Intelligent Software Engineering

When Software Meets Artificial Intelligence

Zhilei Ren

Dalian University of Technology

September 24, 2025

Outline

Outline

About Me

Zhilei Ren

- Professor at Dalian University of Technology
- `https://zhilei.ren`
- `zren@dlut.edu.cn`
- Research Interests:
  - Intelligent software engineering
  - Software testing, fault localization, and automated repairing
  - Dynamic tracing
- `https://github.com/zhileiren/ise-lecture-notes.git`

Research Team

OSCAR

- “**O**perating **S**ystem and **C**ompiler with **A**pplication **R**esearch”
- “**O**ptimizing **S**oftware by **C**omputation from **AR**tificial intelligence”
- “**O**perating **S**ystem will **C**rash whenever my **A**lgorithm **R**uns”

Outline

1843: Ada Lovelace - The First Programmer

- **Augusta Ada King, Countess of Lovelace** (1815-1852)
- Daughter of the poet Lord Byron, trained in mathematics
- Collaborated with Charles Babbage on his **Analytical Engine**
- Translated an Italian article on the engine, adding her own extensive **Notes**
- **Note G** contained what is considered the first computer program

Portrait of Ada Lovelace

Her Vision

She saw beyond mere calculation to the potential for computers to manipulate any symbols, including music and art.

Lovelace's Legacy

- **The Ada Programming Language**: Named in her honor by the US Department of Defense (1980). Designed for large-scale, safety-critical systems.
- **Ada Lovelace Day**: An international celebration held every October to recognize the achievements of women in STEM.
- **Symbolic Importance**: Represents the birth of the idea of software, a century before the first electronic computers.
- **Bridging Arts and Sciences**: Her unique background highlights the creative aspect of programming.

Her Lasting Impact

"Understand well as I may, I cannot yet make it think." - Ada Lovelace. She asked the fundamental question about AI and machine capability that we still grapple with today.

From Vision to Reality

The Long Gap

1840s
Lovelace's Theoretical Program
⇓ **100 Years** ⇓
1940s
ENIAC - First Electronic Computer
⇓
1968
Software Engineering Born

The conceptual foundation laid by Lovelace had to wait for technology to catch up.

The Dawn of Software Engineering

- **1968 NATO Conference**: The term "Software Engineering" was coined
- Addressing the "software crisis" - growing complexity and cost of software development
- Key goal: Apply engineering principles to software development
- Recognition that software development needed disciplined approache

1968: The Birth of a Discipline

- First NATO Software Engineering Conference in Garmisch, Germany
- Focus on reliable, efficient, and economical software
- Established software development as engineering discipline
- Emphasized need for systematic approaches

1968: "Goto Statement Considered Harmful"

- **Edsger Dijkstra's seminal letter** (1968)
- Advocated against use of GOTO statements
- Promoted structured programming principles
- Foundation for modern control structures (if-else, while, for)
- Emphasized code readability and maintainability

Dijkstra's Insight

"Program testing can be used to show the presence of bugs, but never to show their absence!"

1975: The Mythical Man-Month

- **Frederick Brooks' seminal book** (1975)
- Based on experience managing IBM OS/360 development
- **Core thesis**: "Adding manpower to a late software project makes it later"
- Brooks's Law: The "man-month" is a dangerous and fallacious unit of measurement for software work due to communication overhead.
- Distinction between **essential** and **accidental** complexity.

Cover of "The Mythical Man-Month"

Enduring Legacy

The book remains a cornerstone of software project management, high-

1984: Reflections on Trusting Trust

- **Ken Thompson's Turing Award Lecture** (1984)
- Demonstrated the "trusting trust" attack
- A compiler could insert backdoors that propagate themselves
- Fundamental insight into software supply chain security
- Still critically relevant in modern software security

Key Takeaway

The tools we use to build software must themselves be trustworthy. There is no ultimate root of trust.

