Fundamentals of an Engineering Approach

Chapter 3 of Modern Software Engineering

Fundamentals of an Engineering Approach

Engineering Across Disciplines

- **Variety in Engineering**:
  - Different fields (e.g., bridge building, aerospace, electrical, chemical) have unique methods and challenges.
  - Despite these differences, all engineering disciplines share some foundational principles.

Common Principles of Engineering

- **Scientific Rationalism**:
  - Grounded in scientific principles and reasoning.
  - Relies on logical analysis and evidence-based decision-making.

- **Pragmatic and Empirical**:
  - Practices are practical and focused on real-world applications.
  - Empirical methods, such as experimentation and observation, are essential for making progress.

Fundamentals of an Engineering Approach

Applying Principles to Software Engineering

- **Relevance to Software Engineering**:
  - Software engineering, like all engineering disciplines, should be grounded in scientific rationalism and empirical methods.
  - These principles help create reliable, maintainable, and scalable software systems.

- **Examples in Software Development**:
  - **Test-Driven Development (TDD)**: Emphasizes empirical feedback and evidence-based practices to improve code quality.
  - **Agile Methodologies**: Focus on iterative development, allowing for frequent testing and adaptation based on empirical data.

Building a Foundation for Software Engineering

- **Core Concepts for Longevity**:
  - Principles should be fundamental to software development, such as modularity, abstraction, and design patterns.
  - Must be robust enough to endure technological and environmental changes.

- **Creating a Solid Framework**:
  - These principles guide the development of best practices and behaviors in software engineering.
  - Aim to create a framework adaptable to the evolving landscape of software development.

Fundamentals of an Engineering Approach

Interactive Question:

- Ask students to share examples from their experience where empirical methods or scientific approaches have improved a software project.

- Have students discuss the different ways these principles can manifest in software development and how they help create more robust and adaptable systems.

An Industry of Change?

The Risk of Evolving and Understanding Change in Software Development

Risk of Frequent Change and Hype:

- The software industry is characterized by rapid changes and excitement over new technologies.
- Despite the hype, many of these changes have minimal impact on the core principles of software development and can sometimes lead to inefficiencies and complications.

Risk of Change:

- **Over-complication:** New technologies can increase complexity if they are not a good fit for the project.
- **Performance Issues:** Tools may cause bottlenecks if not optimized for specific use cases.
- **Learning Curve:** New methodologies require time and resources to learn, which can slow development.
- **Poor Support and Documentation:** Inadequate support or documentation can delay development and troubleshooting.
- **Limited Expertise:** Few developers skilled in the new technology can create bottlenecks and increase dependency on select team members.
- **Lack of Stability:** Immature tools and frameworks may have bugs and frequent changes, making them unreliable.

Discussion Prompt:

- Can you think of examples where adopting a trendy technology or methodology led to more harm than good? How did it affect the project, and what could have been done differently?

An Industry of Change?

Case Study: Hibernate vs. SQL

Imagine you are a Java developer working on a project that requires frequent and complex interactions with a database. Your team is excited about adopting new technologies and has decided to use Hibernate, a popular object-relational mapping (ORM) library, believing it will streamline database operations and improve productivity.

At first glance, Hibernate seems promising. It abstracts the underlying SQL queries, allowing developers to interact with the database using Java objects. This abstraction layer is supposed to make database interactions easier and more intuitive, aligning well with the object-oriented nature of Java.

However, as the project progresses, the team begins to notice some unexpected challenges.

Christin Gorman, a software developer, and speaker, highlights these challenges in her presentation comparing Hibernate with direct SQL for database interactions.

An Industry of Change?

