Intelligent Software Engineering

When Software Engineering Meets Artificial Intelligence

Zhilei Ren

Dalian University of Technology

September 28, 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”

What Will be Discussed

Contents

- Introduction to Software Engineering and Artificial Intelligence
- Requirements Engineering
- Design and Implementation
- Software Testing
- A Case Study of Software Engineering Research

Outline

Pioneering Thoughts

Who is the first programmer?

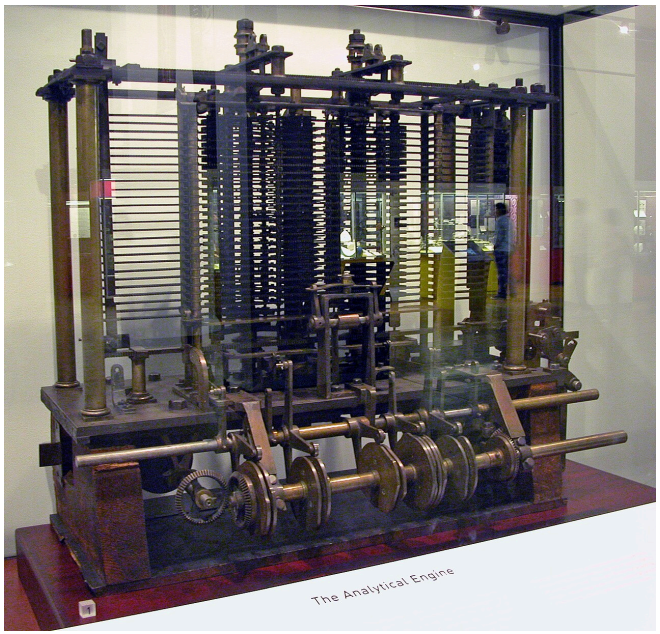
1843: Ada Lovelace - The First Programmer

- **Augusta Ada King, Countess of Lovelace** (1815-1852)
- Daughter of the poet Lord Byron, trained in mathematics

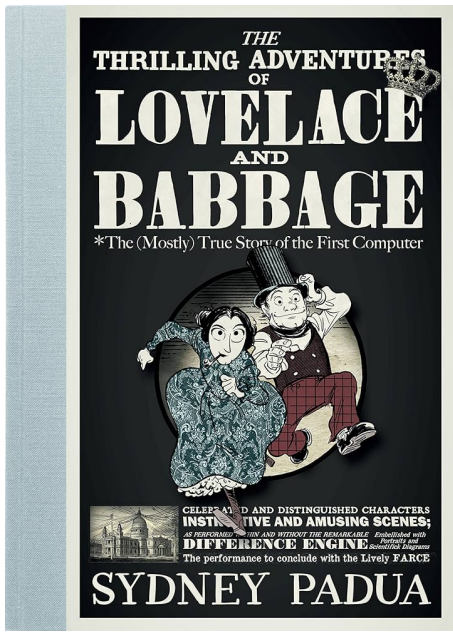
Figure 1: Portrait of Ada Lovelace

1843: Ada Lovelace - The First Programmer

- Collaborated with Charles Babbage on his **Analytical Engine**
- The Thrilling Adventures of Lovelace and Babbage



(a) Trial Model of Analytical Machine



(b) Cover of the Comic

Figure 2: Analytical Machine

1843: Ada Lovelace - The First Programmer

- She translated an Italian article on the engine, adding her own extensive **Notes**
- **Note G** contained what is considered the first computer program

