What Is Engineering?

Chapter 2 of Modern Software Engineering

The Bridge Building Analogy

Common Misconception: "Software Isn't Bridge Building"

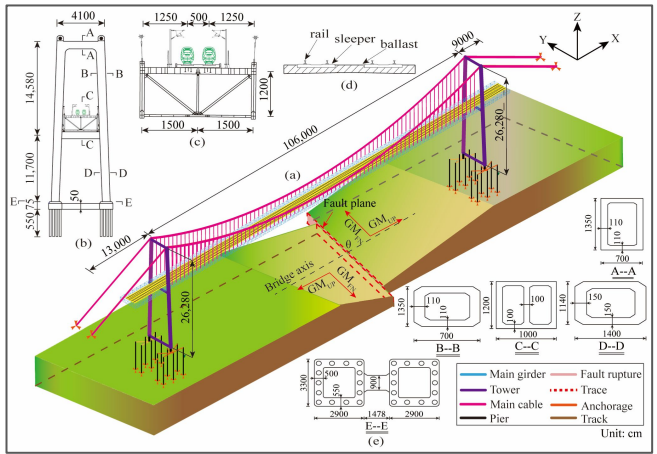
- This phrase is often stated to emphasize differences between software development and traditional engineering.
- However, it reveals a misunderstanding about the nature of engineering in both contexts.

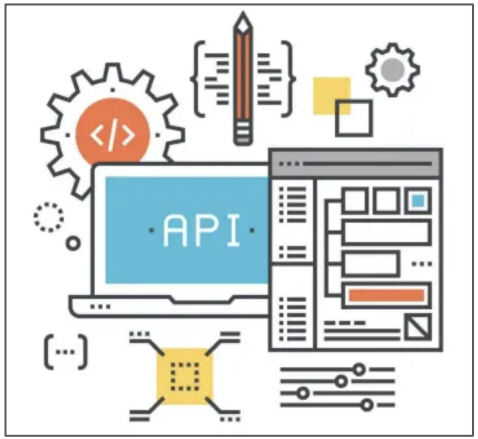
The Bridge Building Analogy

Production Engineering vs. Design Engineering

- **Production Engineering** (e.g., bridge building):
  - Involves creating physical objects with precise specifications.
  - Requires managing constraints like time, location, budget, and material precision.
  - Must adapt designs to real-world conditions to meet safety and quality standards.

- **Design Engineering** (e.g., software engineering):
  - Focuses on creating and iterating digital models.
  - Lacks physical constraints—no material measurements or deliveries.
    - The cost of production is minimal; digital assets can be replicated and modified easily.





Production Is Not Our Problem in SWE=

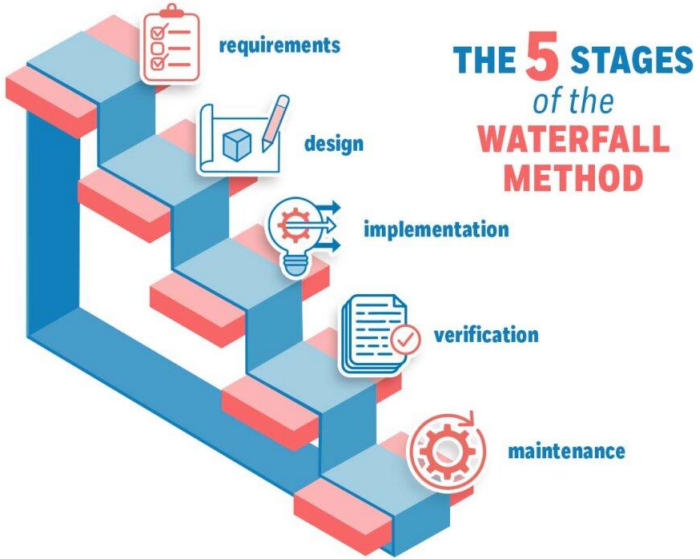
Production vs. Design in Traditional Engineering

- In traditional fields like automotive or aerospace, **production** is the hardest part:
  - Moving from prototype to mass production is costly and complex.
  - Efficiency in production requires significant effort and industrial infrastructure.

Production Is Not Our Problem in SWE

Industrial Age Thinking and Its Influence

- **Industrial age mindset** emphasizes production efficiency and scalability.
- This mindset has influenced software development, leading to methodologies like **Waterfall**:
    - **Waterfall Model**: A linear, sequential approach to software development.
    - Focuses on defined phases and handovers, similar to an assembly line.
    - Treats software development as a production process rather than a creative, iterative endeavor.



Production Is Not Our Problem in SWE

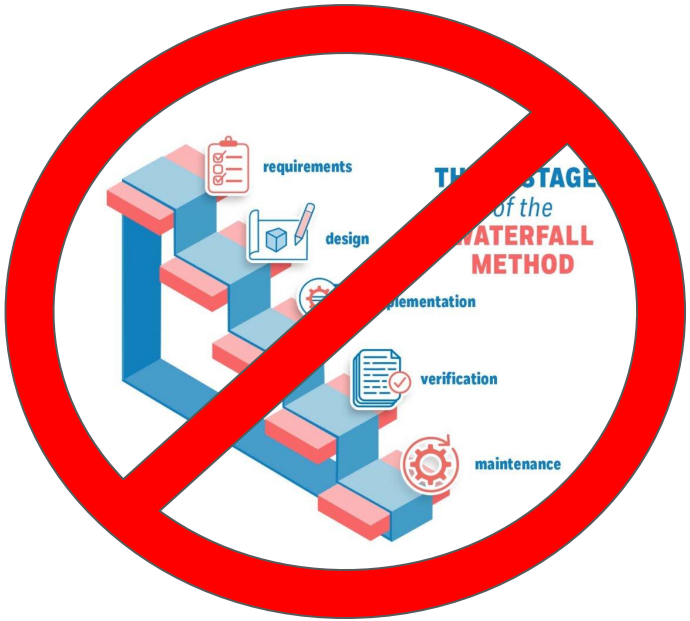
Software Development: A Different Paradigm

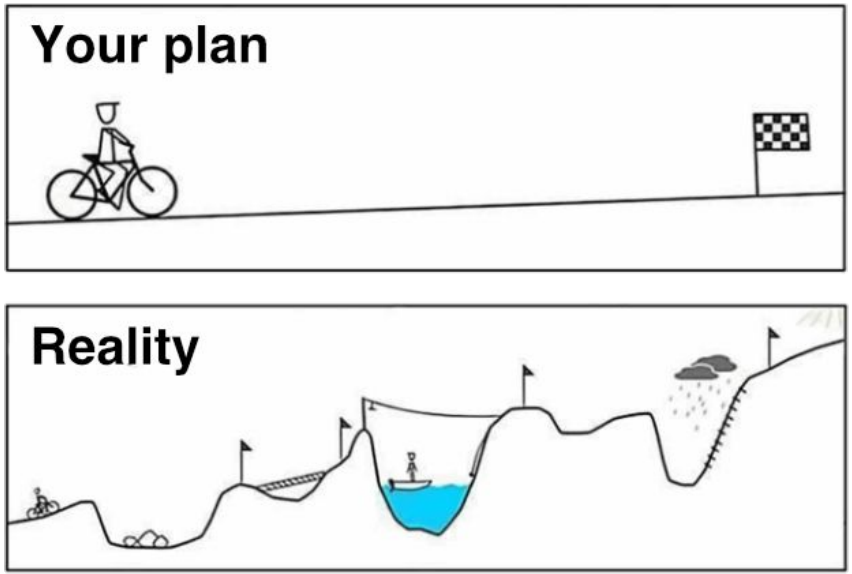
- Unlike traditional engineering, software development doesn't face the same production challenges:
  - **Software production** is automated, scalable, and inexpensive.
  - Building software is often as simple as pressing a button.
  - Errors in production are easily addressed with tools and technology.

Production Is Not Our Problem in SWE

Key Insights

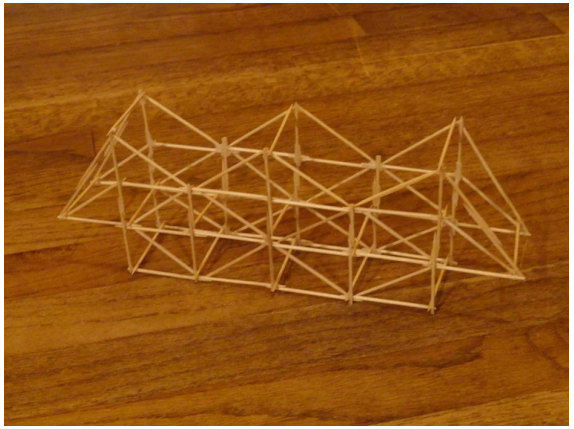
- **Misapplied Thinking**: Applying production-centric methods to software can hinder discovery and innovation.
- **Focus on Discovery & Learning**: Software development should prioritize experimentation & adaptability over rigid production processes.
- Understanding that **production is not our problem** helps distinguish software engineering from traditional production-focused disciplines.