3:12

Case Study: Hibernate vs. SQL

1. **Complexity and Code Volume**:
    - *Initially, the promise of Hibernate was to reduce the amount of boilerplate code by automatically generating SQL queries. However, in practice, the team finds themselves writing more code to configure Hibernate, manage entity mappings, and handle exceptions.*
    - *For example, simple CRUD (Create, Read, Update, Delete) operations that could have been performed with straightforward SQL statements now require understanding and managing Hibernate-specific configurations and annotations. What was supposed to be a simplification turns into a cumbersome process.*

2. **Performance and Understandability**:
    - *The team also starts to notice performance issues. Hibernate's abstraction layer introduces additional overhead, resulting in slower query execution compared to direct SQL.*
    - *Furthermore, debugging becomes more challenging. When something goes wrong, the generated SQL from Hibernate can be complex and hard to decipher, making it difficult for developers to understand what's happening under the hood. Direct SQL, on the other hand, is more transparent and predictable.*

An Industry of Change?

4:12

Case Study: Hibernate vs. SQL

Findings and Reflections:

- *Christin Gorman's insights reveal a critical lesson: adopting new technologies or methodologies doesn't always lead to better outcomes.*
- *In the case of Hibernate, the initial appeal of abstraction and automation was overshadowed by the increased complexity, reduced performance, and difficulty in debugging.*
- *The team reflects on their experience and realizes that, in many cases, directly writing SQL would have been simpler, more efficient, and easier to maintain.*
- *They begin to appreciate the value of choosing the right tool for the job, rather than being swayed by the allure of new technology.*

Insight:

- *This case study serves as a reminder that not all new technologies or methodologies simplify or improve software development.*
- *It underscores the importance of critically evaluating tools and technologies before adopting them, considering not just their potential benefits but also their limitations and suitability for the task at hand.*

An Industry of Change?

The Real Impact of Change

- **Temporary Nature of Changes**:
    - **Short-Lived Trends**: The software development industry is often influenced by buzzworthy trends and technologies—like certain frameworks, tools, or methodologies—that promise substantial improvements but fade quickly.
    - **Lack of Lasting Impact**: Many of these changes fail to bring sustainable improvements or are replaced by new trends before they can be fully integrated or assessed for long-term value.
- **Examples of Ephemeral Changes**:
    - **Hype-Driven Technologies**: Consider the rise and fall of certain JavaScript frameworks (Meteor, Backbone, Angular, Cordova, ColdFusion, Bower) that quickly gain popularity but are soon overshadowed by newer, more optimized options, leaving developers to constantly adapt with minimal long-term benefits.
    - **Methodology Fads**: Agile-like methods or development paradigms that get watered down or misapplied due to misinterpretation or overzealous adoption without proper understanding.

An Industry of Change?

6:12

The Real Impact of Change

Discussion Prompt:

- **Compare/Contrast with Food Science**: Much like the software industry, food science has seen significant shifts in recommendations over the years. For example, the food pyramid was once widely advocated as the standard for a healthy diet but is now considered outdated and even potentially harmful in some aspects. What parallels do you see between this and trends in software development?

- **Student Reflection**:
  - **Examples in Software Development**: Ask students to share examples of technologies, frameworks, or methodologies they've encountered that seemed revolutionary at first but didn't provide lasting benefits. What made these changes seem promising initially, and why did they fail to deliver in the long run?

  - **Analogous Situations in Other Fields**: Encourage students to think about other fields where they've seen similar shifts—such as education, fitness, or business practices. Are there other instances where commonly accepted practices were later proven to be flawed or less effective?

- **Discussion on Ground Truth**:
  - **Question**: Is there such a thing as a "ground truth" in software development, or is it more about adapting to the current understanding and context? How does this compare to fields like food science or medicine, where recommendations evolve based on new research?

An Industry of Change?

The Real Impact of Change

- **10x Losses in Software Development**:
  - While there may not be many "10x gains," there are significant setbacks ("10x losses") caused by poor decisions or outdated practices.

- **Examples of 10x Losses**:
  1. **Sticking with Outdated Processes**:
     - Companies that continue to use outdated development processes, such as the waterfall model, may struggle to adapt to changing requirements and fail to deliver software on time.
     - **Impact**: This can result in projects being delayed for years, or in some cases, failing altogether due to the inability to meet modern development needs.

  2. **Outsourcing Development with Low-Quality Results**:
     - Companies that outsource development work to teams with lower quality standards or insufficient expertise may face significant technical debt.
     - **Impact**: The resulting software may require extensive code refactoring, debugging, and rewriting, leading to increased costs, delayed delivery, and potential damage to the company's reputation.