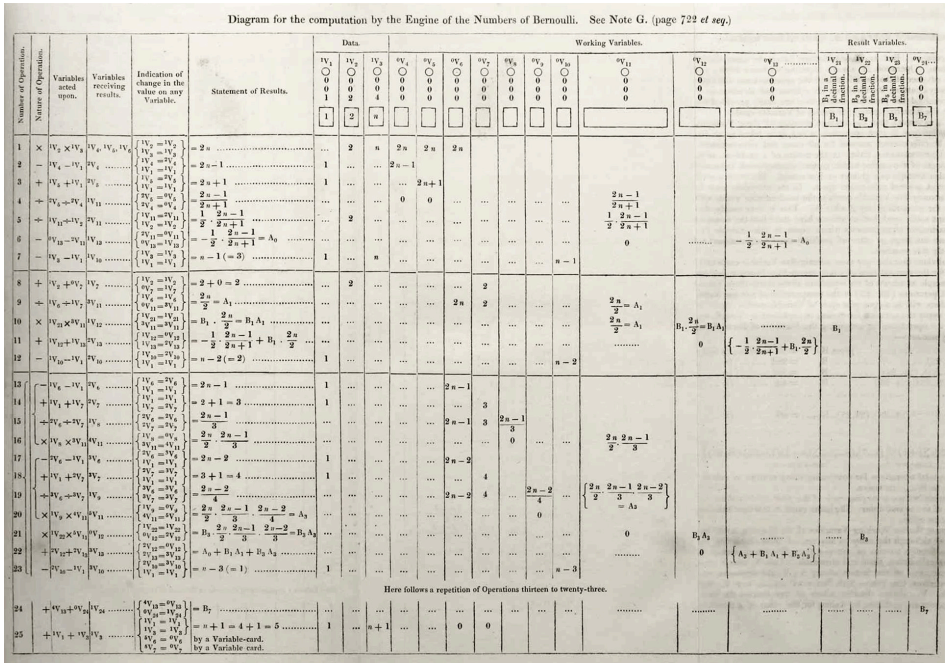


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

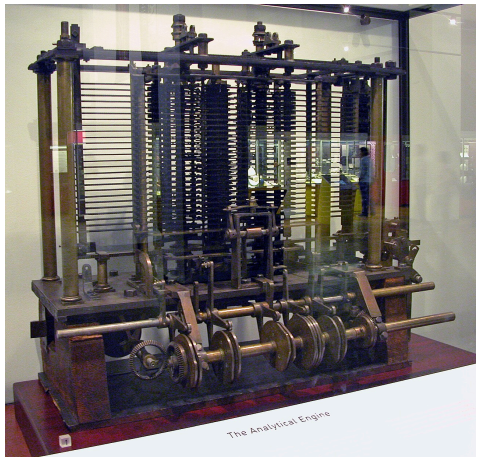
Figure 3: The First Program

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 second Tuesday of October.

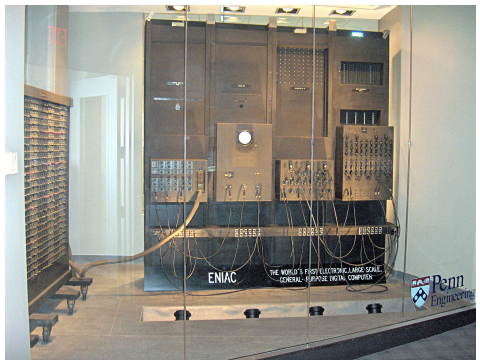
Figure 4: The Ada Programming Language Mascot

From Vision to Reality



1840s Lovelace's Theoretical Program

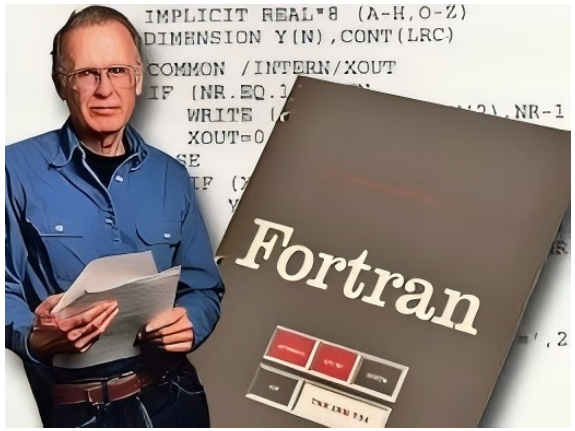
⇓ **100 Years** ⇓

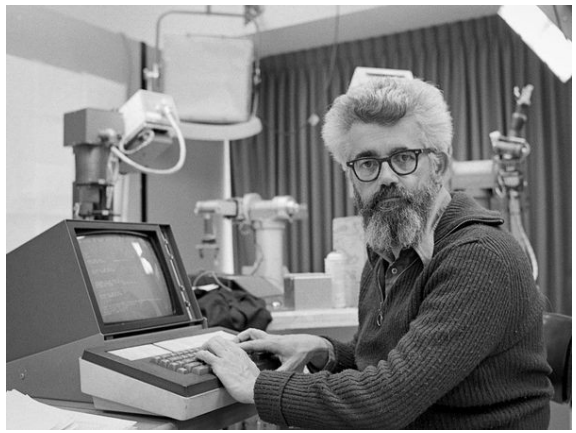


1940s ENIAC - First Electronic Computer

The Dawn of High-Level Programming

- **1950s Context**: Programming in machine code/assembly was tedious and error-prone
- **The Vision**: Create languages that were more human-readable and machine-independent
- **Two Paths Emerged**:
  - **Fortran (1957)**: Designed for scientific computing - *imperative*
  - **Lisp (1958)**: Designed for AI research - *functional*





Left: John Backus, Fortran creator; Right: John McCarthy, Lisp creator

Fortran: The First High-Level Language

- **Name**: FORmula TRANslating system
- **Created**: 1957 by John Backus at IBM
- **Primary Goal**: Make scientific computing easier and more efficient
- **Key Innovations**:
  - First compiler ever created
  - Mathematical notation similar to algebra
  - Array operations and complex numbers
  - Surprisingly efficient code generation
- **Impact**: Revolutionized scientific and engineering computing

Fortran Code Example

Listing 1: Fortran 77 Style

```
C AREA OF A TRIANGLE WITH HERON'S FORMULA
      PROGRAM TRIANGLE
      REAL A, B, C, AREA, S

      READ *, A, B, C
      S = (A + B + C) / 2.0
      AREA = SQRT(S * (S - A) * (S - B) * (S - C))

      PRINT *, 'AREA = ', AREA
      END PROGRAM
```