Design Engineering, Not Production Engineering 1:3

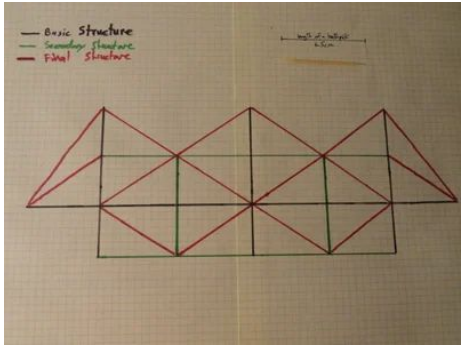
Bridge Building vs. Software Development

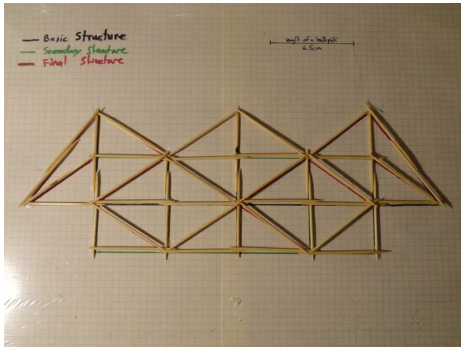


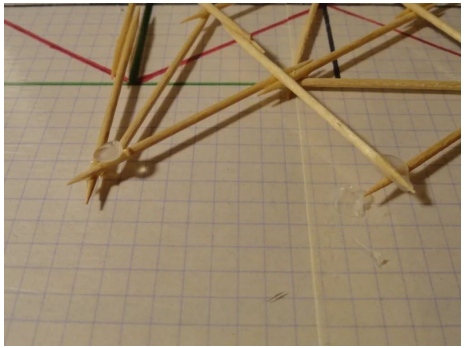
- **First of a New Kind of Bridge**: Involves two key challenges:

    1. **Production Problems** (irrelevant to software):
        - Physical production constraints: materials, precision, and scalability.
        - These challenges are not typically faced in software development.

    2. **Design Challenges** (relevant to software):
        - Innovative design requires experimentation and iteration.
            - Difficult with physical objects due to high cost and time constraints.





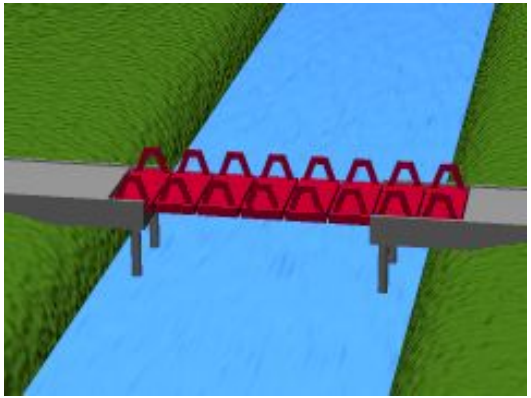


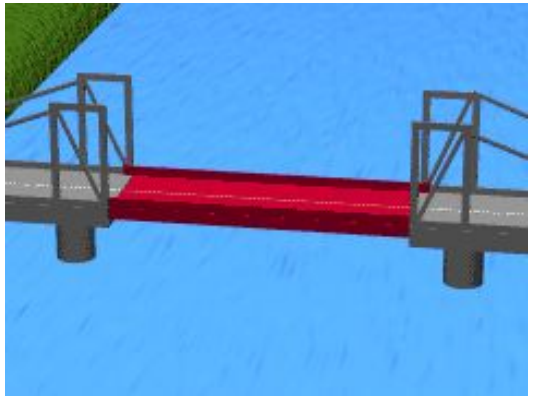
https://www.instructables.com/Toothpick-Bridge/

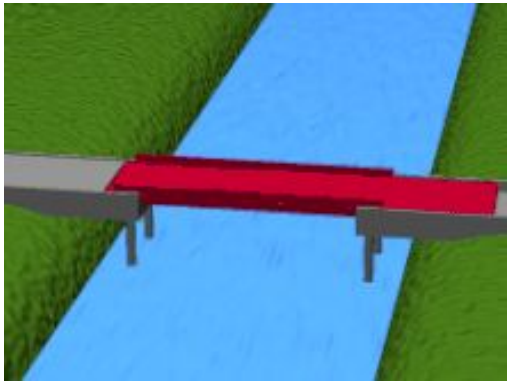
Design Engineering, Not Production Engineering 2:3

Advantages of Software Design

- **Quick Iteration and Adaptability**:
  - Software can be changed and refined rapidly without the constraints of physical materials.
  - Unlike physical models, software simulations are directly the product, allowing immediate validation.
- **Flexibility in Modeling**:
  - In software, our models are our reality, not an approximation.
  - Changes in software are inexpensive and quick, facilitating continuous improvement.





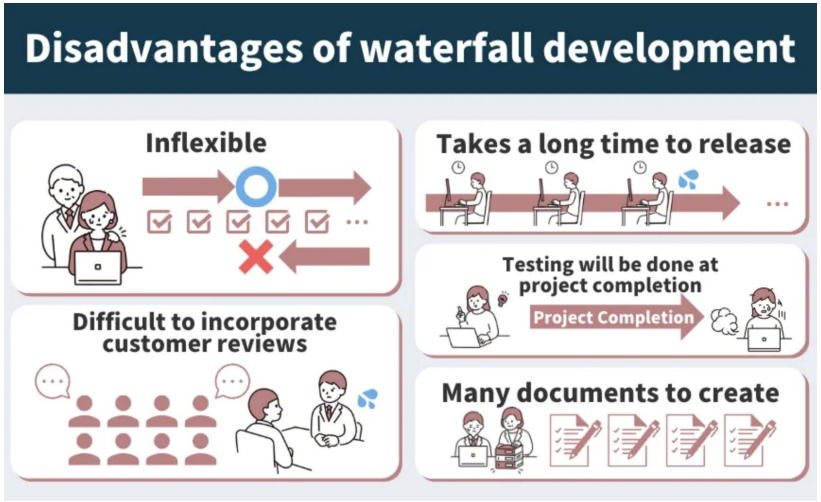


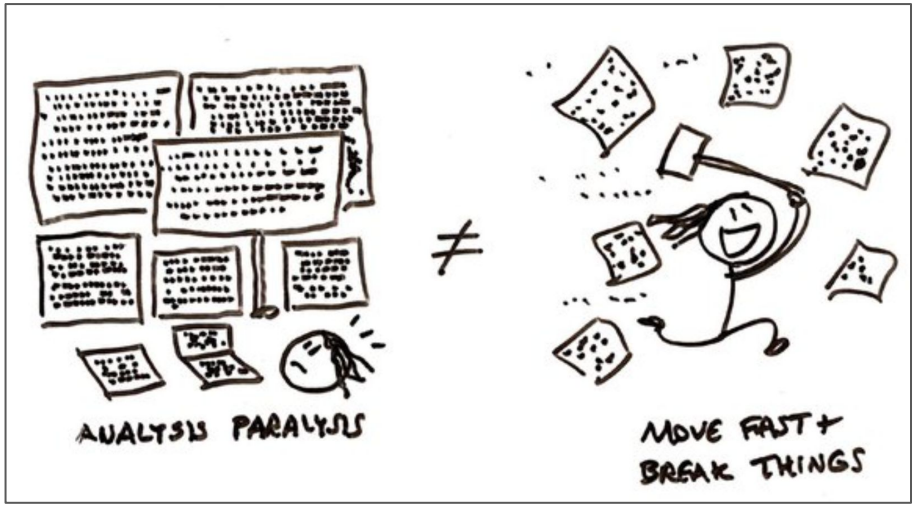
Design Engineering, Not Production Engineering 3:3

Challenges with Scientific Rationalism in Software

- **Missteps in History**:
  - Software development has often deviated from scientific principles, focusing too much on production-like processes.

- **Misconceptions about Precision**:
  - Pursuit of mathematical precision in software can lead to unrealistic expectations.
  - Engineering is about practical application, not idealistic precision.





Engineering as Math in Software Development

Historical Context: The Rise of Formal Methods

- **Late 1980s - Early 1990s**: Increased focus on programming structures and effective problem-solving in software engineering.
- **Formal Methods**: Approaches to software development that integrate mathematical validation, aiming to prove code correctness.
- **Challenge**: Formal methods made code production harder, especially for complex systems, leading to limited adoption.

Mathematical Specification and Formal Methods

Problem Statement:

Given a sorted array A of length n and a target value x, find the index i such that $A[i] = x$, or return -1 if x is not in the array.

Mathematical Specification:

1. **Pre-condition (Assumption):**
   The array A is sorted in non-decreasing order.

$$\forall j \in [0, n-2],\ A[j] \leq A[j+1]$$

2. **Post-condition (Guarantee):**
   If x is in A, the algorithm returns an index i such that $A[i] = x$. If x is not in A, the algorithm returns -1.