1980s-1990s: GNU and Open Source Revolution

- **Richard Stallman launches GNU Project** (1983)
- Free Software Foundation established (1985)
- Linux kernel created by Linus Torvalds (1991)
- Open Source Initiative founded (1998)
- Collaborative development model proven successful

1980s-1990s: Object-Oriented Revolution

- **Smalltalk** (1970s-1980s): Pure OOP concepts
- **C++** (1985): Bringing OOP to systems programming
- **Java** (1995): "Write once, run anywhere"
- Key principles: Encapsulation, Inheritance, Polymorphism
- Design Patterns (Gang of Four book, 1994)

Smalltalk C++ Java

1947: The First "Computer Bug"

- **September 9, 1947**: Operators of the Harvard Mark II computer found a moth trapped in a relay.
- The term "bug" had been used in engineering before, but this incident popularized it in computing.
- The moth was taped into the logbook with the note: "First actual case of bug being found."
- **Debugging** literally started as removing insects from hardware.

The first computer "bug" from the Harvard Mark II log book.

1950s-1960s: The Dawn of Software Debugging

- **Print Statement Debugging**: The simplest and most enduring method. Programmers inserted print statements to trace program execution and variable states.
- **Core Dumps**: Analyzing the contents of memory after a program crash. A low-level, complex, but powerful technique.
- **Ad-hoc Testing**: Testing was manual, informal, and often performed by the developers themselves near the end of the project.
- **The Challenge**: As software grew in complexity, these methods became insufficient. The "software crisis" was, in part, a crisis of quality and reliability.

```
printf("Got here!  Value of x = %d \n", x);
```

1970s-1980s: Formalizing Testing and Tools

The Rise of Testing Theory

- **Black-box vs. White-box Testing**: Distinguishing between testing functionality (black-box) and testing internal structures (white-box).
- **Levels of Testing**: Unit, Integration, System, and Acceptance testing became standard concepts.
- **Static Analysis**: Compilers began to include more sophisticated warnings for potential code issues.

The First Debugging Tools

- **Symbolic Debuggers** (e.g., gdb): Allowed programmers to interact with a running program, set breakpoints, and

Early GNU Debugger (GDB) interface.

1990s: Automation and Process Integration

- **Automated Regression Testing**: Tools like JUnit (1997) for Java, created by Kent Beck and Erich Gamma, revolutionized testing.
  - Tests could be run automatically and frequently.
  - Provided a safety net for refactoring and adding new features.
  - Embodied the **Test-Driven Development (TDD)** philosophy.
- **Testing Becomes a Discipline**: The role of **Quality Assurance (QA) Engineer** became specialized.
- **Continuous Integration (CI)**: Tools like CruiseControl automated the process of building and testing software after every change, catching integration bugs early.

2000s-2010s: Scaling and Shifting Quality Left

Paradigm Shifts

- **Shift-Left Testing**: The idea of testing earlier in the development lifecycle, involving QA and writing tests during development, not after.
- **Test Automation Pyramid**: A strategy for a balanced test suite: many fast, cheap Unit tests; fewer Integration tests; even fewer UI tests.
- **DevOps and Quality**: With rapid deployments, automated testing became non-negotiable. Quality is everyone's responsibility.

The Test Automation Pyramid.

New Frontiers

- **Selenium**: Automated web browser

2020s: The AI-Powered Future of Debugging

- **AI-Assisted Testing**:
  - AI generates test cases, predicts flaky tests, and optimizes test suites.
  - Tools can automatically detect visual regressions in UI.
- **Intelligent Fault Localization**:
  - AI analyzes code, execution traces, and bug reports to suggest the most likely lines of code causing a failure.
  - **Example**: A tool like **Amazon CodeGuru** or **OpenAI's Debugger** can point directly to the suspicious code.
- **Automated Program Repair**:
  - LLMs can not only find bugs but also suggest and even generate fixes.
  - **Example**: GitHub Copilot Chat can explain a bug and propose a patch.
- **Observability**: The evolution beyond monitoring. Using logs, metrics, and traces (the three pillars) to understand the internal state of a system by its external outputs, making debugging in production much more effective.

The Evolution of Debugging at a Glance

Era	Primary Method	Testing Focus	Ke
1940s-50s	Physical Inspection	Ad-hoc	Th
1960s-70s	Print Statements	Manual, Late	Sy
1980s-90s	Interactive Debuggers (gdb)	Formalized Test Levels	Au
2000s-10s	CI/CD Integrated Tools	Shift-Left, Automation	Tes
2020s+	AI-Powered Analysis	AI-Generated Tests	Int

The Unchanging Goal

To move from reactive bug-fixing to proactive bug-prevention, and to reduce the time between discovering a failure and fixing it from months to minutes.