- **Imperative style**: Explicit sequence of operations
- **Mathematical focus**: Direct translation of formulas
- **Simple data flow**: Variables are modified in place
- **Fixed format**: Early versions required specific column positions

Lisp: The Second High-Level Language

- **Name**: LISt Processing
- **Created**: 1958 by John McCarthy at MIT
- **Primary Goal**: Artificial Intelligence research
- **Key Innovations**:
  - Functional programming paradigm
  - Linked list as fundamental data structure
  - Garbage collection
  - Homoiconicity: Code is data, data is code
  - Recursion as primary control structure
- **Impact**: Foundation for AI research and functional programming

Lisp Code Example

Listing 2: Common Lisp Style

```
;; Calculate factorial using recursion
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))

;; Map a function over a list
(defun square-list (numbers)
  (mapcar #'(lambda (x) (* x x)) numbers))

;; Using higher-order functions
(let ((numbers '(1 2 3 4 5)))
  (print (factorial 5))
  (print (square-list numbers)))
```

- **Functional style**: Emphasizes expressions over statements
- **Recursive thinking**: Problems broken down recursively
- **List manipulation**: Built-in operations for list processing

Paradigm Comparison: Imperative vs Functional

- **Fortran (Imperative Paradigm)**:
  - *Philosophy*: "How to compute" - sequence of steps
  - *State changes*: Variables are modified
  - *Control structures*: Loops, conditionals, goto
  - *Primary abstraction*: Procedures and subroutines
  - *Memory management*: Manual/stack-based
- **Lisp (Functional Paradigm)**:
  - *Philosophy*: "What to compute" - expressions and transformations
  - *Immutability*: Avoid changing state when possible
  - *Control structures*: Recursion, function composition
  - *Primary abstraction*: Functions as first-class citizens
  - *Memory management*: Automatic garbage collection

Design Philosophy Comparison

- **Fortran's Practical Focus**:
  - Designed for numerical computation efficiency
  - Close mapping to mathematical notation
  - Optimized for array operations and linear algebra
  - Minimal syntax, maximum computational power
  - Target user: Scientists and engineers
- **Lisp's Theoretical Foundation**:
  - Based on lambda calculus and recursive function theory
  - Embraced symbolic computation and AI problems
  - Powerful metaprogramming through macros
  - Interactive development with REPL (Read-Eval-Print Loop)
  - Target user: AI researchers and computer scientists

The Dawn of Software Engineering

- **1968 NATO Conference**: The term "Software Engineering"
- Addressing the "software crisis" - growing complexity and cost of software development
- Key goal: Apply engineering principles to software development

Figure 5: The 1968 NATO Garmisch Conference

1968: "Goto Statement Considered Harmful"

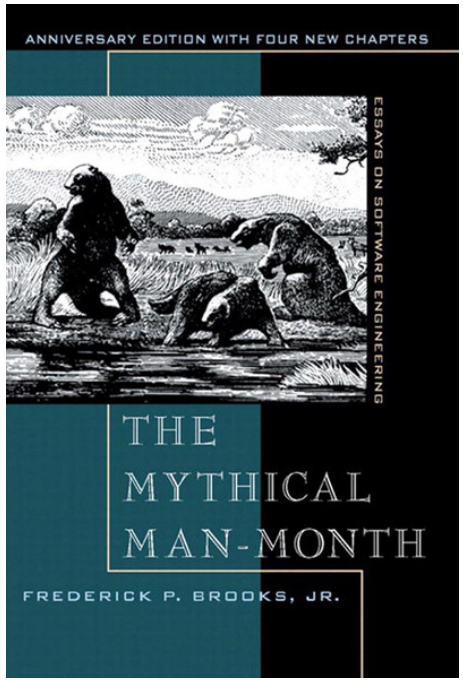
- **Edsger Dijkstra's seminal letter** (1968)
- Advocated against use of GOTO statements
- Promoted structured programming principles
- Foundation for modern control structures (if-else, while, for)
- Emphasized code readability and maintainability

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.
- "No Silver Bullet"



1984: Reflections on Trusting Trust

- **Ken Thompson's Turing Award Lecture** (1984)
- A compiler could insert backdoors that propagate themselves
- Fundamental insight into software supply chain security

Figure 6: Ken Thompson and Dennis Ritchie with UNIX

Key Takeaway

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

1984: Reflections on Trusting Trust

- **Ken Thompson's Turing Award Lecture** (1984)
- A compiler could insert backdoors that propagate themselves
- Fundamental insight into software supply chain security

Figure 6: Ken Thompson and Dennis Ritchie with UNIX

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)

1980s-1990s: GNU and Open Source Revolution

- Linux kernel created by Linus Torvalds (1991)
- Open Source Initiative founded (1998)