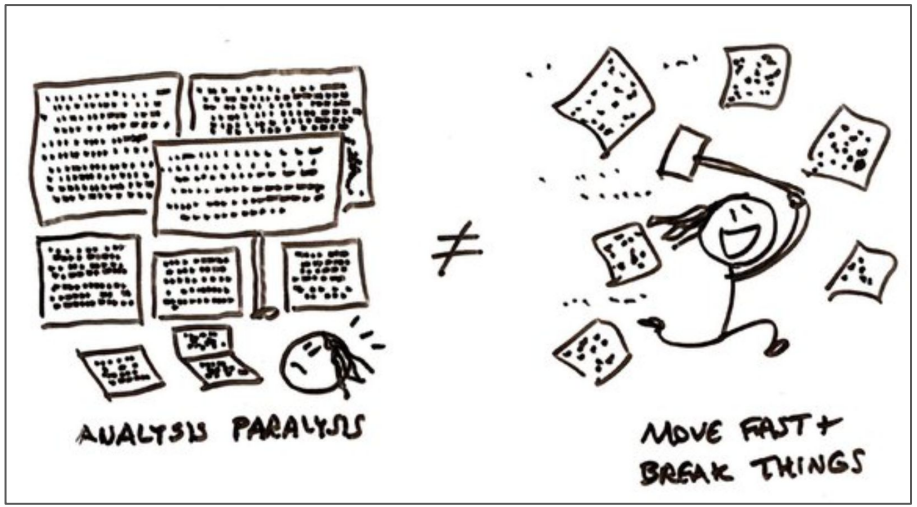
$$(x \in A \implies \exists i \in [0, n-1], A[i] = x) \land (x \notin A \implies i = -1)$$

```python
def binary_search(A, x):
    left, right = 0, len(A) - 1
    while left <= right:
        mid = (left + right) // 2
        if A[mid] == x:
            return mid
        elif A[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Engineering as Math in Software Development

Appeal of Mathematical Approaches in Software

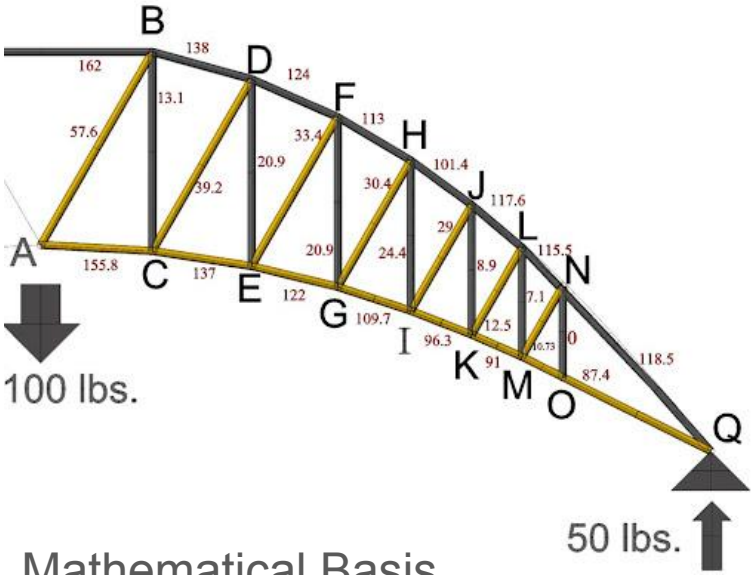
- **Natural Fit**: Many software engineers are drawn to mathematical thinking, making formal methods initially attractive.
- **Limitations**: Over-reliance on mathematics can constrain creativity and practicality in software development.



Engineering as Math in Software Development

Real-World Analogy: Engineering Disciplines and Mathematics

- **Role of Math**: In other engineering fields, math is a critical tool but not a replacement for empirical testing.
- **Testing and Empirical Learning**: Engineers use math for predictions but validate their designs through real-world tests.
- **Example**: Aerospace engineers extensively use math but still build prototypes to test and refine their designs.



Mathematical Basis

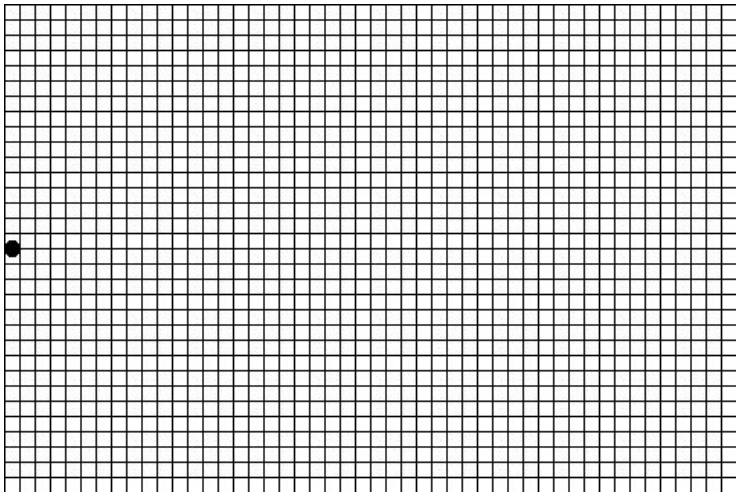
Empirical Testing

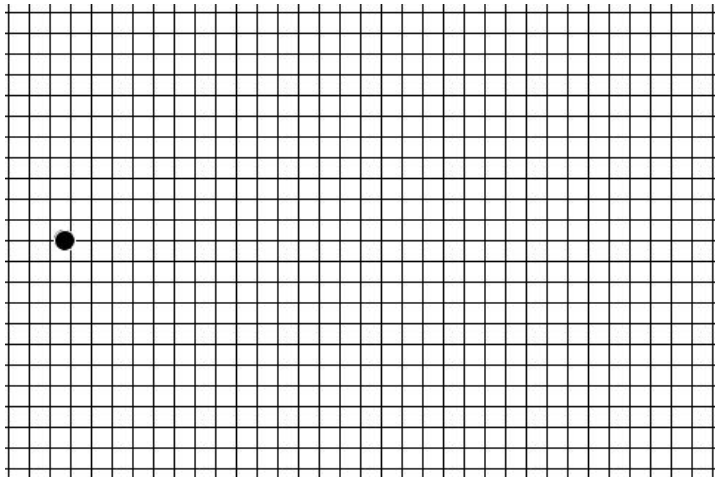
Engineering as Math in Software Development

Software vs. Physical Systems

- **Deterministic Nature of Software**:
  - Software runs on deterministic devices (computers), making mathematical validation possible in some narrow contexts.

- **Challenges with Complexity**:
  - In systems involving concurrency, real-world interactions, or complex domains, formal methods become impractical.
  - The variability in real-world software systems often makes mathematical proofs unfeasible.





Engineering as Math in Software Development

Pragmatic Approach in Software Engineering

- **Hybrid Strategy**: Like other engineers, software developers use mathematical thinking but also adopt empirical, data-driven approaches.
- **Incremental Learning**: Emphasize experimentation and adaptation, building systems iteratively based on real-world feedback and testing.
- **Focus on Practical Outcomes**: Aim to create software that works well in practice, balancing mathematical rigor with flexibility and responsiveness.

Engineering in SW vs. Other Disciplines: SpaceX 1:3

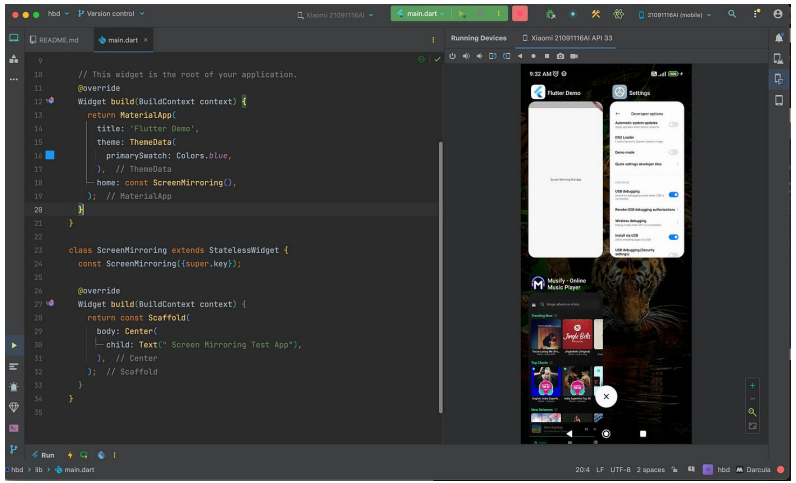
SpaceX's Iterative Engineering Approach

- SpaceX tests and iterates on its designs even after thorough mathematical modeling.
- **Example:** Switching from 4mm to 3mm stainless steel was tested through destructive pressure tests, despite having extensive data and models.
- This approach validates models and reveals unexpected behaviors, demonstrating the importance of testing in physical engineering.

Engineering in SW vs. Other Disciplines: SpaceX

Software Engineering: A Unique Discipline

- **Direct Testing of Products**: In software, models and the final product are the same—when we test software, we test the actual system, not a prototype or approximation.
- **High Fidelity in Testing**: Software tests can replicate the exact conditions of production, allowing precise evaluation and real-world accuracy.
- **Discovery and Learning**: Unlike physical engineering, software development focuses on exploration, learning, and continuous improvement rather than production efficiency.



Engineering in SW vs. Other Disciplines: SpaceX 3:3

Engineering Means “Stuff That Works”

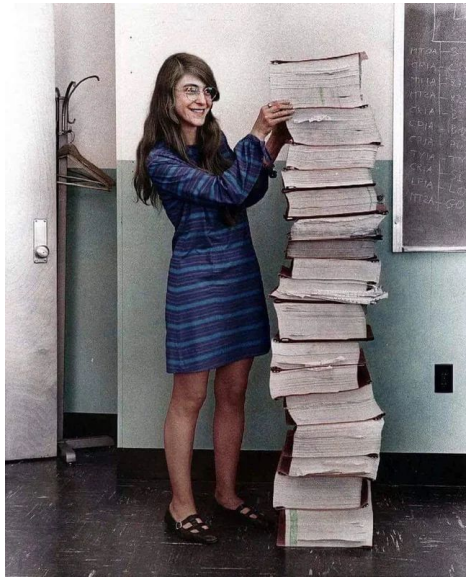
- **Glenn Vanderburg’s Insight**:
  - Critiques traditional academic software engineering as too rigid and ineffective in real-world scenarios.
  - Effective engineering should focus on practical outcomes, not just theoretical correctness or adherence to cumbersome processes.
- **Value in Software Engineering**:
  - The true test of software engineering practices is whether they help us create better software more efficiently.
  - Engineering practices that do not contribute to this goal are not true engineering.

