

Software Engineering

- Teams of developers
- Massive scale ($1,000,000+$ lines)
- Long lifespan (years/decades)
- Primary metrics: Maintainability, Security, Extensibility

The Mythical Man-Month & Brooks's Law

Why we can't just "add more developers"

"Adding manpower to a late software project makes it later." – **Fred Brooks (1975)**

- **Training Overhead:** Senior engineers must stop coding to mentor new hires.
- **Communication Overhead:** As team size (n) grows, the number of communication paths grows quadratically: $\frac{n(n-1)}{2}$.
- Software is not like digging a ditch (perfectly partitionable); it is highly interdependent.

The Complexity of State

Why is software harder to predict than physical bridges?

- Physical structures have continuous, predictable stress limits.
- Software state is **discrete** and exponentially vast.
- A single flipped bit (a 0 instead of a 1) can change a system from a perfectly safe state to a catastrophic failure state instantly.
- Exhaustive testing of all possible software states is mathematically impossible.

Hardware vs. Software Curves

Moore's Law

Hardware capabilities (transistor count) historically doubled every two years.

- As hardware capacity exploded, the *ambition* of software exploded.
- Human cognitive ability to manage complexity did **not** scale at the same rate.
- The gap between what hardware can do and what humans can safely program is where engineering processes are required.

Defining "Systems"

Software rarely exists in a vacuum. We build **Systems**.

- **Components:** Databases, front-ends, APIs, hardware sensors, network layers.
- **Emergent Behavior:** The system acts in ways its individual components do not (both positively and negatively).
- A bug is often not in a single function, but in the *interaction* between two perfectly written components.

The Cost of Software

Where does the money go?

- Writing the initial code is the *cheapest* part of the software lifecycle.
- Students often optimize for speed of writing.
- Industry optimizes for speed of **reading** and **modifying**.

The Maintenance Phase

Maintenance includes:

- **Corrective:** Fixing discovered bugs.
- **Adaptive:** Making it work in a changed environment (new OS, new hardware).
- **Perfective:** Adding new features demanded by users.
- **Preventive:** Refactoring code to prevent future decay.

The 80/20 Rule of Thumb

Lifecycle Distribution

Typically, **20%** of a software's total cost is spent on initial development, while **80%** is spent on maintenance and evolution over its lifetime.

Implication: Code must be written for the engineer who will read it five years from now.

The Concept of "Software Rot"

Does software decay?

- Software does not physically rust or wear out.
- However, it suffers from **Bit Rot** (or Software Entropy).
- The environment changes (libraries update, security protocols deprecate, APIs change).
- Software that is not actively maintained will eventually fail, even if its source code never changes.

Transitioning to Formal Engineering

How do we survive this complexity? We adopt methodologies.

- Requirements elicitation (knowing what to build).
- Architectural design (planning the structure).
- Automated testing (verifying state).
- Version control (tracking history).

Before we look at the solutions (Lectures 2-6), let's look at what happens when we fail.

Why Study Failure?

> *"Engineering is the art of making what you want from things you can get,* **without making mistakes that will kill you.**"

By studying failure, we learn the edge cases of human limitation and system design. Let's examine some case studies of failures to understand the sorts of things that go wrong.

Case Study 1: Therac-25 (1985-1987)

Background:

- A medical linear accelerator used for radiation therapy.
- Designed to destroy tumors using high-energy radiation.
- Built by Atomic Energy of Canada Limited (AECL).
- Successor to the Therac-20.

Therac-25: Hardware vs. Software

The Fatal Flaw in Architecture

- Therac-20 had **hardware interlocks** (physical circuits that prevented the electron beam from firing without the targeting shield in place).
- Therac-25 removed these expensive hardware interlocks.
- Safety was entrusted **entirely to software control**.

Therac-25: The Race Condition

The Concurrency Bug:

- The software used a shared variable to track operator input.
- If the operator typed "X" (X-ray mode), realized a mistake, and quickly pressed "Up" and "E" (Electron mode)...
- A **race condition** occurred. The system configured the high-power beam but did not deploy the physical target shield.

Therac-25: UI Failures

Cryptic and Dismissible Errors

- The machine would halt and display an error like **"Malfunction 54"**.
- Operators had no manual explaining what "Malfunction 54" meant.
- The system allowed operators to simply press "P" (Proceed) to bypass the error and fire again.