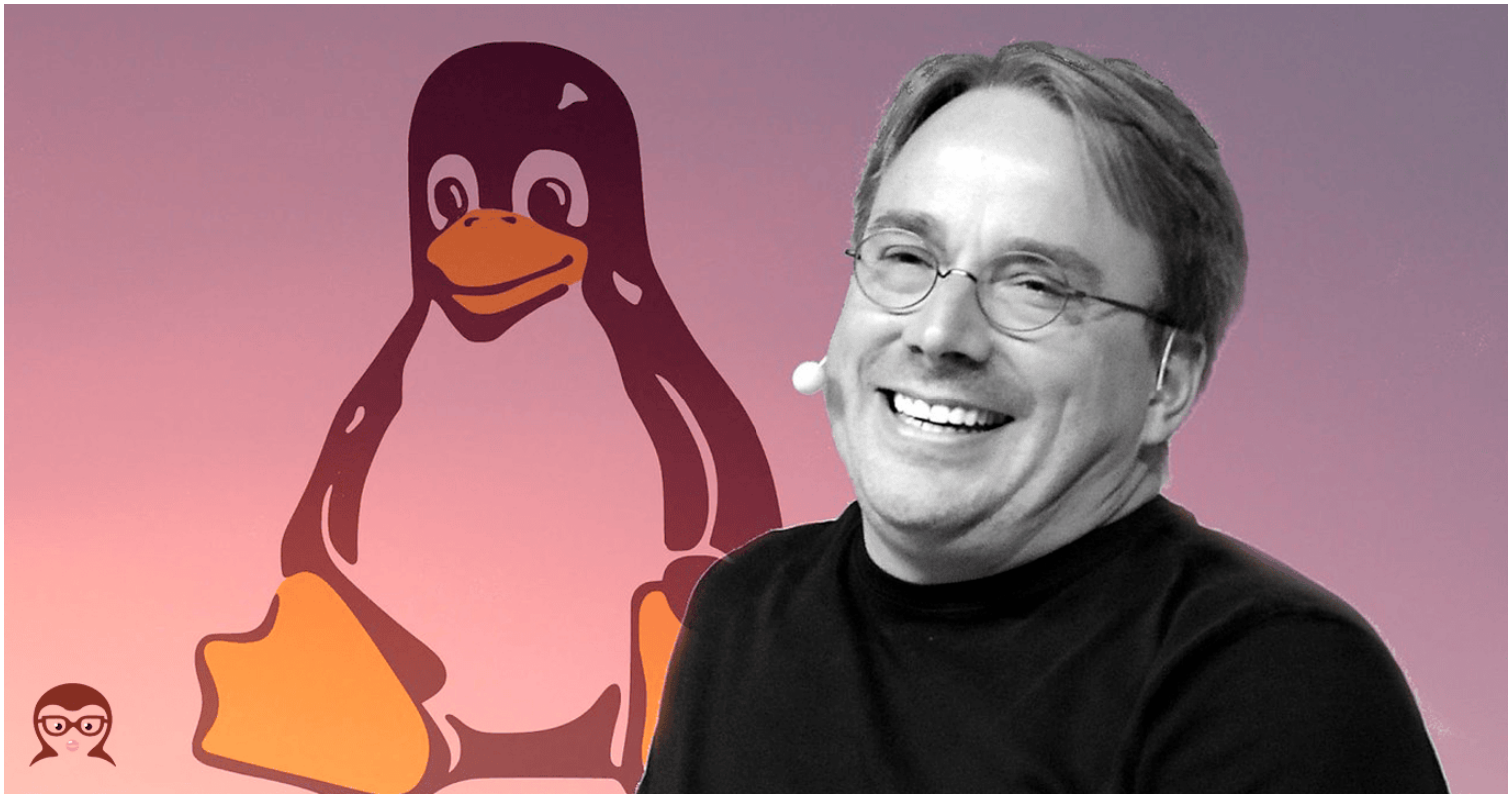


Figure 8: Linus Torvalds and Linux

From Netscape to Firefox

The Fall of a Giant: Netscape's "Cathedral"

- **Netscape Navigator** dominated the early web with 90% market share in the mid-1990s.
- Lost the "first browser war" to Microsoft's Internet Explorer (bundled with Windows).
- Facing irrelevance, Netscape took a radical step in **1998**.

From Netscape to Firefox

The Radical Shift: Embracing the "Bazaar"

- In January 1998, Netscape **open-sourced** its browser code (Netscape Communicator 4.0) to harness the power of distributed developers.
- The **Mozilla Organization** was created to manage the project.
- The old code was unwieldy; developers started from scratch with the new **Gecko** rendering engine.

From Netscape to Firefox

The Phoenix Rises: Birth of Firefox

- From the Mozilla codebase, a lightweight, user-focused browser emerged named **Phoenix** (2002).
- After trademark disputes (Phoenix $\rightarrow$ Firebird $\rightarrow$ **Firefox**), Mozilla Firefox 1.0 launched in **2004**.
- Firefox challenged IE's monopoly, championing open standards and user choice.

