



# 软件工程理论基础

## Theoretical Foundations of Software Engineering

By:

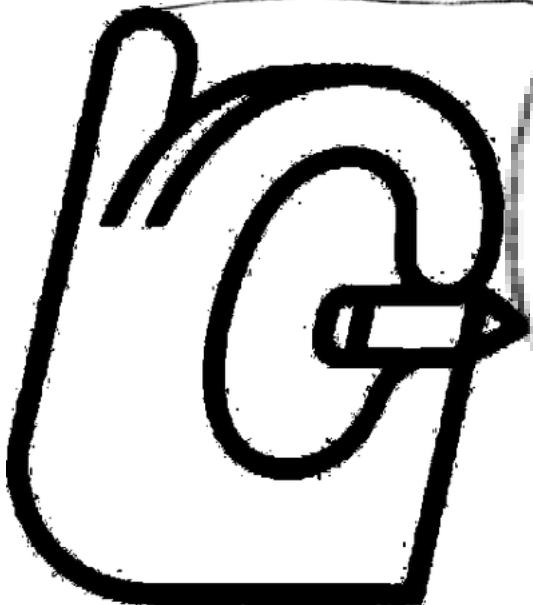
FAHAD SABAH

北京工业大学计算机学院（国家示范性软件学院）





# Contents



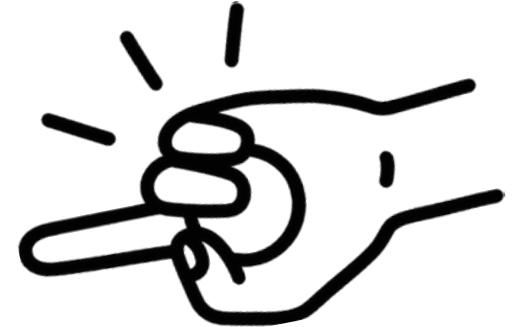
- Course Kick-off
- Introduction to Software Engineering
  - Definition and scope of software engineering.
  - Theoretical vs. practical perspectives.
- Historical evolution
- Synthesis & Forward Look

# 01

Course Kick-off



# Welcome & Syllabus



1.1

Instructor Introduction

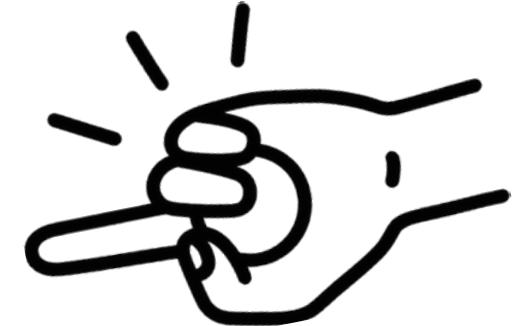
1.2

Syllabus Overview

1.3

Activity

# 1.1 Instructor Introduction



Fahad Sabah (PhD)

College of Computer Science,  
Beijing University of  
Technology, China

Research Interest(s):

Federated Learning, Machine  
Learning, Scientometrics

## Teaching & Research Experience:

- Superior University, Pakistan  
(November 2014 – November 2022)
- 25+ Top Tier Publications
  - Expert Systems with Applications
  - Information Fusion
  - Journal of Supercomputing
  - And many others...
- Citations (October 2025): 300
- H-Index: 8
- i-10 Index: 7

Expertise in software engineering:  
Worked for various industries and  
individual clients

- SoftTechHive
- SoftwareVilla
- Stepping Stones
- RealDeals
- Ceats

# 1.2 Syllabus Overview



□ Overview of the course goals and objectives.

□ Key components of the syllabus are highlighted, including grading policies, major assignments, and project weights.