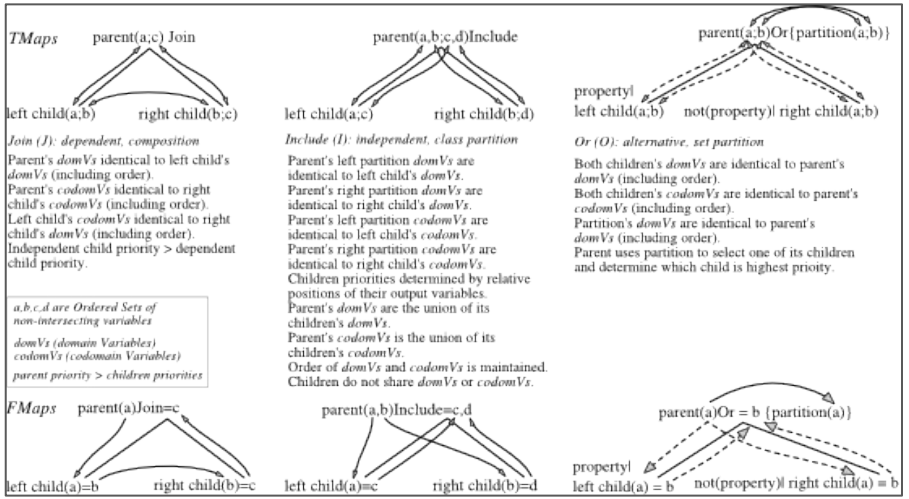
"The way things actually get built is to start with something small... it's an iterative, incremental experimental process – an empirical process – rather than a defined step-by-step linear process, like the rational model." -- Glenn Vanderburg

The First Software Engineer: Margaret Hamilton

Pioneering a New Field

- **Margaret Hamilton** led the development of the Apollo flight control systems during the 1960s, a time when there were no established software engineering practices.
- The team created their own **software engineering rules** as they encountered new challenges, balancing freedom with evolving NASA regulations.





The First Software Engineer: Margaret Hamilton

Challenges and Innovations

- **High Stakes**: Apollo software had to be extremely reliable, capable of detecting and recovering from errors in real time.
- **Lack of Guidance**: The complexity and novelty of the project meant there were no prior experiences or resources to draw upon.
- **Focus on Errors**: Hamilton's approach emphasized understanding and preventing errors, a fascination that led to rigorous testing and anticipation of failures.
- **Engineering Empiricism**: Solutions were treated with skepticism until thoroughly tested against potential failures, embodying a scientific, empirical approach.

The First Software Engineer: Margaret Hamilton

Establishing Software Engineering

- **Software's Reputation**: At the time, software was viewed as a "poor relation" compared to other engineering disciplines.
- **Coining the Term**: Hamilton coined "software engineering" to elevate the perception of software development as a serious engineering discipline.
- **Failing Safely**: A core principle introduced was "failing safely," ensuring systems could handle unexpected scenarios and continue functioning.

The First Software Engineer: Margaret Hamilton 4:4

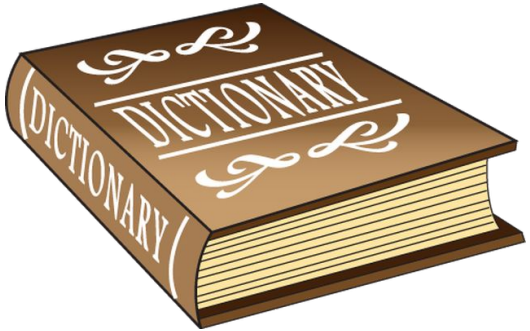
Case Study: Apollo 11 Lunar Landing

- **Unexpected Alarms**: During the descent, the Lunar Module experienced 1201 and 1202 alarms due to computer overload.
- **Fail-Safe Design**: The software was designed to reboot, prioritizing essential tasks & ignoring less critical ones, like erroneous radar jobs.
- **Successful Landing**: This design allowed the mission to continue safely, demonstrating the importance of anticipating and managing failures.

A Working Definition of Engineering

Common Dictionary Definitions

- Engineering is often defined with phrases like:
  - **Application of Math**: Using mathematical principles to solve problems.
  - **Empirical Evidence**: Relying on observations and experiments.
  - **Scientific Reasoning**: Applying logic and scientific methods.
  - **Within Economic Constraints**: Solutions must be cost-effective and feasible.



A Working Definition of Engineering

Proposed Working Definition of Engineering

- **Engineering** is the application of an **empirical, scientific approach** to finding **efficient, economic solutions** to practical problems.

Breaking Down the Definition

- **Application of an Empirical, Scientific Approach**:
  - **Empirical**: Based on observation and experimentation rather than theory alone.
  - **Scientific Approach**: Using systematic methods to investigate and solve problems.

- **Finding Efficient Solutions**:
  - Solutions are designed to maximize effectiveness and minimize waste of resources (time, money, materials).

- **Economic Solutions**:
  - Solutions must consider the cost and feasibility, ensuring they are viable within given economic constraints.

- **Practical Problems**:
  - Engineering focuses on real-world issues, creating tangible, actionable solutions that can be implemented effectively.

A Working Definition of Engineering

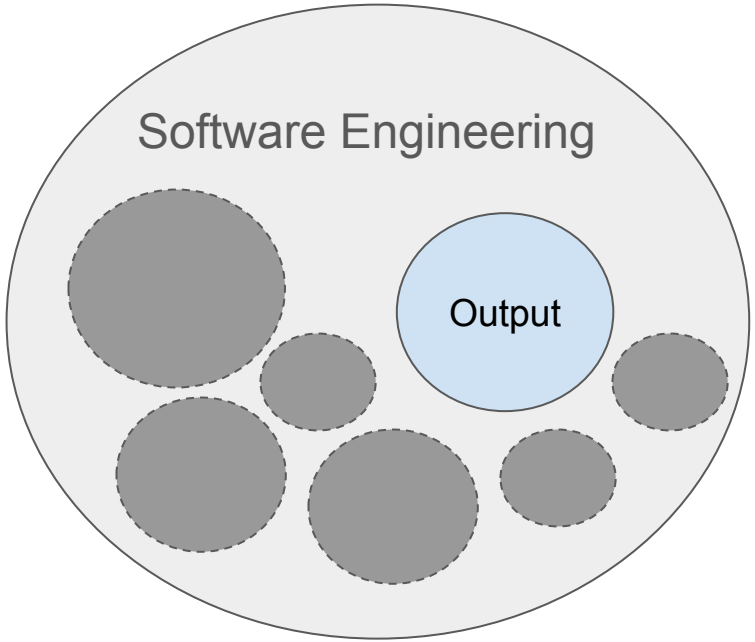
Key Characteristics of Engineering

- **Applied Science**: Practical use of scientific knowledge to solve real-world problems.
- **Pragmatic**: Focused on practical application rather than theoretical constructs.