The Philosophical Engine: Cathedral vs. Bazaar

Cathedral Model

- **Closed, centralized development**.
- Built by a small, dedicated team of experts.
- Releases are infrequent, grand events.
- **Examples**: Pre-1998 Netscape, traditional software development.

Bazaar Model

- **Open, decentralized development**.
- Code is developed publicly over the Internet.
- "Given enough eyeballs, all bugs are shallow."
- **Examples**: Mozilla, Linux, Firefox.

The Core Lesson

The Netscape-to-Firefox story shows that a well-managed **Bazaar** model can build complex, high-quality systems (a "cathedral") more effectively than a closed model by leveraging global collaboration.

Legacy and Lasting Impact

- **Technical Foundations**: The Gecko engine paved the way for Firefox. Netscape's legacy also includes JavaScript, SSL, and Cookies, invented by its engineers.
- **Institutionalization**: The **Mozilla Foundation** (est. July 15, 2003) ensures the project's mission continues beyond any single company.
- **A Lasting Battle**: Firefox became the leading alternative to IE, proving the viability of open-source software for mainstream users and setting the stage for the modern web.

The open-source bazaar, born from a cathedral's ashes, helped ensure the web remained open and competitive.

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)

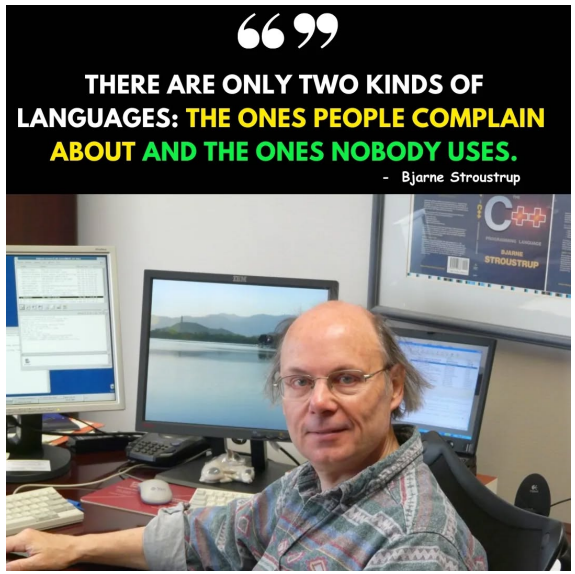


Figure 9: Quotes of the Creator of C++

2000s: Agile and DevOps

- **Agile Manifesto** (2001): Emphasis on iterative development and customer collaboration
- **DevOps movement** (late 2000s)
- Bridging development and operations, featuring Git and CI/CD
- Infrastructure as Code

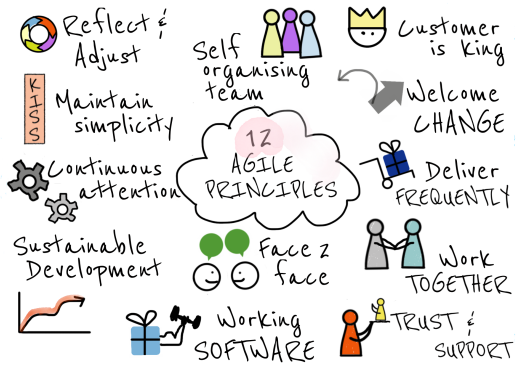


Figure 10: the 12 Agile Principles

1990s: CVS - Concurrent Versions System

- **Developed in 1986**, became dominant in 1990s
- **Centralized Architecture**: Single central repository
- **Key Features**:
  - Basic version tracking
  - Multiple developer support
  - Network-based access
- **Limitations**:
  - No atomic commits
  - Poor branching/merging
  - Slow network operations
  - Single point of failure

CVS brought version control to the masses, but collaboration was sequential and cautious.

2000: Subversion - "CVS Done Right"

- **Created in 2000** by CollabNet to fix CVS limitations
- **Improvements over CVS**:
  - **Atomic commits**: All changes succeed or fail together
  - **Better branching/tagging**: Cheap copy operations
  - **Versioned directories and renames**
  - **Better binary file handling**
- **Still centralized**: Single repository model
- **Adoption**: Quickly became the enterprise standard
- **Limitations**: Central server = single point of failure, offline work difficult

2002-2005: The BitKeeper Controversy

- **Linux kernel development** outgrew patch-based workflow
- **BitKeeper**: Proprietary distributed version control system
- **Adopted by Linux in 2002**: Enabled true distributed development
- **Revolutionary features**:
  - Distributed repositories
  - Excellent branching/merging
  - Fast performance
- **Controversy (2005)**: License disputes led to BitKeeper withdrawal
- **Catalyst**: Forced Linux community to create their own solution

2005: Git - The Linux Community's Answer

- **Created by Linus Torvalds in 2005** after BitKeeper controversy
- **Design goals**:
  - Distributed and peer-to-peer
  - Strong cryptographic integrity
  - Fast branching/merging
  - Efficient with large projects
  - Support for non-linear development
- **Key innovations**:
  - Content-addressable storage (SHA-1 hashes)
  - Directed acyclic graph (DAG) for history
  - Three-stage thinking (working directory, staging area, repository)

Why Git Won: Distributed Workflows

Centralized (CVS/SVN)

- Single source of truth
- Sequential commits
- Server required
- Branching is expensive
- Linear history

Distributed (Git)

- Every clone is a full repository
- Parallel development
- Work offline, sync later
- Cheap branching
- Non-linear history

The Game Changer

Git enabled workflows like **feature branching** and **pull requests**, making open-source collaboration scalable.

