COMP433

Software Engineering

Introduction to software engineering
Introduction to SLDC: Software Development Lifecycle
Course Outlines

Hisham Ihshaish

Birzeit University

Semester 2 (Spring) 2025/2026 | Sections 5 & 6

What we will cover today

01 **When software goes wrong**
High-profile failures and their consequences

02 **Defining software engineering**
What it is and how it differs from programming

03 **Why we need it**
The software crisis and characteristics of modern software

04 **The development life cycle**
A first look at SDLC and its core activities

05 **Professional responsibility**
Ethics, codes of conduct, and real-world cases

06 **Course overview**
Structure, assessment, tools, and expectations

When software goes wrong

High-profile failures that shaped our discipline

Software failures with real consequences

Ariane 5 (1996)

A 64-bit to 16-bit conversion overflow caused the rocket to self-destruct 37 seconds after launch. Cost: $370 million.

Reusing untested legacy code in a new context

Knight Capital (2012)

A deployment error activated obsolete trading code, causing $440 million in losses in 45 minutes.

Inadequate deployment and configuration management

Boeing 737 MAX (2018--2019)

The MCAS flight control system, relying on a single sensor, contributed to two crashes and 346 deaths.

Insufficient redundancy and poor requirements validation

Why these failures matter

Software is everywhere. It controls medical devices, financial systems, transport, energy grids, and communications. The consequences of failure are no longer limited to inconvenience; they can be catastrophic.

This is why we engineer software, rather than simply write code.

Scale	Modern systems contain millions of lines of code and integrate with dozens of other systems.
Complexity	Software is intangible and its behaviour is difficult to predict without systematic methods.
Change	Requirements evolve continuously; software must be designed to accommodate change.

Defining software engineering

From programming to engineering discipline

What is software engineering?

IEEE definition

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Sommerville's definition

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

Both definitions share a common thread: software engineering is not just about writing code. It encompasses the entire life of a software system, from conception to retirement, using systematic methods.

Software engineering, computer science, and programming

Computer science

Concerned with **theories and foundations** that underpin computation.

Focuses on algorithms, data structures, formal methods, and computability.

Asks:
"Can this be computed, and how efficiently?"

Software engineering

Concerned with the **practical challenges** of producing software.

Focuses on processes, design, quality, teamwork, and managing complexity.

Asks:
"How do we build software that works, on time, within budget, and can evolve?"

Programming is a core activity within software engineering, but SE also covers requirements, design, testing, project management, and maintenance.

The building analogy

Building construction	Software construction
Laying bricks	Writing code
Architectural drawings	System design and UML models
Surveying the site	Requirements elicitation
Building inspections	Testing and quality assurance
Project management	Sprint planning, scheduling, risk management
Renovation and upkeep	Maintenance and evolution

Programming is one activity. Software engineering encompasses the entire construction process.

Why we need software engineering

The software crisis and characteristics of modern software

The software crisis

The term was coined at the 1968 NATO Software Engineering Conference in Garmisch, Germany. It described a growing recognition that software projects were routinely failing.

- Projects delivered late and over budget
- Software that did not meet user needs
- Code that was difficult to maintain or extend
- Unreliable systems with frequent defects
- Inability to scale development to larger teams and projects

These problems have not disappeared entirely, but SE practices have significantly improved outcomes.

What makes software different?

Intangible

You cannot see or touch software. Its behaviour emerges only at runtime, making defects harder to detect than in physical artefacts.

Malleable

Software can be changed more easily than bridges or buildings. This is both an advantage and a risk -- it invites constant change without safeguards.

Does not wear out

Software does not degrade physically, but it decays functionally as the environment around it evolves. This is sometimes called software ageing.

Complex

Even modest applications involve intricate logic, concurrency, and integration. Complexity grows non-linearly with system size.

These characteristics explain why engineering practices -- not just programming skill -- are essential.

The software development life cycle

A first look at how software is built systematically

The software development life cycle (SDLC)

The SDLC describes the stages a software system passes through, from initial concept to eventual retirement. Different process models organise these stages differently, but the core activities remain the same.

1. **Specification**
   Defining what the system should do (requirements engineering)
2. **Development**
   Designing and implementing the system (design and coding)
3. **Validation**
   Checking that the system meets requirements (testing)
4. **Evolution**
   Adapting the system to changing needs (maintenance)

We will explore different ways of organising these activities in Lectures 2 and 3: waterfall, V-model, incremental, and agile.

Verification and validation: a crucial distinction

Verification

"Are we building the product right?"

Checks that the software conforms to its specification. Focuses on the process and intermediate products.

Techniques: code reviews, inspections, static analysis, walkthroughs.

Validation

"Are we building the right product?"

Checks that the software meets the user's actual needs. Focuses on the final product and user expectations.

Techniques: user acceptance testing, prototyping, beta testing, demonstrations.

Boehm's formulation (1979) remains the clearest way to remember the distinction. Both are essential; neither alone is sufficient.

Professional responsibility

Ethics, codes of conduct, and accountability in software

The ACM/IEEE-CS code of ethics

The ACM/IEEE-CS Software Engineering Code of Ethics and Professional Practice (1999) establishes eight principles that software engineers should adhere to. It represents the profession's commitment to public welfare.