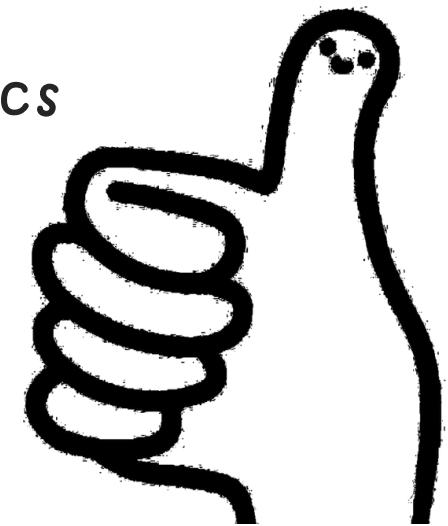
课程名称（中文） Course Title (Chinese)	软件工程理论基础（英文） Theoretical Foundations of Software Engineering			
课程名称（英文） Course Title (English)	Theoretical Foundations of Software Engineering			
适用层次(可多选) Applicable level(s)	<input checked="" type="checkbox"/> 学硕 (Academic master)	<input type="checkbox"/> 学博 (Academic Ph.D)	<input checked="" type="checkbox"/> 专硕 (Professional master)	<input type="checkbox"/> 专博 (Professional Ph.D)
授课语言 Teaching Language	英语 English	适用学科/专业 学位类别(领域) Discipline	软件工程 Software Engineering	
学分数 Course Credit(s)	48	开课学期 Semester		
总学时 Teaching Hours in Total	共 48 学时 48 teaching hours	实验/实践学时 Hours for Experiments /Practice	共 0 学时 0 teaching hours	

# 1.2 Syllabus Overview

- ❑ Overview of the course goals and objectives.
- ❑ Key components of the syllabus, including grading policies, major assignments, and project weights.

## Course Objective

- ❑ Formal Foundations for Complex Systems
- ❑ Logic Tools and Quality Metrics
- ❑ Ethics Economics Emerging Tech
- ❑ Critical Cases Distributed Futures



# Course Objective(s)

# Formal Foundations for Complex Systems

## Theoretical Groundings

This course provides a robust theoretical foundation in software engineering, enabling students to analyze, design, and verify complex software systems using formal methods. It emphasizes the importance of foundational theories to build reliable systems.



## Balancing Abstraction and Practicality

Students will learn to critically evaluate trade-offs between abstraction, modularity, and practical implementation, ensuring that theoretical knowledge is applied effectively to real-world challenges.

# Course Objective(s) Logic Tools and Quality Metrics



01

## Model Checking Tools

Students will master logic-based tools like SPIN and TLA+ to model and verify system behavior, ensuring that software meets specified requirements and operates correctly.

02

## Mutation Testing

The course covers mutation testing techniques, enabling students to identify and fix potential defects in software, thereby enhancing overall quality and reliability.