  3. **Using AI Tools that Produce Erroneous Output**:
     - Relying on AI tools for code generation or other development tasks without proper oversight can result in erroneous or suboptimal output.
     - **Impact**: Developers may spend considerable time and effort debugging and understanding the AI-generated code,

An Industry of Change?

8:12

The Real Impact of Change

Case Study: iPad hardware versus software

- **The Role of Hardware Advancements:**
  - The iPad's hardware has significantly improved each year, with more powerful processors, increased memory, and enhanced graphics, rivaling some MacBooks.
  - These upgrades allow the iPad to perform complex tasks like video editing and gaming, showcasing its hardware potential.

- **Progress Masked by Hardware Improvements:**
  - Despite powerful hardware, the iPad's operating system (iPadOS) often limits its capabilities. Restrictions on multitasking and file system access prevent professional-grade apps from fully utilizing the hardware.
  - This creates a misleading sense of progress—while the hardware suggests the iPad could replace a laptop, software limitations keep it from being fully functional in many professional contexts.

- **True Progress in Software Methodologies:**
  - Real progress requires more than hardware upgrades; it needs software advancements that fully leverage the hardware.
  - The iPad example shows that as hardware evolves, software must also advance to remove limitations and support robust applications. True progress comes when both hardware and software evolve together.

An Industry of Change?

Reflecting on Significant Changes in Software Development

Transformative Changes:

- Consider changes that have genuinely transformed your approach to software development:
  - **Moving from Assembly to C:** Introduced higher-level programming concepts.
  - **Transitioning from Procedural to Object-Oriented (OO) Programming:** Enabled managing greater complexity through abstraction.

Impact of Programming Paradigms:

- Shifts in programming paradigms (e.g., from procedural to OO) have a more significant impact than changes between individual languages.

An Industry of Change?

Risk of NOT Evolving and the Challenges in Learning and Unlearning

Struggles in Adapting and Discarding:

- The software industry often struggles to transition from outdated methodologies to newer, more effective practices.

- **Example:** A company that didn't release any software for over five years due to outdated processes and fear of change, highlighting the risks of clinging to obsolete practices.

Risk in the Resistance to Change:

- **Stagnation and Technical Debt:** Failure to adopt new methodologies can cause stagnation and accumulate technical debt, making future changes costly and challenging.
- **Missed Opportunities:** Holding onto outdated processes can delay product delivery, miss market opportunities, and prevent staying competitive.
- **Loss of Competitive Edge:** Companies that resist change may fail to meet evolving market demands, resulting in a loss of relevance and market share.

Discussion Prompt:

- Can you think of other examples where resistance to change has hindered progress in a project or organization? What were the consequences, and how could they have been avoided?

An Industry of Change?

Case Study: Blockbuster vs. Netflix

Background:

- **Blockbuster:** A leader in video rentals with thousands of stores.
- **Netflix:** Began as a DVD rental service and transformed into a streaming giant.

The Challenge:

- **Blockbuster:** Resistant to adapting to on-demand digital content.
- **Netflix:** Embraced streaming early, aligning with technological and consumer trends.

Key Events:

- **Early 2000s:** Netflix proposed a partnership; Blockbuster declined.
- **Mid-2000s:** Netflix launches streaming services, capitalizing on technology advancements.
- **Late 2000s:** Blockbuster's delayed digital pivot failed, leading to its decline.

Lessons Learned:

- **Adaptability:** Netflix thrived by quickly innovating.
- **Resistance to Change:** Blockbuster's reluctance to evolve led to its downfall.
- **Takeaway for Software Development:** Adapting to change is crucial for staying relevant and competitive.

An Industry of Change?

Key Takeaways

Not All Change Is Progress:

- Evaluate the real impact of changes before adopting them.
- Focus on paradigm shifts that truly enhance software quality, scalability, or complexity.

Learning and Adaptation:

- Embrace meaningful advancements and be ready to discard ineffective practices to drive genuine progress.