1.	**Public**	Act in the public interest
2.	**Client and employer**	Act in the best interests of client and employer, consistent with public interest
3.	**Product**	Ensure products meet the highest professional standards
4.	**Judgement**	Maintain integrity and independence of professional judgement
5.	**Management**	Promote an ethical approach to software management
6.	**Profession**	Advance the integrity and reputation of the profession
7.	**Colleagues**	Be fair to and supportive of colleagues
8.	**Self**	Participate in lifelong learning and promote ethical practice

When ethics were tested

Volkswagen emissions scandal (2015)

Engineers wrote software that detected when a vehicle was being tested for emissions and reduced pollutant output accordingly. During normal driving, emissions were up to 40 times the legal limit. The software was deliberately designed to deceive regulators.

Violated principles 1 (public), 3 (product), and 4 (judgement)

Horizon Post Office scandal (UK)

Fujitsu's Horizon accounting system contained bugs that made it appear sub-postmasters were stealing money. Over 900 were wrongly prosecuted between 1999 and 2015. Many were imprisoned, bankrupted, or took their own lives.

Violated principles 1 (public), 3 (product), and 6 (profession)

In both cases, software decisions had devastating human consequences. Professional responsibility is not optional.

Course overview

Structure, assessment, and expectations

Course structure: six blocks, fifteen lectures

1	**Introduction to SE**	1 session	Lecture 1 (today)
2	**Software processes and models**	2 sessions	Lectures 2--3
3	**Requirements engineering**	3 sessions	Lectures 4--6
4	**System modelling with UML**	6 sessions	Lectures 7--12
5	**System and architectural design**	2 sessions	Lectures 13--14
6	**Detailed design**	1 session	Lecture 15

The largest block is UML modelling (40% of the course). Start forming your project groups now.

Assessment breakdown

25%

Midterm exam

Covers foundational concepts, processes, and requirements (Blocks 1--3)

40%

Group project, assignments, and exercises

Phased submissions across the semester; the backbone of practical learning

35%

Final exam

Comprehensive, covering all blocks including UML and design

The group project has multiple phased submissions. Late submissions are not accepted.

Textbooks and tools

Primary textbooks

- **R1** Bruegge & Dutoit -- Object-Oriented SE Using UML, Patterns, and Java (3rd ed.)
- **R2** Sommerville -- Software Engineering (9th/10th ed.)
- **R3** Seidl et al. -- UML @ Classroom (Springer, 2015)

UML modelling tools

- **Lucidchart** Free educational licence (use your university email)
- **Astah** Free student licence available
- **Enterprise Architect / Rational Rose** Full UML-compliant modelling environments

AI-generated UML diagrams are not permitted. Submissions containing them will receive zero.

Discussion: what went wrong?

Scenario

A university builds a new student registration system. After two years of development, it launches on the first day of the semester. Students cannot register for courses. The system crashes under load. Faculty cannot see enrolment lists. The university reverts to paper forms for a week.

Think about what we covered today and discuss in the Teams chat:

1. Which SDLC activity was likely neglected? (specification, development, validation, evolution)
2. Which principle from the code of ethics is most relevant here?
3. What is one concrete practice that might have prevented this?

Take 3 minutes. Type your answers in the Teams chat.

Summary

- Software failures have real consequences: financial loss, legal liability, and loss of life.
- Software engineering applies systematic, disciplined methods to the entire software lifecycle.
- It is distinct from computer science (theory) and programming (a single activity within SE).
- The software crisis of the 1960s motivated the discipline; its challenges persist in evolved forms.
- The SDLC organises work into specification, development, validation, and evolution.
- Verification asks "are we building it right?"; validation asks "are we building the right thing?".
- The ACM/IEEE code of ethics places public interest above all other obligations.

Next session

Lecture 2: Software processes and models (part 1)

- What is a software process?
- The waterfall model: origins, stages, and limitations
- The V-model and incremental development
- Plan-driven versus agile: a first comparison

Preparation

Read Sommerville (R2) Chapter 2, paying particular attention to the description of the waterfall model and its critique. Also form your project group (3--4 students) by next week.

References

Bruegge, B. & Dutoit, A. (2013). Object-Oriented Software Engineering Using UML, Patterns, and Java. 3rd Edition. Prentice Hall.

Sommerville, I. (2016). Software Engineering. 10th Edition. Pearson.

Seidl, M., Scholz, M., Huemer, C. & Kappel, G. (2015). UML @ Classroom. Springer.

IEEE (1990). Standard Glossary of Software Engineering Terminology. IEEE Std 610.12.

ACM/IEEE-CS (1999). Software Engineering Code of Ethics and Professional Practice.

IEEE Computer Society (2024). SWEBOK: Guide to the Software Engineering Body of Knowledge. Version 4.

Brooks, F.P. (1987). No Silver Bullet: Essence and Accident in Software Engineering. IEEE Computer, 20(4), 10--19.

Boehm, B. (1979). Guidelines for Verifying and Validating Software Requirements and Design Specifications. Euro IFIP.

Naur, P. & Randell, B. (Eds.) (1968). Software Engineering: Report on a Conference. NATO Science Committee.