Therac-25: The Cost

The Human Tragedy

- Between 1985 and 1987, at least six patients received massive radiation overdoses (up to 100 times the intended dose).
- Multiple patients died as a direct result of radiation poisoning.
- AECL initially denied any possibility of software error.

Therac-25: Lessons Learned

Engineering Takeaways for Safety-Critical Systems:

- **Defense in Depth:** Never rely on a single software check for human safety. Hardware interlocks are essential.
- **Concurrency is dangerous:** Shared state in multi-threaded environments must be strictly managed.
- **UX is safety:** Error messages must be actionable, clear, and difficult to blindly dismiss.

Case Study 2: Mars Climate Orbiter (1999)

Background:

- A $327 million robotic space probe launched by NASA.
- Mission: Study the Martian climate and atmosphere.
- Joint effort between NASA's Jet Propulsion Laboratory (JPL) and Lockheed Martin Astronautics.

MCO: The Failure

Loss of Signal

- September 23, 1999: The orbiter began its Mars orbit insertion maneuver.
- The spacecraft passed behind Mars and its radio signal was lost.
- The signal was never re-established. The spacecraft had disintegrated in the upper atmosphere.

MCO: The Root Cause

A Failure of Units

- Lockheed Martin (Colorado) wrote the software for thruster output using **Imperial units** (pound-seconds).
- NASA JPL (California) wrote the navigation software expecting **Metric units** (Newton-seconds).
- The navigation software calculated trajectories based on data that was off by a factor of 4.45.

MCO: Team Boundaries

Why wasn't this caught?

- **Interface Definitions:** The Software Interface Specification (SIS) explicitly required metric units. Lockheed Martin failed to adhere to it.
- **Lack of Integration Testing:** Teams tested their own components in isolation but did not run end-to-end simulations together.

MCO: The Cost

Financial and Scientific Loss

- $327 million completely lost.
- Years of scientific planning and potential climate data wasted.
- Severe embarrassment for the space agency.

MCO: Lessons Learned

Engineering Takeaways for Integration:

- **Type Safety:** Modern programming practices (like strongly typed languages) can catch unit errors at compile time.
- **Integration Testing:** Component isolation is not enough; the boundaries between teams are where systems fail.
- **Process over Blame:** The failure was a systemic process breakdown, not just one programmer's mistake.

Case Study 3: London Ambulance Service (1992)

Background:

- October 1992: Rollout of a new Computer Aided Dispatch (CAD) system for the LAS.
- Goal: Automate ambulance dispatch to improve response times across London.

LAS: Non-Technical Issues

A Doomed Procurement Process

- The contract was awarded to a company with no prior experience building emergency CAD systems.
- They won by significantly underbidding the competition.
- Timelines were heavily compressed due to political pressure.

LAS: Hostile User Base

Ignoring the Human Element

- The system tracked ambulance locations via imperfect radio systems.
- Paramedics felt micromanaged and distrusted the new system.
- Due to lack of training, paramedics often pressed the wrong status buttons, feeding the system corrupt data about their availability.

LAS: System Overload

The Day of Deployment

- The system suffered a memory leak and rapidly slowed down.
- Ambulances were dispatched multiple times to the same address, while other critical calls were ignored.
- As response times climbed, citizens called emergency lines repeatedly, flooding the system with duplicate entries.

LAS: The Fallback

Total Collapse

- Within 36 hours, the system locked up entirely.
- The control room was forced to revert to the old manual system of paper slips and whiteboards.
- Tragically, numerous patients died waiting hours for ambulances that never arrived.

LAS: Socio-Technical Systems

Software does not exist in isolation.

- Software is part of a **socio-technical system** that includes human psychology, organizational culture, and hardware.
- A system designed perfectly in code will fail if the humans operating it do not understand, trust, or accept it.

LAS: Lessons Learned

Engineering Takeaways for Deployment:

- **Phased Rollouts:** Never do a "Big Bang" deployment for a critical system. Roll out incrementally.
- **User Buy-in:** Users must be involved in the design process to ensure the software fits their real-world workflow.
- **Load Testing:** Systems must be tested under extreme, anomalous load, not just "happy path" conditions.

Case Study 4: The UK Post Office Scandal (Horizon)

Background:

- Introduced in 1999: The Horizon IT system (by Fujitsu) was meant to automate accounting for thousands of local post offices.
- **The Issue:** Subpostmasters began reporting inexplicable financial shortfalls in their accounts.
- **The Stance:** The Post Office insisted Horizon was "robust" and "effectively infallible," blaming the subpostmasters for the missing money.