- **Context-Sensitive**: Solutions are tailored to specific problems and environments.
- **Economically Viable**: Solutions are developed with an understanding of the economic limitations and goals.

Engineering ≠ Code

Common Misconception

- **Narrow View**: Many believe engineering in software development is solely about the **output**—the code or its design.
- **Broader Perspective Needed**: This view overlooks the comprehensive nature of engineering beyond just the end product.



Engineering ≠ Code

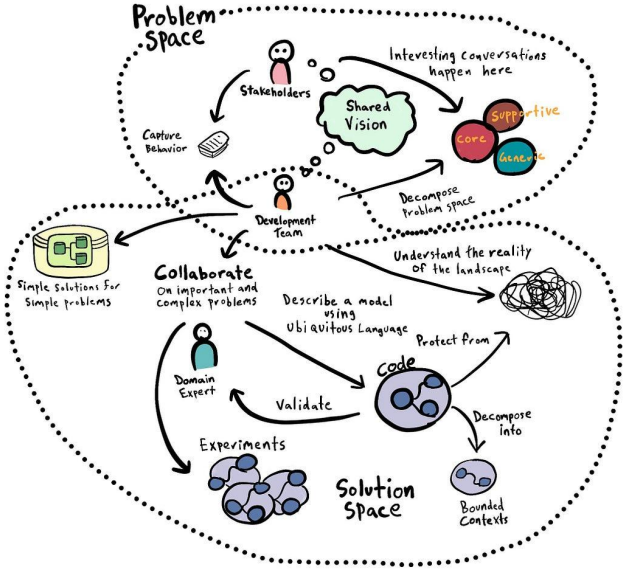
What Engineering Truly Means

- **Example from SpaceX**:
  - Engineering isn't just the rocket; it's the **process of creating** the rocket.
  - The rocket is a **product of engineering**, but the engineering itself involves the methods, tools, and processes used to build it.
  - Welding metal is part of engineering, but not the entirety of it—it’s the broader application of scientific principles to solve problems.

Engineering ≠ Code

Defining Engineering in Software

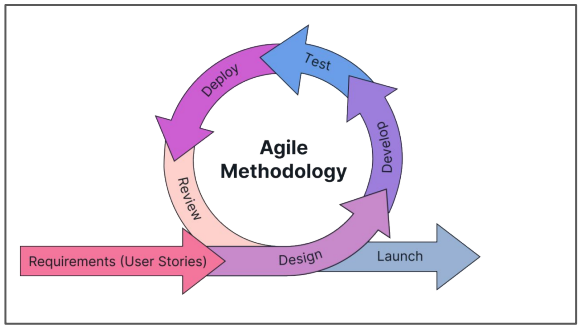
- **Not Just the Code**: Engineering is not limited to writing code or designing software architecture.
- **Holistic Approach**: Engineering encompasses the **processes, tools, techniques**, and **philosophy** used to develop software.
- **Scientific Rationalism**: Applying a rational, scientific approach to solving practical problems is the essence of engineering.



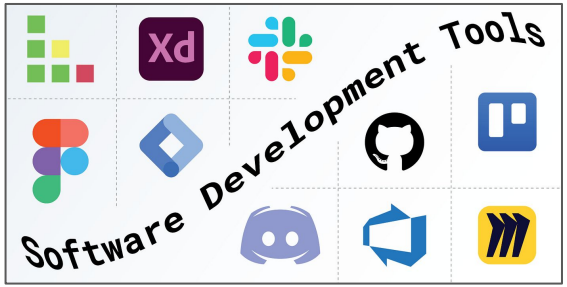
Engineering ≠ Code

Engineering as a Discipline

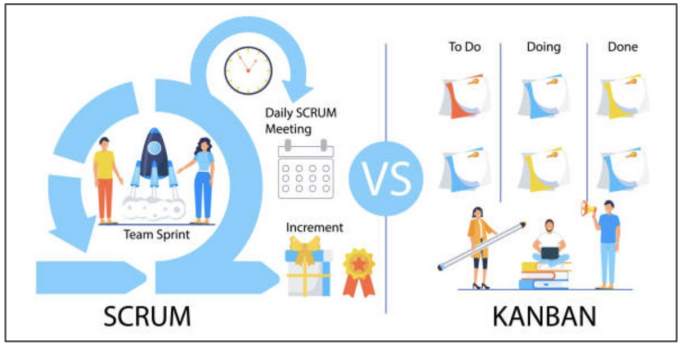
- **Problem-Solving Focus**: The real essence of engineering lies in the **solving of problems**, not just in the creation of the final product.
- **Components of Software Engineering**:
  - **Process**: The methodologies and frameworks used to guide software development.
  - **Tools**: The software, platforms, and utilities that aid in building and managing software projects.
  - **Culture**: The organizational environment and practices that influence how software is developed.



Process



Tools



Culture

Engineering ≠ Code

Case Study: Misinterpreted Failure → The Cyberpunk 2077 release was a disaster.

- **Example from a Game Failure (Cyberpunk 2077)**:
  - The failure wasn't just about bad coding—it was a failure in the **entire approach** to producing software.
  - Poor planning, a negative culture, and subpar coding practices all contributed to the failure.

- **Whole System Perspective**: Effective engineering requires considering all aspects that contribute to software development, not just the technical ones.

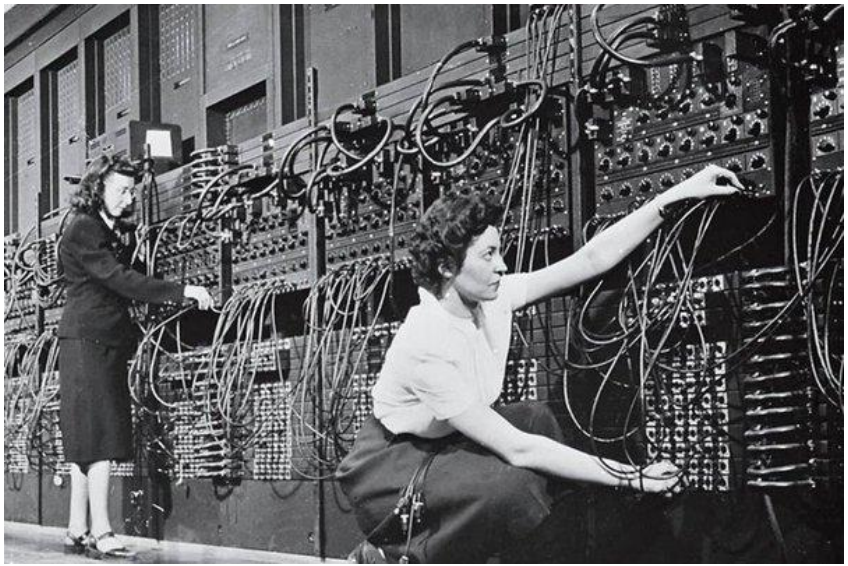
"You can't make a baby in one month with 9 women." -- David Farley

David Farley YouTube Video: https://www.youtube.com/watch?v=E-jGEtqB4wU

The Evolution of Programming Languages

Early Software Engineering Focus

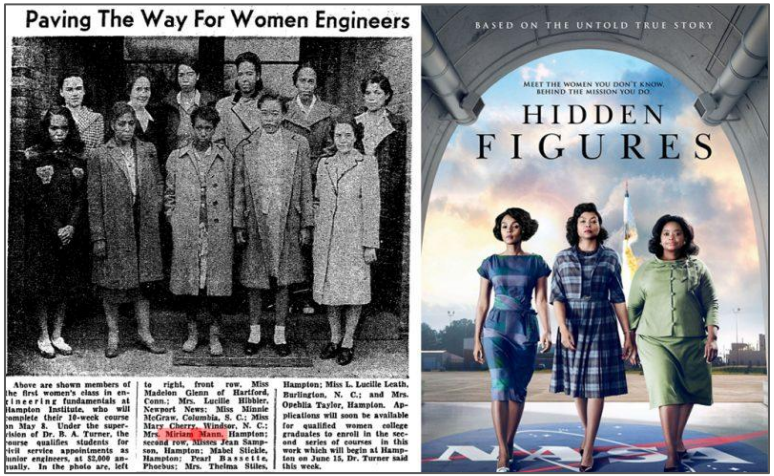
- Initially centered on creating **better languages** to program machines.
- **First Computers**: No clear separation between hardware and software; programming involved manually configuring hardware (e.g., plugging wires, flipping switches).



The Evolution of Programming Languages

Human “Computers” and Early Programming

- Early programmers were often **women** who previously performed calculations manually.
- These individuals translated mathematical problems into machine-specific settings, making them the first true programmers.
- **Terminology Evolution**:
  - What we now call **requirements** were instructions given to these programmers.
  - Their planning and problem-solving efforts were the earliest forms of **programming**.



The Evolution of Programming Languages

Advancements in Programming

- **Stored Programs**: Introduction of **paper tape** and **punched cards** allowed for programs to be stored and reused.
- **Machine Code**: Early programs were written in machine code, which was directly executed by the hardware.
- **High-Level Languages**: Provided a higher level of abstraction, enabling faster development and easier comprehension.