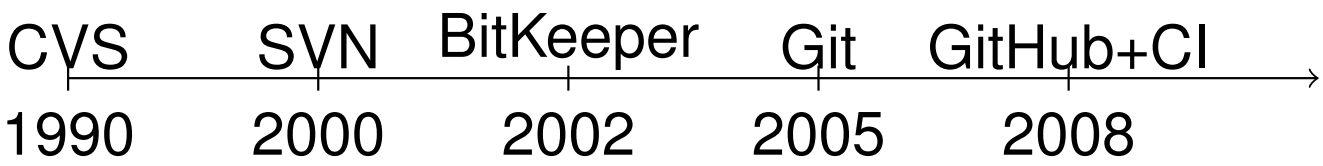
GitHub and the Social Coding Revolution (2008)

- **GitHub launched 2008**: Added social layer to Git
- **Key innovations**:
  - Pull Requests: Code review as first-class citizen
  - Fork & Pull model: Easy contribution to any project
  - Issues, Projects, Wikis: Integrated project management
- **Network effects**: Made open-source collaboration mainstream
- **Enterprise adoption**: GitHub Enterprise, GitLab, Bitbucket

Git-Enabled CI/CD: The Automation Revolution

- **Continuous Integration (CI)**: Automatically build and test every commit
- **Continuous Deployment/Delivery (CD)**: Automatically deploy passing code
- **Git as the trigger**:
  - Push to repository triggers automated pipeline
  - Branch protection rules enforce quality gates
  - Pull request status checks prevent bugs
- **Key tools**: Jenkins, GitHub Actions, GitLab CI, CircleCI

The Evolution Timeline



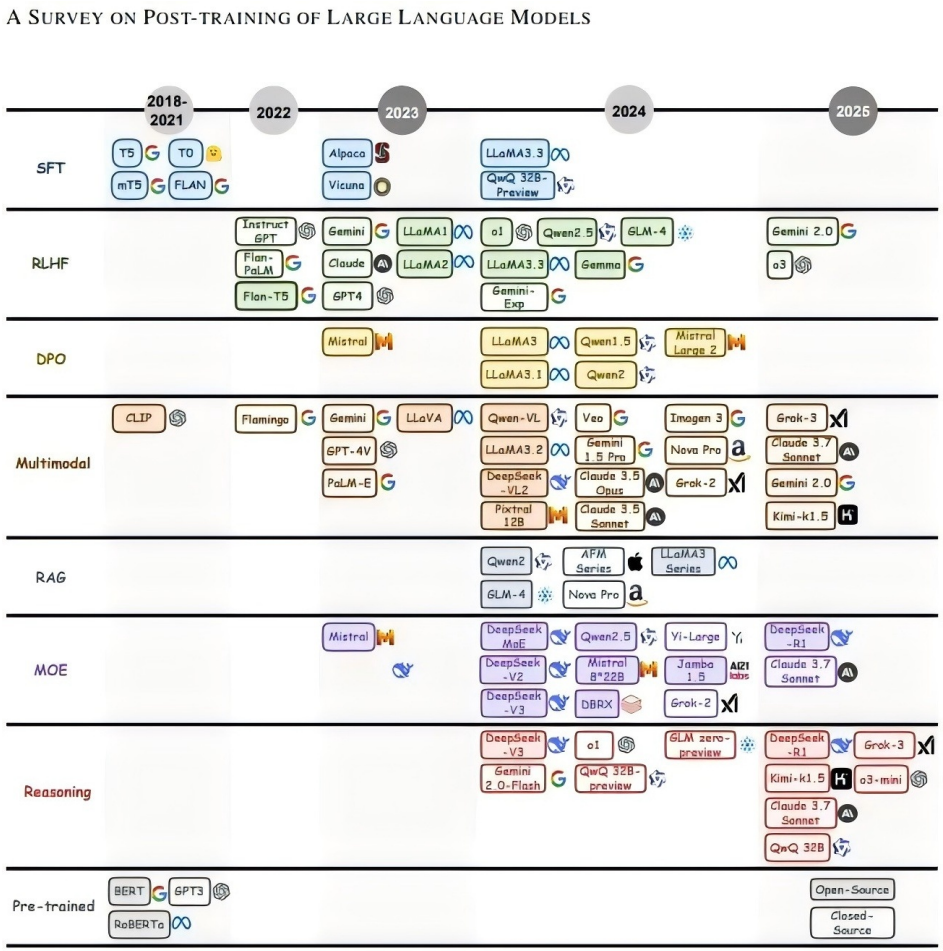
- **1990s**: Centralized era - sequential collaboration
- **2000s**: Centralized improvement - better but still limited
- **2002-2005**: Distributed awakening - BitKeeper shows the way
- **2005+**: Git revolution - parallel, distributed collaboration
- **2008+**: Social coding + automation - CI/CD ecosystem