Horizon: The Cost of Infallibility

Software as Evidence

- **The Reality:** The system had multiple critical bugs that could silently alter financial records. Fujitsu's developers were aware of these bugs.
- **Consequences:** Over 700 subpostmasters were prosecuted for theft and false accounting. Many were imprisoned, went bankrupt, or took their own lives.
- **Takeaway:** Software bugs can have devastating real-world legal and human consequences. A blind trust in "the system" is a dangerous engineering and management failure.

Horizon: Lessons Learned

Socio-Technical Takeaways:

- **Audit Trails:** In financial or legal systems, every automated action must be transparent and auditable by a human.
- **Engineering Integrity:** Developers identified the bugs but were ignored by management. Technical truth must override business convenience.
- **Presumption of Guilt:** The law (at the time) presumed computer evidence to be correct unless proven otherwise. Engineers must challenge this "myth of infallibility."

Case Study 5: Boeing 737 MAX (MCAS)

Software compensating for Hardware

- **The Problem:** To fit larger engines, Boeing moved them forward and higher. This changed the plane's aerodynamics, making it prone to "pitching up."
- **The Solution (MCAS):** A software system designed to automatically push the nose down if it detected a high angle of attack.
- **The Failure:** MCAS relied on a *single* sensor. If that sensor failed, the software would repeatedly force the plane into a dive.

Boeing 737 MAX: Lessons Learned

Engineering Takeaways for Safety-Critical Systems:

- **Single Point of Failure:** Never design a critical safety system that relies on data from only one sensor.
- **Complexity Hiding:** To avoid pilot retraining costs, Boeing "hid" the existence of MCAS. Software must not act autonomously without the operator's knowledge.
- **The "Software Fix" Trap:** Don't use software as a "patch" for fundamental hardware or design flaws unless it is built with absolute rigour.

From Individual to Team Member

The Shift in Mindset

- As you transition to engineering, you are no longer just coding for yourself.
- You are writing code for your team, your company, and your users.
- Clarity, documentation, and reliability become more important than clever, convoluted algorithms.

Code as a Liability

A Hard Truth

- Managers often view lines of code as an asset.
- Engineers must view lines of code as a **liability**.
- Every line you write is a line you have to test, maintain, and secure against vulnerabilities.
- The best code is the code you didn't have to write.

Introduction to Technical Debt

Technical Debt (Ward Cunningham, 1992)

A metaphor comparing the trade-off of writing quick, messy code (to meet a deadline) to taking out a financial loan.

- **Principal:** The effort required to eventually rewrite the code properly.
- **Interest:** The extra time it takes to add new features or fix bugs because the messy code slows you down.

Intentional vs. Unintentional Debt

Not all debt is bad.

- **Intentional Debt:** "We will hardcode this database connection to hit our launch deadline on Friday, and fix it next sprint." (Strategic).
- **Unintentional Debt:** Writing poorly structured code because of a lack of skill or understanding of the domain. (Dangerous).

The Interest Rate of Tech Debt

Why projects grind to a halt:

- As debt accumulates, the "interest payments" take up all the developers' time.
- Every new feature breaks three old features.
- Morale plummets. Developers leave.
- Eventually, the company declares "Technical Bankruptcy" and has to rewrite the entire system from scratch.

Paying Down the Debt: Refactoring

Refactoring

The process of restructuring existing computer code without changing its external behavior.

- Teams must dedicate time to "pay down" debt by refactoring.
- Refactoring is impossible without automated tests (how else do you prove you didn't change the behavior?).

Professional Responsibility

Your Ethical Duty

- As engineers, you hold specialized knowledge that management and users do not have.
- You have a professional responsibility to push back against unrealistic deadlines if they compromise safety or core quality.
- You cannot rely on "management told me to do it" as an excuse for catastrophic failure.

The Reality of Tradeoffs

The Iron Triangle

- Fast, Good, Cheap. You can only pick two.
- Software engineering is a continuous negotiation of trade-offs.
- Perfect code shipped two years late is a failure.
- Terrible code shipped tomorrow that bankrupts the company in technical debt is a failure.
- Finding the balance is what makes you an engineer, not just a programmer.

Review of Lecture 1 Themes

What we covered today:

- The shift from programming to Software Engineering.
- The overwhelming cost of software maintenance.
- How complexity, concurrency, and poor integration lead to deadly failures (Therac-25, MCO).
- How socio-technical and legal factors doom projects (LAS, Horizon).
- The concept of Technical Debt and code as a liability.