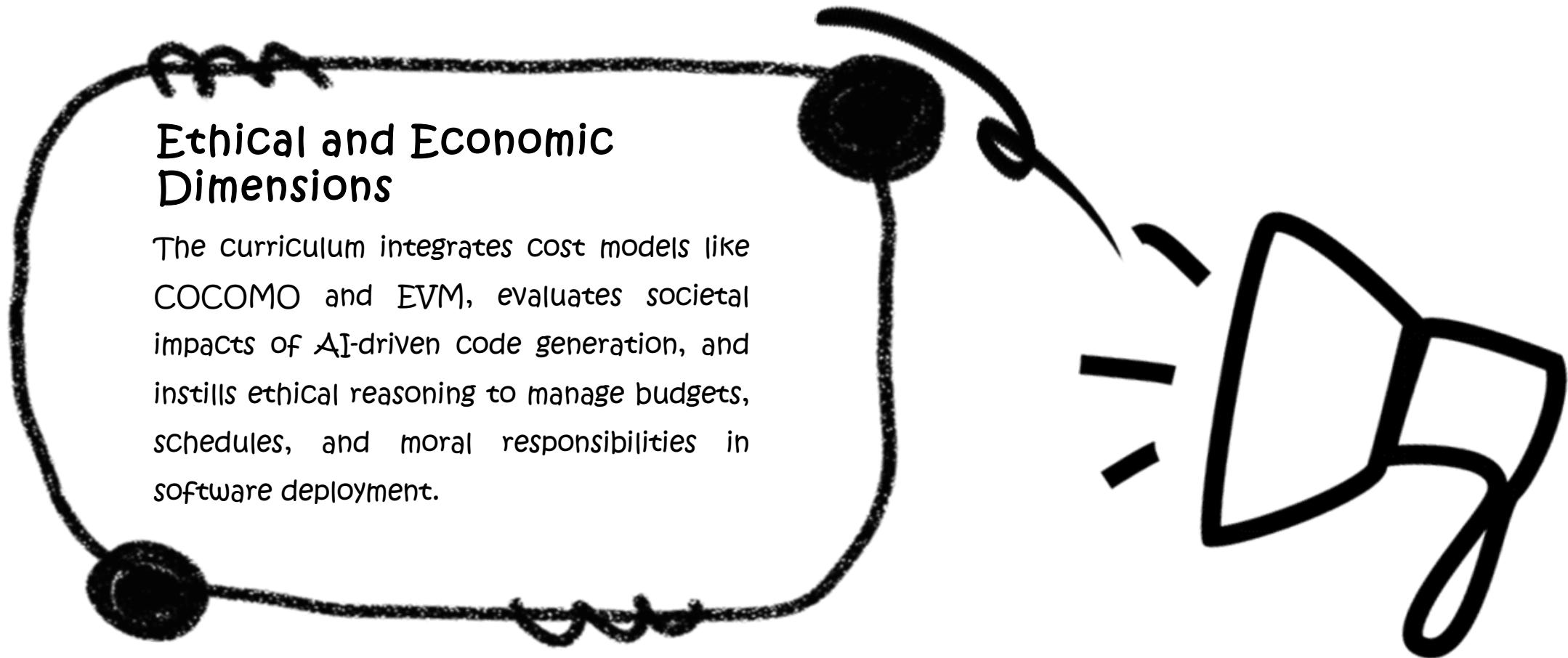
03

## Complexity Metrics

Students will learn to compute complexity metrics such as cyclomatic and Halstead measures to quantify software complexity, helping to predict and manage potential issues.

# Course Objective(s)

## Ethics Economics Emerging Tech





# Course Objective(s)

## Critical Cases Distributed Futures

01

Case Studies

02

Anticipating Challenges

03

Ethical Practices

04

Innovative Solutions

By dissecting failures like Therac-25 and successes such as Kubernetes architecture, students develop critical thinking to anticipate challenges in distributed systems and advocate for ethical practices.

Students will learn to identify potential issues in distributed systems, leveraging case studies to understand common pitfalls and develop strategies to mitigate them.

The course emphasizes the importance of ethical engineering practices, ensuring that students can advocate for responsible software development in critical applications.

Students will be equipped to innovate solutions grounded in formal rigor, combining theoretical knowledge with practical insights to advance software engineering.

# 课程考核及成绩评定

# Course Assessment & Grading

考核指标* Assessment Criteria	权重 Percentage	评定标准 Assessment Standard
课堂表现 Participation	10	积极参与课堂讨论 Participation in group discussions
作业/实验 Assignment(s)	20	按时提交所有作业 Timely submission of all assignments 代码符合规范要求 Code meets specified standards 功能实现完整 Complete implementation of required functions 报告内容清晰准确 Clear and accurate reports
课程考试 Course Exam(s).	30	期末考试 (30%) : 综合考核全部课程内容 Final (30%): Comprehensive assessment of all course content
项目 Project	40	项目完整性 (16%) : 实现所有要求功能 Completeness (16%): All required features implemented 代码质量 (12%) : 符合PEP8规范, 结构清晰 Code quality (12%): PEP8 compliant, well-structured 文档 (8%) : 包含清晰的README和注释 Documentation (8%): Clear README and comments 创新性 (4%) : 体现创造性解决方案 Creativity (4%): Demonstrates innovative solutions

# 02

## Defining Software Engineering



# Invisible Code, Visible Impact

## Software in Daily Life

Software is the invisible force that powers our daily lives, from medical devices that keep us healthy to financial systems that manage our wealth. Its decisions can have profound impacts, making it a critical component of modern society.



## Engineering Responsibility

Given its pervasive influence, software engineering is not just a technical pursuit but a societal responsibility. Engineers must ensure that software is reliable, safe, and ethical, as failures can have severe consequences.



## Components of Software

# What Exactly Is Software?

Software comprises **executable instructions**, **data structures**, and **documentation**. It is a logical construct, unlike hardware, and must be engineered with precision to ensure its functionality and reliability.

## Software Characteristics

Software does not wear out physically but can deteriorate due to changes. It is developed, not manufactured, and often requires custom solutions, though reuse is increasingly important.

## Maintenance Challenges

Maintaining software is complex due to its intangible nature. Unlike hardware, it can degrade over time if not properly managed, making disciplined engineering practices essential.