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



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

Looking Forward

- Continuous evolution from disciplined engineering to AI augmentation
- Increasing emphasis on automation and intelligence
- The human element remains crucial in system design and oversight
- AI is powerful, but it is neither a silver bullet nor an elixir

The journey continues...

Outline

The Seeds of an Idea

- **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".

Figure 12: Capek's play R.U.R

1950s: The Foundational Decade

- **Alan Turing (1950)**: Publishes "Computing Machinery and Intelligence", proposing the **Turing Test**.

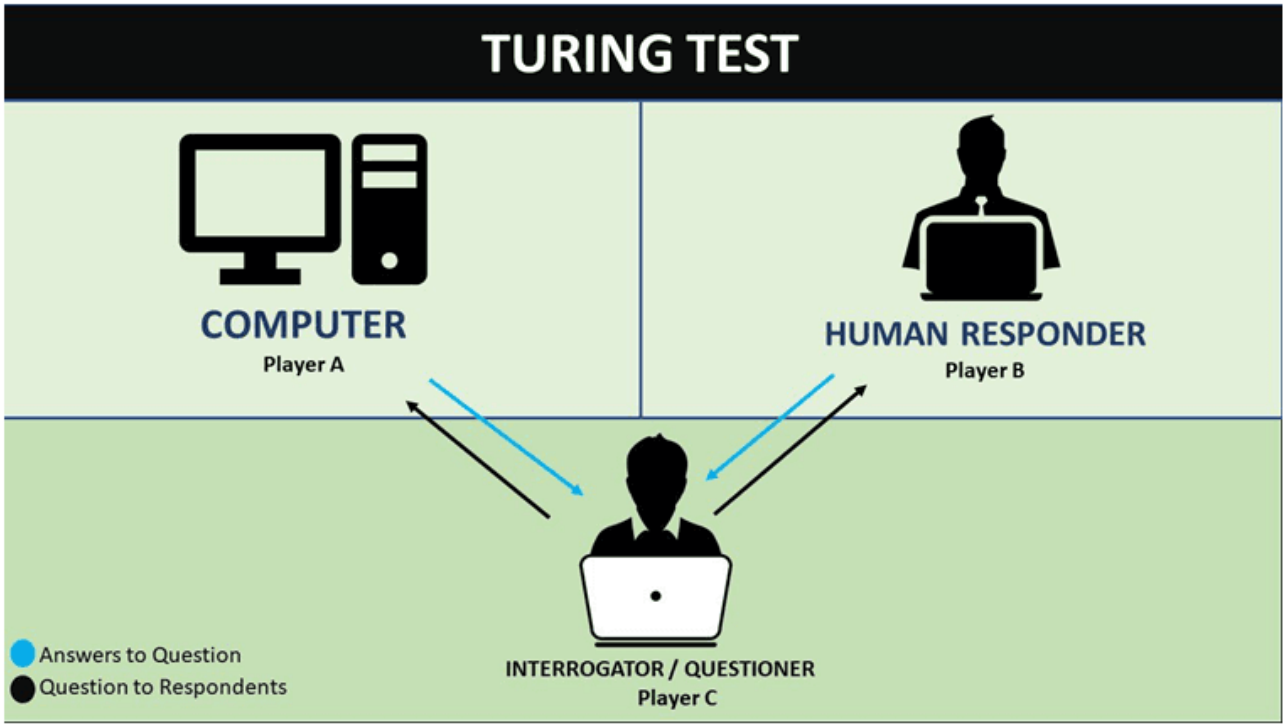


Figure 13: Diagram of the Turing Test

1950s: The Foundational Decade

- **The Name is Coined**: John McCarthy organizes the **Dartmouth Conference** (1956) and coins the term "Artificial Intelligence".

Figure 14: Dartmouth Conference

1960s-70s: Symbolic AI & Limits

- **Symbolic AI (GOFAI)**: Focused on manipulating symbols and logical rules to represent knowledge and solve problems.
- **ELIZA** (1966): An early natural language processing program that simulated a Rogerian psychotherapist.
- **The Lighthill Report (1973)**: Criticized AI's failure to meet its grand promises, leading to sharp cut in funding - the First AI Winter.