```
00011010 00000001
00100010 00000010
10100011 00000011
```

Machine Code (Binary)

```
MOV AX, 01     ; Move 1 into register AX
ADD AX, 02     ; Add 2 to the value in register AX
MOV BX, AX     ; Move the result from AX to BX
```

Assembly Language

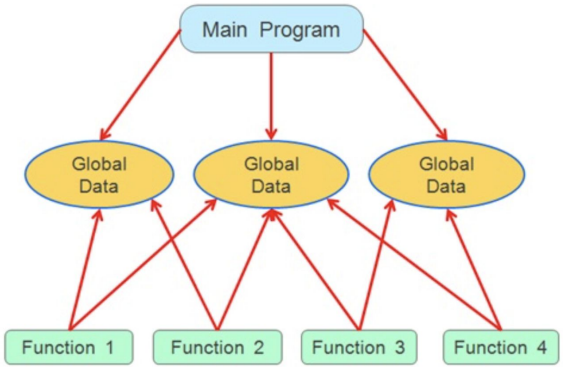
```c
#include <stdio.h>

int main() {
    int a = 5;
    int b = 10;
    int sum = a + b;
    printf("Sum = %d\n", sum);
    return 0;
}
```

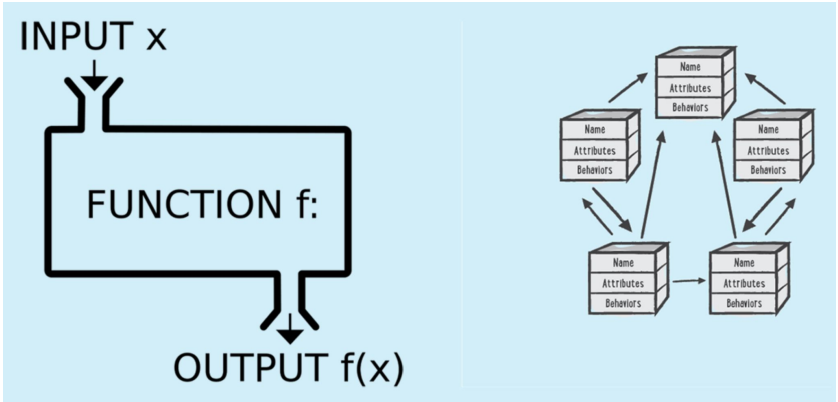
C Code

The Evolution of Programming Languages

Significant Evolutionary Steps

- Transition from **machine code** to **high-level languages**: Drastically improved programmer productivity, nearing a tenfold increase (as suggested by Fred Brooks).
- **Procedural Programming**: Introduced structured approaches to coding.
- **Object-Oriented Programming (OOP)**: Enabled encapsulation and modularity.
- **Functional Programming**: Emphasized immutability and first-class functions.





The Evolution of Programming Languages

Modern Reflections on Language Development

- **Obsession with Syntax Over Structure**: Focus has often been on syntactic changes rather than meaningful structural advances.
- **Limited Progress**: Despite extensive efforts, significant breakthroughs in language design have been rare since the 1980s.
- **Fred Brooks' Insight**: Emphasized managing expectations and focusing on foundational principles rather than seeking “magical solutions” or revolutionary changes in languages.

The Evolution of Programming Languages

Foundational Principles of Software Engineering

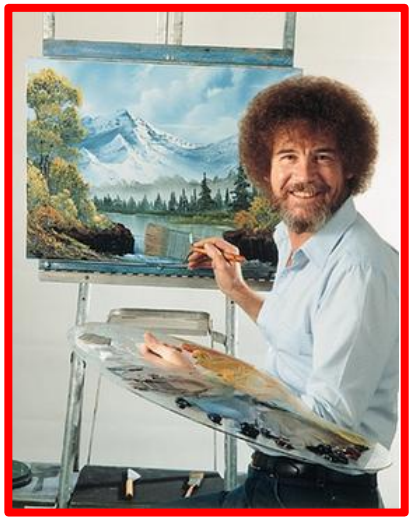
- **Practical, Incremental Development**: Brooks advocated for building systems gradually, starting with a minimal running system and incrementally adding functionality.

- **Philosophy Over Technology**: Emphasized the importance of foundational principles and philosophical approaches over specific language or tool choices.

Why Does Engineering Matter?

Historical Context: Craft vs. Engineering

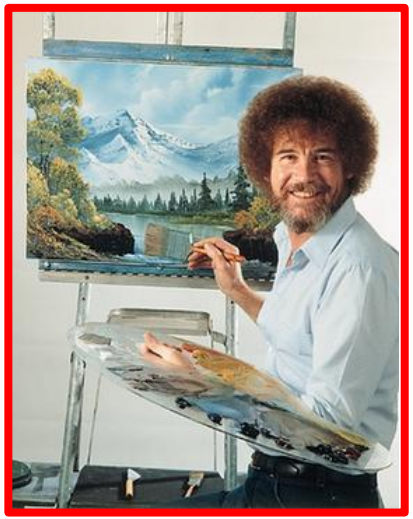
- **Craft-Based Production**:
  - For most of human history, production was dominated by craft.
  - **Characteristics of Craft**:
    - **One-Off Creations**: Each item is unique, tailored to specific requirements.
    - **Low Precision and Repeatability**: High variance in quality and specifications due to manual processes.
    - **Human Limitation**: Precision and tolerance are limited to human capabilities, impacting consistency.



Why Does Engineering Matter?

Limitations of Craft in Production

- **Inconsistent Quality**: Variability in production leads to items that may differ significantly from one another.
- **Low Scalability**: Craft methods are not suitable for mass production or creating multiple identical items.
- **High Variance**: Even skilled craftspeople cannot achieve the same level of precision and repeatability as engineered processes.



Why Does Engineering Matter?

Transition to Engineering

- **Engineering Approach**:
  - In contrast to craft, engineering seeks to standardize and optimize production processes.
  - **Precision and Consistency**: Engineering introduces methods and tools that ensure high precision and repeatability.
  - **Scalability**: Engineered processes are designed for scalability, enabling mass production with consistent quality.

Why Does Engineering Matter?

Why Engineering is Essential

- **Reliability**: Engineering enables the creation of reliable products that perform consistently under defined conditions.
- **Efficiency**: Engineering processes are optimized for efficiency, reducing waste and improving output.
- **Foundation of Knowledge**: As Grace Hopper stated, programming (a form of engineering) is not just practical but also foundational to knowledge, reflecting the importance of engineering principles in advancing technology.

The Limits of “Craft”

Emotional Appeal of Craft

- **Human Connection**: We often value handcrafted items for their uniqueness and the perceived skill, love, and care of the craftsperson.
- **Perceived Quality**: Craft items are often seen as high-quality due to their individuality and the artisan’s touch.

The Limits of “Craft”

Precision and Quality in Craft vs. Engineering

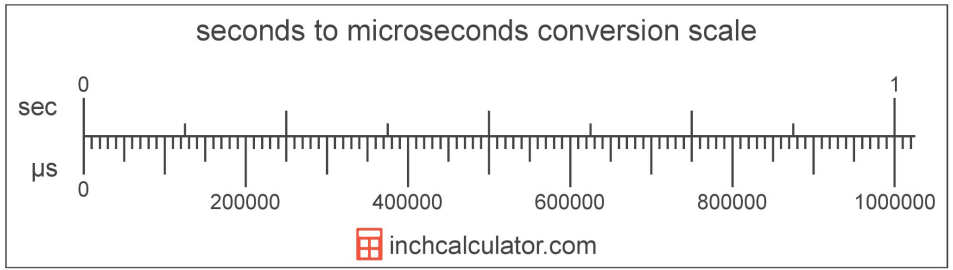
- **Craft Limitations**:
  - **Low Precision**: Even the most skilled craftsperson is limited to manual accuracy, typically around **1/10 of a millimeter**.
  - **Machine Precision**: Modern machines can manipulate objects at the atomic level, achieving precision millions of times greater than human capability.
  - **Example**: Atoms are typically measured in tens of picometers (**1 x 10^-12 m**), far beyond human ability.

The Limits of “Craft”

Relevance of Precision in Software Engineering

- **Human Perception**:
  - Humans perceive changes with a minimum threshold of approximately **13 milliseconds**.
  - Reaction times, like processing an image, often take **hundreds of milliseconds**.

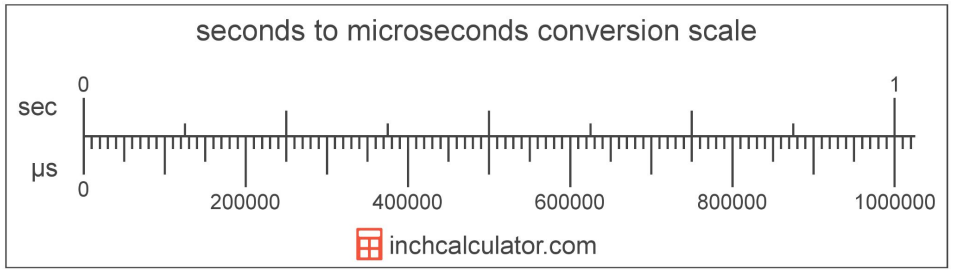
- **Computer Processing Power**:
  - Modern consumer computers operate at around **3 GHz**, or **3 billion cycles per second**.
  - Even ignoring multicore processing, this means a computer can perform **3 billion operations per second**.
  - Within the time it takes a human to perceive a change (13 ms), a computer can execute approximately **39 million instructions**.



The Limits of “Craft”

Implications for Software Development

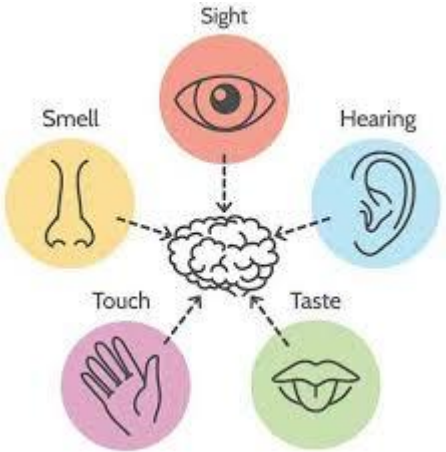
- **Human vs. Machine Accuracy**:
  - Limiting software quality to human perception ignores the incredible precision and speed of computers.
  - At a sampling rate of **1:(39 million)**, relying on human-scale accuracy risks missing critical computational details.
- **Need for Rigorous Standards**:
  - Software engineering must go beyond craft-based, human-scale perceptions to fully leverage machine capabilities.
  - Emphasizes the importance of precise, systematic approaches in software development.



Precision and Scalability

Craft vs. Engineering

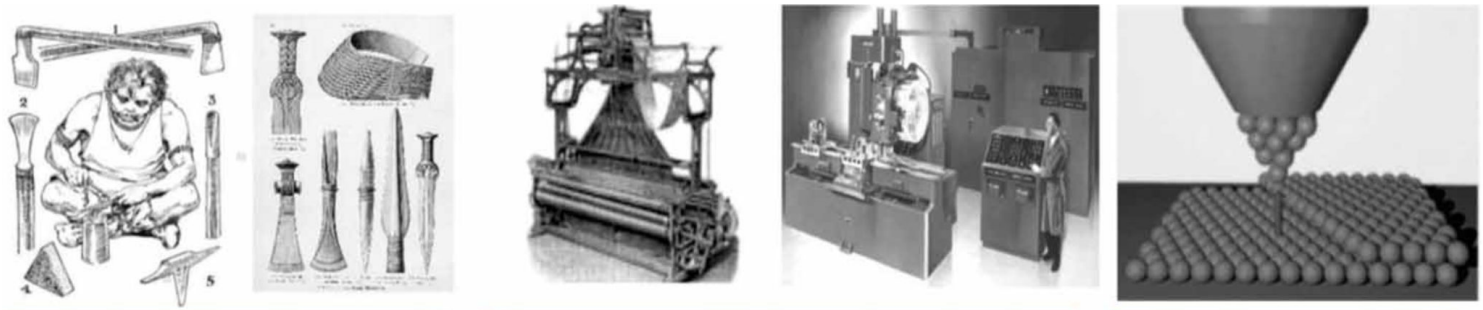
- **Craft-Based Approach**: Relies on human skill and manual effort.
    - **Limitations**: Precision and scalability are constrained by human capabilities.
    - **Human Limits**: Even with exceptional skill, human precision has hard limits due to the inherent limitations of muscles and senses.



Precision and Scalability

Engineering Approach

- **Precision**:
  - **Higher Resolution of Detail**: Engineering allows manipulation at a much finer scale than manual methods.
  - **Machine Precision**: Machines can achieve precision down to atomic and subatomic levels, far beyond human ability.
- **Scalability**:
  - **Beyond Human Limits**: Engineering creates tools and machines that can be scaled up or down to handle tasks from the quantum scale to the cosmological scale.
  - **Theoretical Limitlessness**: In theory, engineering has no boundaries in terms of scale, whether manipulating atoms or potentially controlling celestial bodies in the future.



Precision and Scalability

Scalability in Software Testing

- **Human Testing Limitations**:
  - Even with rapid testing, a human can only perform a limited number of tests per minute.
  - **Example**: At best, a human might execute one test per minute, but this is not sustainable long-term.

- **Computational Testing Power**:
  - **Automated Testing**: Computers can run thousands to millions of tests in the time it takes a human to perform a single test.
  - **Real-World Example**:
    - Systems can run **30,000 test cases in about 2 minutes**.
    - Google runs approximately **150 million test executions per day**, averaging **104,166 tests per minute** .
  - **Continuous Operation**: Computers can maintain this testing pace indefinitely as long as they have power, demonstrating true scalability.

Dice total	Actual	Predicted	Variance	Percentage
2	145,508	145,500	-8	-0.005498282
3	290,446	291,000	554	0.190378007
4	437,179	436,500	-679	-0.155555556
5	582,591	582,000	-591	-0.101546392
6	728,053	727,500	-553	-0.076013746
7	872,988	873,000	12	0.00137457
8	726,607	727,500	893	0.122749141
9	582,077	582,000	-77	-0.013230241
10	435,819	436,500	681	0.156013746
11	291,223	291,000	-223	-0.076632302
12	145,509	145,500	-9	-0.006185567
Total Rolls→		5,238,000		

Managing Complexity

Engineering vs. Craft in Complexity Management

- **Craft-Based Production**:
  - Historically, products like guns were crafted by individual artisans (e.g., gunsmiths).
  - **Unique Creations**: Each product was unique, with components specifically made for each item (e.g., custom bullets and screws).
  - **Limited Scalability**: The entire product was crafted by a single person, making scaling and consistency difficult.

Managing Complexity

Transition to Engineering and Mass Production

- **Example: American Civil War**
  - Marked a shift from craft to mass production in armaments.
  - **Standardization**: Introduced interchangeable parts that could be assembled easily, allowing mass production.
  - **Modular Design**: Components were designed to be uniform and compatible, enabling scalable production processes.
- **Impact of Mass Production**:
  - **Increased Precision and Scalability**: Machines could produce parts with consistent quality, far surpassing human craftsmanship.
  - **De-skilling Production**: Less skilled workers could produce complex products with the aid of machinery, making production faster and cheaper.
  - **Enhanced Quality and Affordability**: Mass-produced items became more precise and affordable than handcrafted alternatives.

Managing Complexity

Engineering Principles in Software Development

- **Modular and Componentized Design**:
  - Similar to how guns were designed with interchangeable parts, software can be modularized.
  - **Independent Development**: Different components or modules can be developed, tested, and improved independently.
  - **Managing Complexity**: Modular design simplifies complexity by allowing incremental changes and improvements without affecting the entire system.
- **Focus on Design, Not Production**:
  - The lesson from mass production is not about replicating production processes but applying design principles that allow for scalable, manageable software systems.
  - **Design Thinking**: In software, managing complexity means organizing code and systems in a way that allows for easy updates, maintenance, and scalability.

Managing Complexity

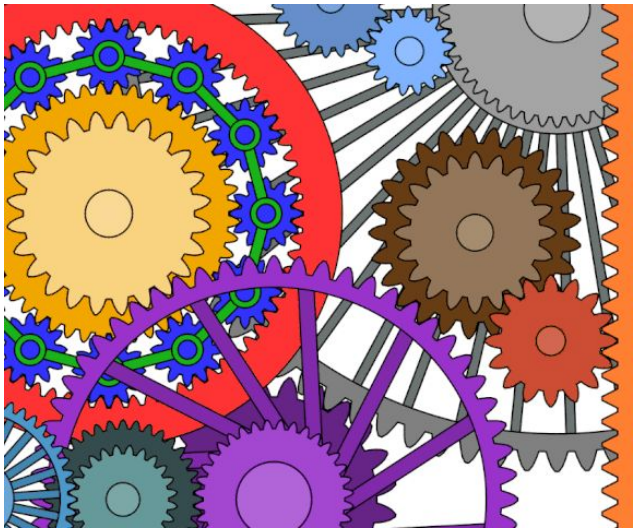
Key Takeaways

- **Complexity Management**: Engineering techniques, such as modular design, help manage and reduce complexity by compartmentalizing problems.
- **Modular Design in Software**: Allows for independent development and easier management of large systems, akin to modular weapons manufacturing.
- **Incremental Improvement**: Emphasizes iterative enhancement and adaptation, improving systems step-by-step rather than requiring complete overhauls.
- **Edsger Dijkstra's Insight**: "The art of programming is the art of organizing complexity." Effective software engineering is about structuring and managing complexity through thoughtful design.

Repeatability and Accuracy of Measurement

Engineering Principles: Repeatability and Precision

- **Repeatability**:
  - In traditional engineering, repeatability ensures that components (e.g., nuts and bolts) are produced with consistent quality and can reliably work together.
  - This concept is often misunderstood when applied to software, as it is seen as a production issue rather than a design principle.
- **Precision in Measurement**:
  - **Fundamental Idea**: The ability to measure accurately is crucial in any engineering discipline.
  - In software, precision allows early detection of issues, preventing catastrophic failures.



Repeatability and Accuracy of Measurement

Software Failures and Engineering Solutions

- **Craft vs. Engineering Approach**:
  - **Craft-Focused Team**:
    - Likely to address software failures by increasing testing duration, such as using soak tests.
    - **Soak Test**: Runs software for a prolonged period to observe failures and identify bugs (e.g., running for three weeks to detect a failure that occurs every two weeks).
    - **Limitations**: Reactive approach; waits for issues to become visible, leading to longer testing times and potential production failures.
  - **Engineering-Focused Team**:
    - Uses precise measurement techniques to proactively detect issues before they escalate.
    - **Early Detection**: Measures software performance continuously to catch resource leaks or other issues early, avoiding extended failures and reducing testing time.

Repeatability and Accuracy of Measurement

Real-World Analogy: Detecting Leaks

- **Obvious Detection Strategy**: Waiting for visible signs of a leak, similar to observing software failures after they occur.
- **Engineered Solution**: Using sensitive tools (like a microphone for detecting water leaks) to identify issues early with high precision, preventing extensive damage and enabling targeted fixes.

Repeatability and Accuracy of Measurement

Benefits of Engineering-Focused Approaches

- **Preventing Catastrophic Failures**: Early detection through precise measurement minimizes risks and avoids major production failures.
- **Efficient Testing**: By incorporating accurate monitoring into regular testing, issues can be identified quickly, often within minutes, rather than waiting for prolonged soak tests.
- **Continuous Feedback**: Provides valuable, timely insights into system health, enabling quicker responses and ongoing improvements.

Engineering, Creativity, and Craft

Engineering vs. Craftsmanship in Software

- **Software Craftsmanship**:
  - Emerged as a response to rigid, production-centered approaches (like Waterfall) in the 1980s and 1990s.
  - Emphasizes key principles:
    - **Skill**: Mastery of software development techniques.
    - **Creativity**: Freedom to explore and innovate.
    - **Freedom to Innovate**: Encourages creative problem-solving and new ideas.
    - **Apprenticeship**: Mentorship and learning from experienced developers.
- **Engineering in Software**:
  - **Design Engineering**: Focuses on solving complex, exploratory problems rather than repetitive production tasks.
  - Combines craftsmanship with scientific methods, enhancing quality, scalability, and effectiveness.

Engineering, Creativity, and Craft

Misconceptions About Engineering

- **Not Just Production**: Engineering is not only about manufacturing and production; it's about **designing solutions**.
- **Incorporates Creativity and Skill**: Engineering disciplines require creativity, innovation, and mastery—traits also valued in craftsmanship.
- **Training and Mentorship**: Like in craftsmanship, new engineers work alongside experienced professionals to learn and grow.

Engineering, Creativity, and Craft

Advantages of Engineering Over Craft

- **Scalability**:
  - Engineering solutions are scalable, making them suitable for complex, large-scale problems.
  - Unlike craft, which is limited by individual effort, engineering can leverage technology and processes to scale up.
- **Quality and Efficiency**:
  - Engineering disciplines focus on improving quality, reducing costs, and delivering robust, resilient solutions.
  - **Data-Driven Improvement**: Measurable improvements in quality and productivity are achieved through disciplined engineering practices.

Engineering, Creativity, and Craft

Engineering as an Evolution of Craft

- **Craft Enhanced by Science**: Engineering builds on the foundations of craftsmanship, incorporating scientific rationalism and precision.
- **High-Tech vs. Agrarian Civilizations**: The transition from craft to engineering is akin to the shift from agrarian to high-tech civilizations, enabling complex problem-solving and advanced solutions.
- **Accelerate Study**: Teams using disciplined engineering approaches spend **44% more time on new work**, highlighting increased productivity and effectiveness.

Why What We Do Is Not Software Engineering 1:3

Engineering Decisions: A Case Study from SpaceX

- **2019 Decision by SpaceX**:
  - SpaceX was working on Starships to enable human exploration and colonization of Mars and beyond.
  - Initially used **carbon fiber** for spacecraft construction but switched to **stainless steel**.
- **Reasons for the Material Change**:
  - **Cost**: Stainless steel was significantly cheaper per kilogram than carbon fiber.
  - **High-Temperature Performance**: Stainless steel could better withstand the extreme temperatures during re-entry compared to aluminum.
  - **Low-Temperature Performance**: Stainless steel performed much better at cryogenic temperatures than both carbon fiber and aluminum.

Why What We Do Is Not Software Engineering 2:3

Characteristics of True Engineering Decisions

- **Rational Criteria**: Decisions are based on logical, measurable factors such as cost, strength, and material properties.
- **Empirical and Iterative**:
  - Engineering involves making decisions based on current evidence and theories.
  - It includes testing and refining ideas through experimentation and observation.
  - For example, SpaceX tested the cryogenic performance of stainless steel by building test structures and pressurizing them with water and liquid nitrogen.
- **Exploratory Approach**: Engineering is an ongoing process of discovery, constantly testing and adjusting to improve designs and outcomes.

Why What We Do Is Not Software Engineering

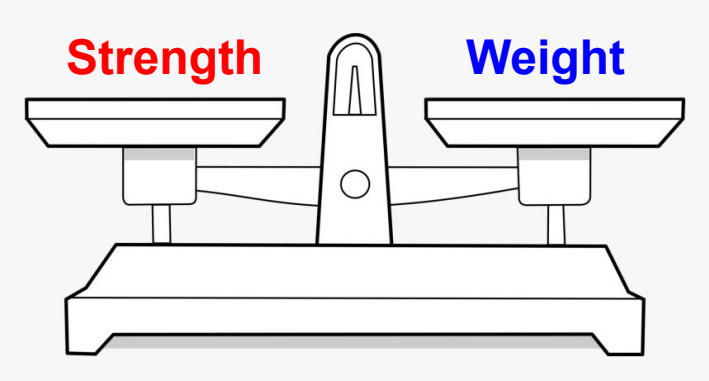
Contrast with Software Development Practices

- **Lack of Engineering-Like Decision Making**:
  - Software creation often lacks the rigorous, evidence-based decision-making process seen in engineering.
  - Decisions in software development are rarely based on empirical criteria such as performance under specific conditions or cost-effectiveness.
- **Software vs. Engineering**:
  - True engineering decisions are methodical and data-driven, considering a range of factors before committing to a solution.
  - Software development often does not follow this level of structured, scientific decision-making, which can lead to less predictable and less optimized outcomes.

Trade-Offs

The Nature of Engineering

- **Optimization and Trade-Offs**:
  - All engineering disciplines involve optimizing solutions by making informed trade-offs.