# Software Engineering vs Programming



## Definition of Software Engineering

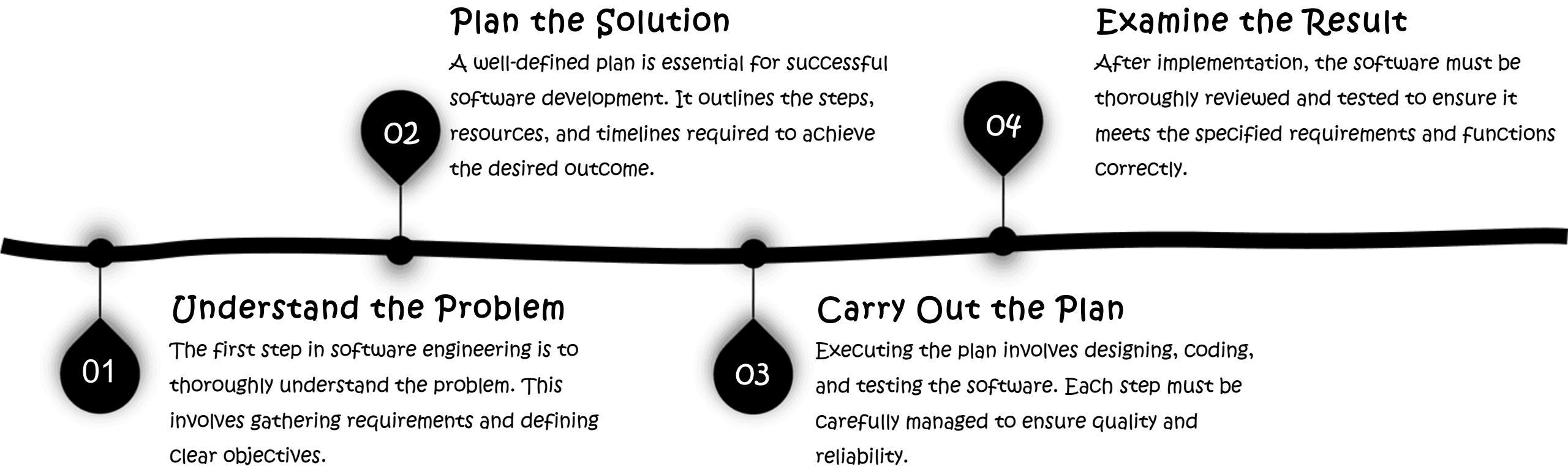
Software engineering is defined as the systematic application of engineering principles to the development of software. It emphasizes a structured approach to manage complexity and ensure quality.



## Contrasting with Programming

Programming focuses on writing code, while software engineering encompasses the entire lifecycle, including requirements, design, testing, and maintenance. It involves teamwork and process adherence.

# Universal Practice Principles



# Domains Where Bugs Hurt

## Diverse Application Domains

Software operates in various domains, each with unique challenges. From embedded systems in medical devices to cloud-based financial applications, understanding the context is crucial for effective engineering.



# 03

## Dangerous Myths



# Management Fiction



## Myth of Standards

01

Managers often believe that having a set of standards ensures quality. However, standards alone do not guarantee success without proper implementation and continuous improvement.

## Adding People to Catch Up

02

The myth that adding more people to a late project will speed it up is pervasive. In reality, this can introduce more complexity and delays.

## Vague Requirements

03

Managers sometimes assume that vague requirements are sufficient, leading to misunderstandings and misaligned expectations.

# Customer & Coder Lore

....

## Customer Misconceptions

Customers often believe that software is infinitely flexible and can be changed easily. This overlooks the complexity and potential risks involved in making changes.

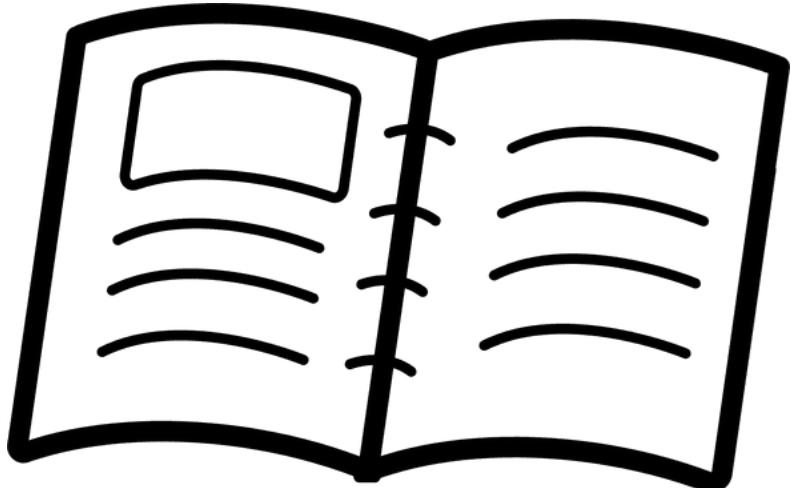
## Coder Misconceptions

Developers sometimes assume that once the code is written, the project is done. However, maintenance, testing, and documentation are equally important.