Figure 15: ELIZA

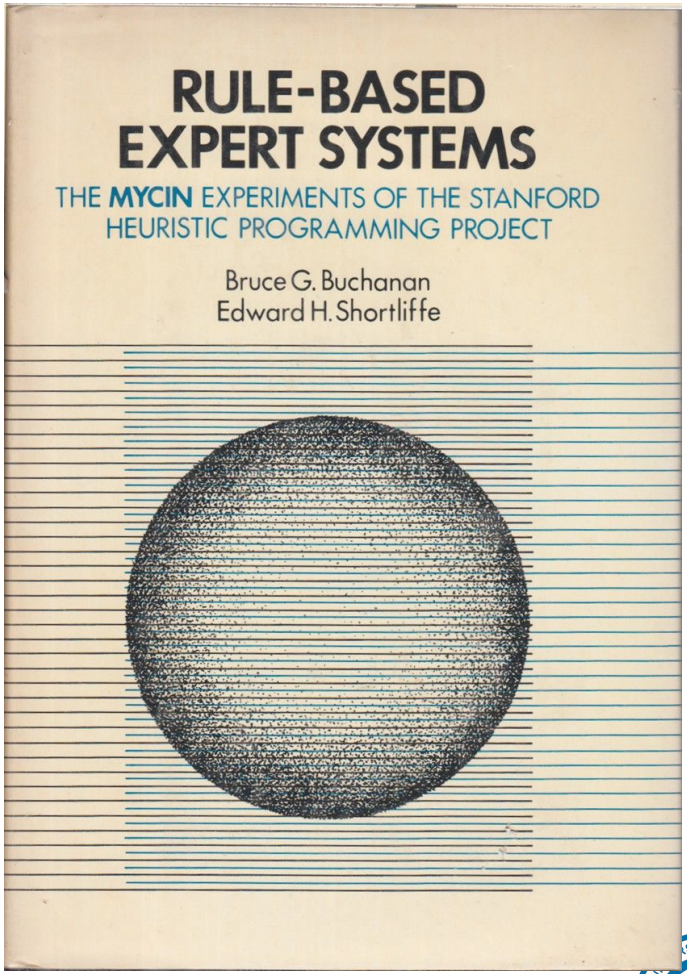
1980s: The Rise and Fall of Expert Systems

Expert Systems

- Aimed to capture the knowledge of human experts in rule-based systems.
- MYCIN system for diagnosing blood infections performed as well as experts.

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 crashed again, leading to the **Second AI Winter**.



Rule-Based Expert System

1990s-2000s: A New Paradigm

- **Shift from Logic to Statistics**: Instead of programming logical rules, researchers focused on creating systems (e.g., SVM, Random Forests) that could **learn from data**.
- **IBM Deep Blue (1997)**: Defeated world chess champion Garry Kasparov, a landmark symbolic achievement powered largely by search and evaluation functions.

Figure 16: Deep Blue vs. Garry Kasparov

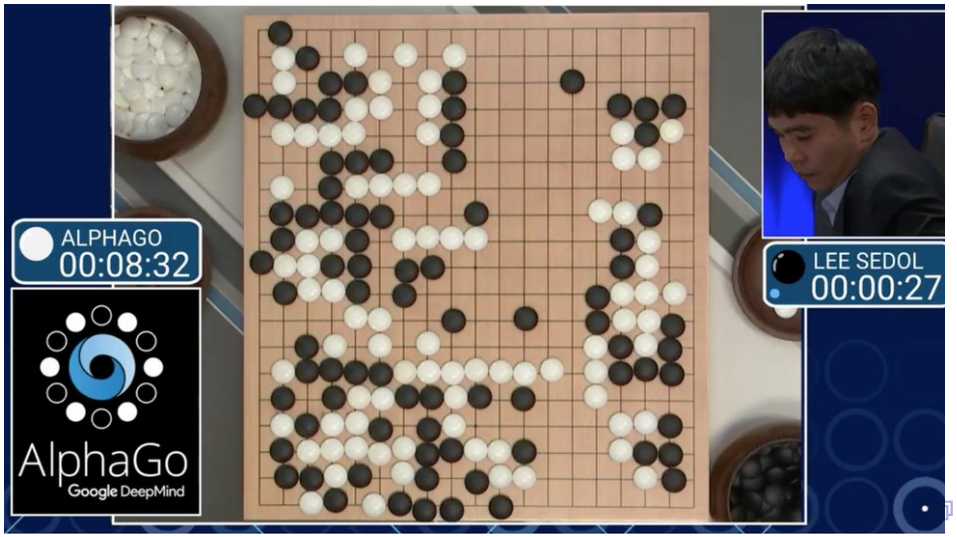
2010s: Deep Learning Unleashed

The Breakthrough

- **Key Enablers**: Massive datasets (Big Data) and powerful parallel processing (GPUs).
- **The Technique**: **Deep Learning** uses neural networks with many layers to automatically learn hierarchical features from raw data.

Landmark Achievements

- **AlphaGo (2016)**: Defeated the world champion in Go, a game considered far more complex than chess.



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

Many activities in software engineering can be stated as optimization problems. 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].

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"