2000s: Agile and DevOps

- **Agile Manifesto** (2001)
- Emphasis on iterative development and customer collaboration
- **DevOps movement** (late 2000s)
- Bridging development and operations
- Continuous Integration/Continuous Deployment
- Infrastructure as Code

2010s: AIOps and Cloud Native

- **AIOps**: Applying AI to operations
  - Automated monitoring and incident response
  - Predictive analytics for system performance
  - Anomaly detection and root cause analysis
- **Cloud Native Development**
  - Microservices architecture
  - Containerization (Docker, Kubernetes)
  - Serverless computing

2020s: The LLM Revolution

- **Large Language Models transform software engineering**
- GitHub Copilot (2021) and similar tools
- AI-assisted code generation and review
- Automated testing and documentation
- Shift in developer roles and skills

LLM Impact on Software Engineering

- **Code Generation**: From autocomplete to entire function generation
- **Debugging Assistance**: AI-powered bug detection and fixes
- **Documentation**: Automated documentation generation
- **Code Review**: AI-assisted quality assurance
- **Testing**: Intelligent test case generation

Future Directions

- AI pair programmers becoming standard
- New programming paradigms emerging
- Focus shifting to prompt engineering and AI supervision

Software Engineering Evolution Timeline

Visual Timeline Here

(1968 NATO 1975 Man-Month 1984 Trusting Trust 1990s OSS/OOP 2000s Agile/DevOps 2010s Cloud 2020s LLM)

Looking Forward

- Continuous evolution from disciplined engineering to AI augmentation
- Increasing emphasis on automation and intelligence
- Software engineering becoming more accessible
- New challenges in ethics, security, and quality assurance
- The human element remains crucial in system design and oversight

The journey continues...

Outline

The Seeds of an Idea

- **Ancient Myths**: Stories of artificial beings endowed with intelligence (e.g., Talos, Golems).
- **Philosophical Foundations**: Descartes' mind-body problem, Hobbes' view of reasoning as calculation.
- **Ada Lovelace (1843)**: Questioned whether machines could originate ideas, or only do what we command.
- **Early 20th Century**: Karel apek's play "R.U.R." (1920) coins the term "robot".

1950s: The Foundational Decade

- **Alan Turing (1950)**: Publishes "Computing Machinery and Intelligence", proposing the **Turing Test**.
- **The Name is Coined**: John McCarthy organizes the **Dartmouth Conference** (1956) and coins the term "Artificial Intelligence".
- **Early Optimism**: Founders predicted a machine as intelligent as a human within a generation.
- **Key Early Programs**:
  - Logic Theorist (Newell & Simon, 1956)
  - Samuel's Checkers Player (1959)

Diagram of the Turing Test

The Goal Was Set

1960s-70s: Symbolic AI & Limits

- **Symbolic AI (GOFAI)**: Dominant paradigm. Focused on manipulating symbols and logical rules to represent knowledge and solve problems.
- **Successes**:
  - **ELIZA** (1966): An early natural language processing program that simulated a Rogerian psychotherapist.
  - **SHRDLU** (1970): A program that could understand natural language commands in a "blocks world".
  - Expert systems began development.
- **The Limits**: Symbolic AI struggled with the complexity and ambiguity of the real world. It required massive, hand-coded knowledge bases.
- **The Lighthill Report (1973)**: Criticized the failure of AI to meet its grand promises, leading to a sharp cut in funding - the First AI Winter.

1980s: The Rise and Fall of Expert Systems

Expert Systems

- Commercial applications of AI.
- Aimed to capture the knowledge of human experts in rule-based systems.
- **Success Story**: MYCIN system for diagnosing blood infections performed as well as experts.
- Led to a wave of commercial investment.

Architecture of an Expert System

The Second AI Winter

- Expert systems were brittle, expensive to maintain, and could not learn.
- They failed to scale beyond narrow domains.
- By the late 1980s, the hype cycle

1990s-2000s: A New Paradigm

- **Shift from Logic to Statistics**: Instead of programming logical rules, researchers focused on creating systems that could **learn from data**.
- **The Rise of Machine Learning**:
  - **Support Vector Machines (SVMs)** and other statistical models became powerful tools.
  - **The Web** provided vast amounts of data for training.
- **Practical Successes**:
  - **Spam Filters** using Naive Bayes classifiers.
  - **Recommendation Systems** (e.g., Amazon, Netflix).
  - **Search Engines** (e.g., Google's PageRank algorithm).
- **IBM Deep Blue (1997)**: Defeated world chess champion Garry Kasparov, a landmark symbolic achievement powered largely by search and evaluation functions.