# 04

## Therac-25 Deep Dive





# Machine That Promised Cure

## Therac-25 Overview

The Therac-25 was a radiation therapy machine designed to deliver precise doses of radiation for cancer treatment. It was marketed as a safer and more efficient alternative to previous models.



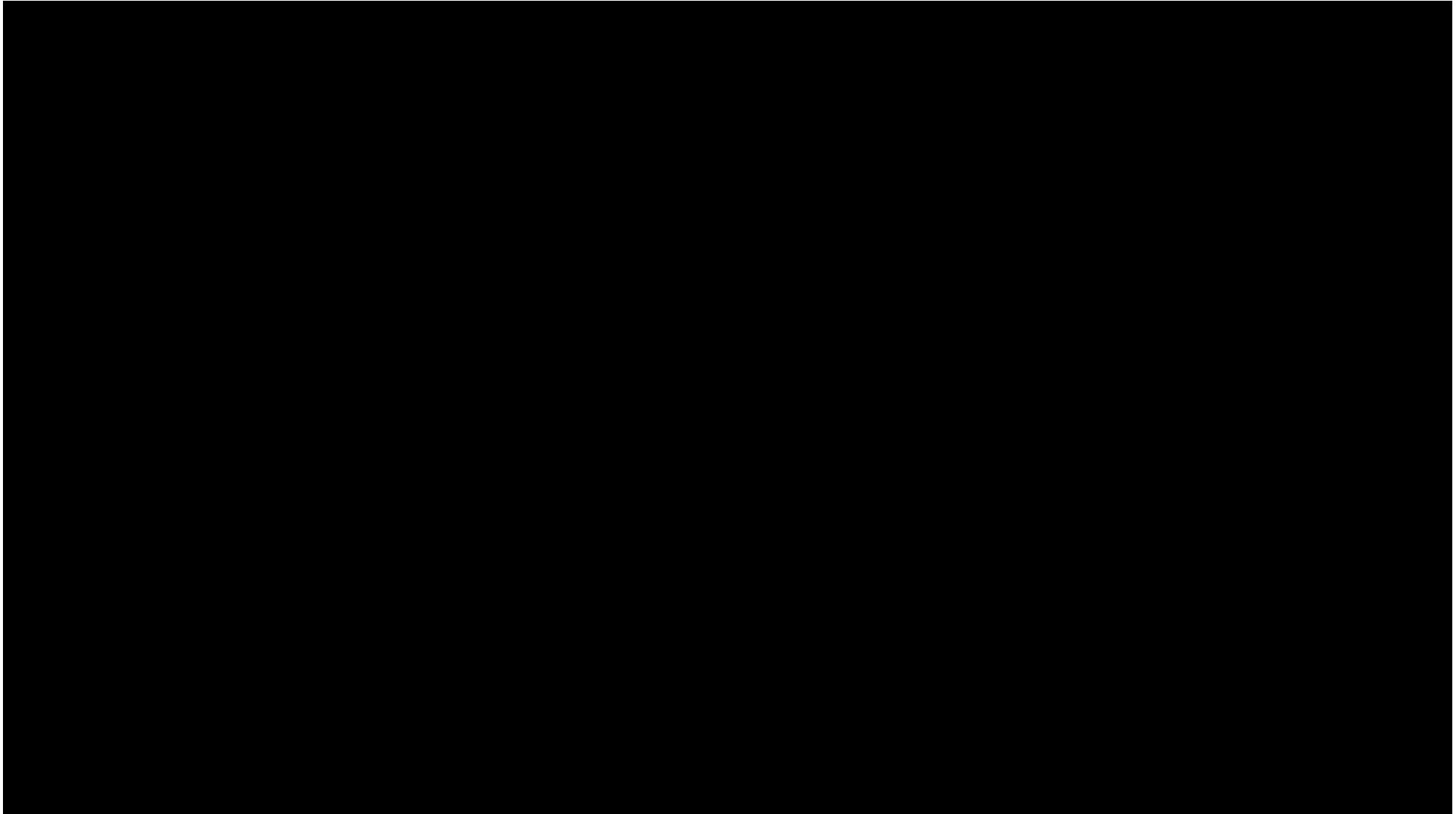
## Underestimated Complexity

Despite its promise, the Therac-25's complexity was underestimated. Its reliance on software control led to a series of catastrophic failures that highlighted the importance of rigorous testing and safety measures.