Strategies for Successfully Evolving *(Survival of the Fittest requires a Fitness Function)*

- **Develop Systems of Measures:** Establish effective metrics and measurement systems to analyze changes and their impacts.
- **Support Analysis:** Use these measures to make informed decisions about adopting new technologies or processes, ensuring they contribute to long-term success.

Interactive Poll/Question for Students:

- **Question:** Reflecting on both risks (i.e. evolving, not evolving)—can you think of an example where a change in technology or process resulted in a "10x loss," or where resisting change led to significant setbacks? How did these decisions impact the project or organization?

Importance of Measurements in SWE

Measuring Success and the Challenges in Discarding Ineffective Practices

Inadequate Performance Metrics:

- Common metrics like lines of code, velocity, and test coverage can be misleading.
- Traditional metrics often fail to accurately measure software quality and productivity.
    - Example: Evaluating an author by the word count instead of the story quality.

Why Traditional Metrics Can Be Misleading:

- **Lines of Code:**
    - Higher code volume doesn’t equate to better productivity or code quality.
    - Can result in unnecessarily complex code and discourage essential refactoring.

- **Test Coverage:**
    - High coverage rates may create a false sense of security.
    - Emphasizing coverage over test quality can overlook the effectiveness of tests and miss critical issues.

Importance of Measurements in SWE

Evolving Towards Meaningful Measurement

Traditional Views on Measurement:

- **Agile Development:** Historically, agile practices have been skeptical about using metrics to measure team performance, favoring flexibility and adaptability.

- **Martin Fowler (2003):** Highlighted the industry's challenge in finding reliable productivity measures, pointing out that many traditional metrics don’t accurately reflect team effectiveness.

The Accelerate Model:

- **Developed by:** Nicole Forsgren, Jez Humble, Gene Kim.
- **Focus:** Uses empirical data to establish meaningful metrics for predicting performance.
- **Key Metrics:**
    - **Stability:** Measures reliability (e.g., change failure rate, recovery time).
    - **Throughput:** Assesses efficiency (e.g., lead time, deployment frequency).

Importance of Measurements in SWE

Key Metrics in the Accelerate Model

Stability:

- **Change Failure Rate:** Frequency of defects from changes.
- **Recovery Time:** Speed of recovery from failures.
- **Significance:** Reflects software reliability and quality.

Throughput:

- **Lead Time:** Duration from idea to deployment.
- **Deployment Frequency:** Rate of successful deployments.
- **Significance:** Indicates development efficiency and speed.

Importance of Measurements in SWE

Transforming Decision-Making with the Accelerate Model

Predictive Capabilities:

- Uses stability and throughput metrics to forecast performance.
- Helps teams pinpoint areas for improvement.

High-Performing Teams:

- Achieve high stability and throughput for superior software delivery.
- Practices include test automation, trunk-based development, deployment automation, and continuous delivery.

Low-Performing Teams:

- **Characteristics:** Low stability and throughput, often struggling with frequent failures and slow delivery.
- **Common Issues:** Lack of automation, complex branching strategies, manual deployment, infrequent releases.

Importance of Measurements in SWE

Interactive Question for Students

Question: Think about a project you've worked on. How did the metrics used influence the project's outcomes?

- **Did they lead to a focus on quantity over quality?** (e.g., lines of code, feature count)
- **Did they encourage a focus on quality and efficiency?** (e.g., stability, throughput)
- **Did they significantly impact the project outcomes?**
- **Were there any formal metrics defined and used?**

Discussion:

- Invite students to share their experiences with metrics in past projects. *(i.e. Autolab? Canvas? Moodle?)*
- Discuss the impact that different types of measurements had on team behavior and project outcomes.
- Explore how adopting stability and throughput as metrics might change their approach to future projects.

Applying Stability and Throughput in SWE

The Importance of Evidence-Based Decision Making

Using Stability and Throughput Metrics:

- **Stability**: Measures the quality and reliability of software by tracking metrics such as change failure rate and recovery time.
- **Throughput**: Evaluates efficiency and productivity through metrics like lead time and deployment frequency.
- These metrics provide a data-driven framework to evaluate and improve software development processes, culture, and technology.