  - **Example from SpaceX**: When building rockets, a critical trade-off is between **strength** and **weight**—essential for balancing performance and efficiency.
  - Trade-offs are common in many engineering domains, including aerospace, automotive, and software.



Trade-Offs

Understanding Trade-Offs in Software Development

- **Security vs. Usability**:
  - Increasing security measures often makes a system more secure but can reduce its ease of use.

- **Distribution vs. Integration Complexity**:
  - Making a system more distributed can improve scalability and reliability but may increase the complexity of data integration and consistency.

- **Team Size vs. Communication Overhead**:
  - Adding more developers to a project can speed up development initially but increases communication overhead, which can lead to greater coupling and complexity.

  - This can counterintuitively slow down progress due to the need for coordination and integration among team members.

Trade-Offs

Key Trade-Off: Coupling in Software Systems

- **Coupling**:
  - Refers to the degree of interdependence between software modules or components.
  - **Low Coupling**: Modules are more independent, easier to understand, test, and maintain.
  - **High Coupling**: Modules are more interconnected, which can lead to increased complexity and difficulty in modifying or extending the system.
- **Importance of Managing Coupling**:
  - Properly managing coupling is crucial at all levels of software development, from entire enterprise systems to individual functions.

The Illusion of Progress

High Rate of Change in the Industry

- **Impressive but Not Always Significant**:
  - Many changes in software development are more about new tools than fundamental improvements in design or architecture.
  - Example: The move to **serverless computing** is often discussed in terms of tools rather than the architectural principles it introduces.

The Illusion of Progress

Serverless Computing: A Case Study

- **What is Serverless Computing?**
  - A cloud-based model providing **Functions as a Service (FaaS)**.
  - Code is executed on demand, with the cloud provider managing the server infrastructure.

- **Key Design Considerations**:
  - **State Management**: Where and how to store and manipulate state in a serverless environment.
  - **Function Decomposition**: Deciding how to break down the system into functions.
  - **System Navigation**: Managing complexity when the unit of design is a function.

- **Common Misfocus**:
  - Presentations and discussions often focus on the tools and specific features of platforms (e.g., AWS, Azure) rather than on fundamental design and architecture decisions.

The Illusion of Progress

Understanding True Progress

- **Beyond Tools**: The tools and platform-specific details are less important than understanding the **principles** that guide good design.
- **Analogy to Carpentry**:
  - It’s like being taught the differences between types of screws without understanding their use cases or when to choose screws over nails.
  - The focus should be on when and how to use different tools and techniques effectively, not just on the tools themselves.

The Illusion of Progress

Why Serverless Matters

- **Modular Design**: Encourages a more modular approach with better separation of concerns, particularly around data handling.
- **Changing Economics**: Shifts the cost focus from storage (cost per byte) to computation (cost per CPU cycle).
    - **Optimizing for Serverless**: Requires rethinking optimizations, such as favoring non-normalized data stores and eventual consistency over strict normalization.
    - **Impact on System Modularity**: These changes enhance the modularity and flexibility of software systems, which is a more fundamental improvement than any specific tool or platform feature.

The Journey from Craft to Engineering

The Value of Craft

- **Attention to Detail**: Craft emphasizes careful, detailed work, which is essential for creating high-quality products.
- **Importance in Engineering**: Engineering builds upon the foundation of craft, enhancing the quality and effectiveness of products through systematic methods and tools.

The Journey from Craft to Engineering

Case Study: The Wright Brothers

- **Pioneers of Flight**: The Wright Brothers were the first to build a controllable, heavier-than-air, powered flying machine.
- **Blend of Craft and Engineering**:
  - **Craftsmanship**: They constructed their airplane with wood, wire, and cloth, showcasing detailed craftsmanship.
  - **Engineering**: They conducted empirical research and developed a wind tunnel to test and refine their wing designs, blending craft with scientific methods.

The Journey from Craft to Engineering

Engineering Principles in Action

- **Empirical Discovery**: The Wright Brothers used a process of trial and error, controlled experiments, and empirical measurements to improve their designs.
- **Use of Measurement Tools**: They were the first to build a wind tunnel to measure the effectiveness of different wing shapes and refine their understanding of aerodynamics.
- **Controlled Experimentation**: By controlling variables and conducting systematic tests, they deepened their understanding of flight and aerodynamics.

The Journey from Craft to Engineering

Evolution of Aerodynamics and Wing Design

- **Wright Flyer**:
  - Had an **8.3:1 glide ratio**, meaning for every foot of descent, it traveled 8.3 feet forward.
  - Built with natural materials like wood and cloth, it was a significant step forward but limited by the materials and understanding of the time.

- **Modern Aircraft**:
  - **Advanced Materials**: Use of carbon fiber and other advanced materials for lightweight and strong wings.
  - **Refined Theories**: Advances in aerodynamic theory, computer modeling, and materials science have dramatically improved design efficiency.
  - **High Glide Ratios**: Modern sailplanes achieve glide ratios of more than **70:1**, nearly nine times better than the Wright Flyer.

Craft Is Not Enough in Software Development 1:1

Importance of Craft

- **Creativity**: Craft emphasizes creativity and attention to detail, which are essential for high-quality software development.

Engineering as a Creative Discipline

- **Beyond Craft**: Engineering is also a deeply creative endeavor. It combines creativity with structured approaches to solve complex problems.

- **Height of Human Creativity**: Engineering represents the peak of human ingenuity, using innovative thinking to create functional, scalable, and reliable software systems.

Time for a Rethink in Software Engineering?

Current State of Software Engineering

- **Achievements and Challenges**:
  - Software has revolutionized the world with innovative and exciting systems.
  - Despite this, many teams, organizations, and developers struggle to find clear paths to success or consistent progress.

- **Industry Complexity**:
  - The field is filled with diverse philosophies, practices, processes, and technologies.
  - There is ongoing debate over the best programming languages, architectural approaches, development processes, and tools.

Time for a Rethink in Software Engineering?

Common Issues in Modern Software Development

- **Pressure and Uncertainty**:
  - Teams face intense schedule pressures, quality issues, and maintainability challenges.
  - Often, they lack the time to fully understand the problem domain, technology, or opportunities for innovation.

- **Misalignment with Organizational Goals**:
  - Organizations frequently feel dissatisfied with software development outcomes.
  - Misunderstandings about how to support development teams and achieve desired results are common.

Time for a Rethink in Software Engineering? 3:5

Need for a Fundamental Rethink

- **Expert Consensus**:
  - There is a broad but not always clearly articulated agreement among experts on some fundamental principles of software engineering.

- **Core Questions**:
  - What are the enduring principles that underpin successful software development?
  - What foundational practices should guide us, regardless of the specific tools or technologies we use?

Time for a Rethink in Software Engineering?

Foundational Principles of Software Engineering

- **Universal Practices**:
  - Certain practices and ways of thinking have a profound impact on the effectiveness of software development across all domains and tools.

- **Focus on Fundamentals**:
  - This includes understanding what works and why, and applying these insights consistently to improve outcomes.

Time for a Rethink in Software Engineering?

The Future of Software Engineering

- **Towards a Genuine Engineering Discipline**:
  - We are at a pivotal moment where we can transition from an ad hoc, craft-based approach to a true engineering discipline.
  - This shift could transform how software development is practiced, organized, and taught.

- **Potential Benefits**:
  - Faster, more cost-effective software development.
  - Higher quality, more maintainable, adaptable, and resilient software systems.
  - Better alignment with user needs and organizational goals.

End.

Questions?