2010s: Deep Learning Unleashed

The Breakthrough

- **Key Enablers**: Massive datasets (Big Data) and powerful parallel processing (GPUs).
- **ImageNet Competition (2012)**: A deep neural network (AlexNet) drastically reduced error rates, shocking the research community. This was the "big bang" moment for modern AI.
- **The Technique**: **Deep Learning** uses neural networks with many layers to automatically learn hierarchical features from raw data.

ImageNet Error Rates Plummet with Deep Learning

Landmark Achievements

- **AlphaGo (2016)**: Defeated the world

2020s: The LLM and Generative AI Era

- **Scale is All You Need**: The discovery that vastly scaling up neural networks (size, data, compute) leads to emergent abilities.
- **The Transformer Architecture (2017)**: Became the foundational model for almost all modern AI, enabling parallel processing of sequences (like text).
- **Large Language Models (LLMs)**:
  - Models like GPT-3, GPT-4, Claude, Llama are trained on most of the public internet.
  - Capable of **generating human-like text**, translating, summarizing, and coding.
- **Generative AI**: LLMs power tools like ChatGPT, which bring AI capabilities to the general public.
- **Multimodal AI**: Models that can understand and generate across different modalities (text, images, audio).

AI History: A Journey of Waves

The AI field has progressed through waves of optimism followed by "winters" of reduced funding, each wave building on the last.

The Pattern

Symbolic AI (Rules) $\rightarrow$ Statistical ML (Data) $\rightarrow$ Deep Learning (Representations) $\rightarrow$ Generative AI (Creation)

Looking Forward

- **Artificial General Intelligence (AGI)**: The original, elusive goal of human-level intelligence remains on the horizon.
- **Ethics and Safety**: As AI becomes more powerful, concerns about bias, control, and alignment with human values become paramount.
- **AI as a Commodity**: AI capabilities are being integrated into almost every software product.
- **The Symbiosis**: The future likely involves a close collaboration between human and artificial intelligence, augmenting human capabilities.

The dream of thinking machines is older than computing itself.
The journey is far from over.

Outline

Search-Based Software Engineering

Search-based software engineering (SBSE) applies metaheuristic search techniques such as genetic algorithms, simulated annealing and tabu search to software engineering problems. Many activities in software engineering can be stated as optimization problems. Optimization techniques of operations research such as linear programming or dynamic programming are often impractical for large scale software engineering problems because of their computational complexity or their assumptions on the problem structure. Researchers and practitioners use metaheuristic search techniques, which impose little assumptions on the problem structure, to find near-optimal or "good-enough" solutions[1].

[1] https://en.wikipedia.org/wiki/Search-based_software_engineering

Mining Software Repositories

Within software engineering, the mining software repositories (MSR) field analyzes the rich data available in software repositories, such as version control repositories, mailing list archives, bug tracking systems, issue tracking systems, etc. to uncover interesting and actionable information about software systems, projects and software engineering[2].

[2] https://en.wikipedia.org/wiki/Mining_software_repositories

Empirical Software Engineering

Empirical software engineering (ESE) is a subfield of software engineering (SE) research that uses empirical research methods to study and evaluate an SE phenomenon of interest. The phenomenon may refer to software development tools/technology, practices, processes, policies, or other human and organizational aspects[3]. Common research methods used in ESE for primary and secondary research are the following:

1. Primary research (experimentation, case study research, survey research, simulations in particular software Process simulation)
2. Secondary research methods (Systematic reviews, Systematic mapping studies, rapid reviews, tertiary review)

[3] https://en.wikipedia.org/wiki/Empirical_software_engineering

References

[1] NATO Software Engineering Conference (1968)

[2] Dijkstra, E. W. (1968). "Go To Statement Considered Harmful"

[3] Brooks, F. P. (1975). *The Mythical Man-Month*. Addison-Wesley.

[4] Thompson, K. (1984). "Reflections on Trusting Trust"

[5] Stallman, R. (1983). GNU Manifesto

[6] Beck, K. (2001). Agile Manifesto

[7] Chen, M. et al. (2023). "Evaluating Large Language Models Trained on Code"