Applying Stability and Throughput in SWE

Case Study: Change Approval Boards (CABs) in Software Development

Background:

- **CABs in Software Development:** Many organizations introduced Change Approval Boards (CABs) with the intention of improving software quality.
- **Purpose of CABs:** Designed as a quality checkpoint to review every change before deployment, aiming to reduce bugs and ensure more stable releases.

The Challenge:

- **Intended Benefits:** CABs were expected to enforce standards and improve overall stability by catching issues early through thorough reviews.
- **Unexpected Outcomes:** However, the implementation did not go as planned, leading to unforeseen challenges.

Applying Stability and Throughput in SWE

Case Study: The Reality of CABs

Negative Correlation with Performance Metrics:

- **Impact on Processes:** Rather than enhancing software delivery, CABs actually slowed down processes, causing delays in lead times and reducing deployment frequency due to the extra review layer.

No Improvement in Stability:

- **Stability Concerns:** Despite rigorous reviews, there was no significant reduction in change failure rates. The quality of releases did not improve as expected, indicating that CABs were not effectively preventing issues.

Insights and Reflections:

- **Lessons Learned:** The case of CABs illustrates that well-intentioned practices can backfire if not properly evaluated. Overemphasis on reviews without assessing their impact on overall performance can lead to inefficiencies.

Applying Stability and Throughput in SWE

Case Study: Moving Forward from CABs

Evidence-Based Practices:

- **Importance of Metrics:** This example highlights the need for evidence-based practices in software development. Using metrics like stability and throughput provides a clearer understanding of the real impact of organizational changes.

Applying Metrics Across Development:

- **Adopting New Technologies:** Assess the effects of new tools on stability and throughput before full integration.
- **Process Improvements:** Measure the impact of switching from manual to automated testing on key metrics.
- **Cultural Changes:** Evaluate how adopting new methodologies (like Agile or DevOps) influences stability and throughput.

Applying Stability and Throughput in SWE

Practical Examples of Using Metrics for Decision-Making

Continuous Delivery:

- **Description:** Continuous Delivery is a practice where code changes are automatically built, tested, and prepared for a release to production.
- **Impact:** Teams using continuous delivery often improve stability and throughput because frequent releases lead to faster feedback on code changes and more reliable software by continuously testing the software in a production-like environment.

Trunk-Based Development:

- **Description:** Trunk-Based Development involves all developers working on a single branch or "trunk" and committing their changes directly to this branch frequently.
- **Impact:** High-performing teams using trunk-based development experience reduced change failure rates and quicker recovery times due to fewer merge conflicts and streamlined integration, as changes are merged continuously before they diverge too much.

Deployment Automation:

- **Description:** Deployment Automation is the use of automated processes to deploy code to production environments without manual intervention.
- **Impact:** Automating deployments enhances throughput by reducing manual steps, increasing deployment frequency, and contributing to more stable and efficient releases, as automated processes are less prone to human error and can be executed more quickly.

Applying Stability and Throughput in SWE

Interpreting Results and Making Informed Decisions

Understanding Trade-offs:

- Recognize that changes might not positively affect both stability and throughput.
- Example: Increasing testing might improve stability but reduce throughput.
    - Adding more automated tests can catch bugs earlier and enhance software stability.
    - However, this increases the time needed for testing, slowing down the release process and reducing throughput.

Critical Thinking:

- Analyze results within the context of your team's goals. Assess whether trade-offs are acceptable and align with project priorities.

Continuous Improvement:

- Regularly evaluate changes using stability and throughput metrics to optimize and adapt processes.
- Use insights to refine practices, balance speed and quality, and drive sustainable improvements.

Applying Stability and Throughput in SWE

Interactive Question for Students

Question: Think about a project you've worked on, either solo or in a team. Reflect on the process you used to deliver code. How did your approach affect the speed, stability, or ease of debugging? Did it align with your project goals?

Follow-Up Options:

- It improved stability but slowed down delivery.
- It sped up delivery but introduced more bugs.
- It made debugging easier but took longer to implement.
- It balanced speed and stability effectively.
- It didn't significantly change the outcomes.