# Therac-25 Documentary





# Race Condition Explained

## Keyboard-Edit Race

A critical race condition allowed operators to change the energy setting during calibration. This concurrency issue led to high-power electron beams being delivered without proper safety checks.

## Overlapping Flags

The software's flags for concurrent operations overlapped, causing the system to misinterpret commands. This resulted in massive overdoses of radiation to patients.

## Lack of Immediate Feedback

The console displayed 'safe' even when dangerous conditions were present, leading operators to believe the machine was functioning correctly.

# Error Messages Ignored

## Cryptic Error Messages

The system's error messages were vague and easily overlooked. Operators learned to dismiss them, unaware of the potential danger they signaled.

# Missing Hardware Guard

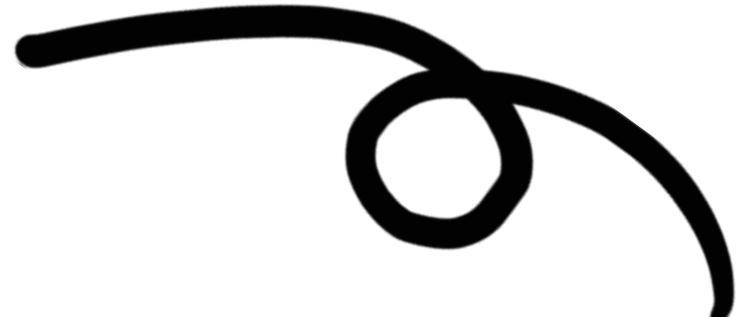
## Removed Mechanical Interlocks

Earlier models of the machine included mechanical interlocks to prevent high-power beams in direct mode. These were removed in the Therac-25, relying solely on software for safety.

## Defense-in-Depth

The absence of hardware safety mechanisms violated the principle of defense-in-depth, which advocates multiple layers of protection to prevent catastrophic failures.

# Testing & Review Gaps



## Lack of Independent Verification



The Therac-25 lacked independent safety reviews and code audits. This oversight allowed critical defects to go unnoticed.

## Inadequate Testing

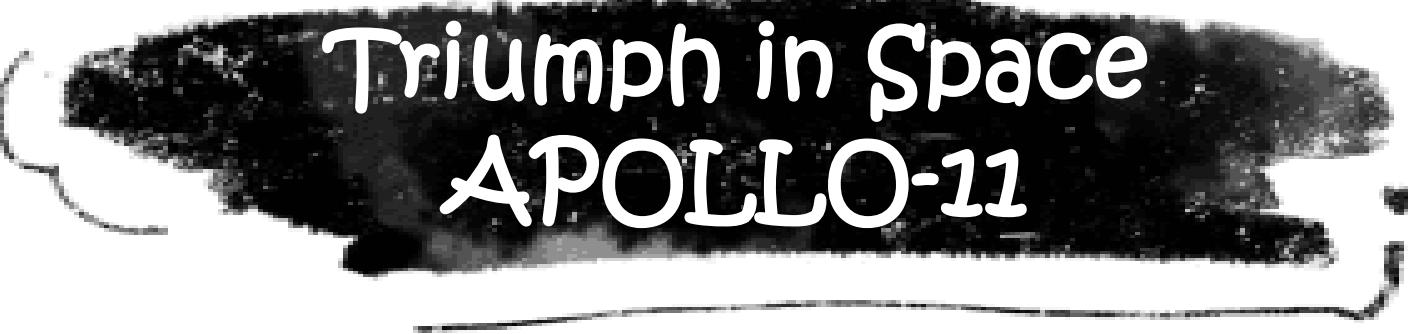


The machine was not thoroughly tested for edge cases and failure modes. This led to the discovery of critical issues only after accidents occurred.

## Overconfidence in Software



Engineers assumed that the software would never fail, leading to a lack of robust error handling and safety mechanisms.



Triumph in Space  
APOLLO-11



1. You're not just learning to code - you're learning to build trust
2. Code with care. Test with rigor. Design with empathy.

See you next time.



# THANK YOU