Discussion:

- Encourage students to share their experiences with different approaches to delivering code.
- Discuss how these approaches affected speed, stability, and debugging time, and how these outcomes aligned with their project goals.
- Question what they might do differently in future projects to optimize their process.

Foundations of a Software Engineering Discipline 1:4

Core Competencies in Software Engineering

Expertise in Learning:

- **Embracing a Creative Discipline**:
  - Software engineering is inherently a creative process, requiring continuous exploration, discovery, and adaptation.
  - Unlike traditional engineering fields, it thrives on flexibility and innovation.

- **Focus on Learning**:
  - Mastery in learning is a core skill for software engineers.
  - Engage in a **scientific style of reasoning**: Hypothesize, experiment, and learn from outcomes to continuously improve.

- **Long-Term Relevance**:
  - The ability to learn and adapt is crucial for addressing evolving problems and technologies.
  - Continuous learning helps engineers stay relevant in a rapidly changing industry.

Foundations of a Software Engineering Discipline 2:4

Five Core Behaviors for Effective Learning

1. **Working Iteratively**:
   - Develop software in small, repeated cycles to allow for regular feedback and improvements.
   - **Example**: Agile teams using iterative sprints to deliver and refine features incrementally.

2. **Employing Fast, High-Quality Feedback**:
   - Quickly gather feedback on code and design to make timely adjustments.
   - **Example**: Continuous integration (CI) practices that provide immediate feedback on code changes.

3. **Working Incrementally**:
   - Build software piece by piece, delivering small, functional segments that can be tested and refined.
   - **Example**: Incremental development seen in feature flags and MVPs (Minimum Viable Products).

4. **Being Experimental**:
   - Embrace experimentation to discover the best solutions.
   - **Example**: A/B testing different features or algorithms to determine which performs better.

5. **Being Empirical**:
   - Make decisions based on observations and data rather than assumptions.
   - **Example**: Data-driven development that relies on metrics and analytics to guide decision-making.

Contrasting Effective and Less Effective Approaches

- **Effective Learning Practices**:
  - Foster adaptability and continuous improvement.
  - Encourage frequent testing, iteration, and feedback loops.

- **Less Effective Approaches**:

  - **Waterfall Model**:
    - Follows a linear, sequential design process that lacks flexibility.
    - Delays feedback and adjustments until late in the development cycle, often leading to costly changes and rework.

- **Key Differences**:
  - Iterative and incremental approaches provide ongoing learning and adaptation, whereas the waterfall model restricts changes until the end.

Interactive Question for Students

Question: Can you share an example from your experience where working iteratively or employing fast feedback significantly improved the outcome of a project?

Follow-Up Options:

- Iterative development allowed for early detection of design flaws.
- Fast feedback helped catch bugs before they became costly to fix.
- Both iterative development and fast feedback led to a more successful project.
- I have not experienced these practices yet.

Suggestion:
Consider examples beyond software projects. It could be a research paper for a literature course that went through multiple drafts, feedback, and revisions, or a complex task at your job where learning and adapting were essential for success.

Discussion:
Encourage students to discuss their experiences with iterative development or fast feedback, whether in software projects or other contexts. Highlight the differences they noticed compared to more rigid approaches and how these practices contributed to the project's success

Experts at Managing Complexity in SWE

Understanding Complexity in Software Development

Inherent Complexity:

- Software development involves multiple interconnected components, which creates inherent complexity.
- **Key challenges:**
  - **concurrency** (managing multiple processes simultaneously)
  - **coupling** (managing dependencies between components).

Transitioning to Organizational Complexity:

- **From Software to Organizations:** Just as software systems face complexity of interconnected components, organizations themselves are similarly intricate systems.

Next Up: Conway's Law

- **Linking Structure to Design:** We will explore how organizational structures influence the design and architecture of the software systems they create.

Experts at Managing Complexity in SWE

Conway's Law:

Definition:
"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." – Mervin Conway, 1967.

Implication:
The structure of a software system often mirrors the communication patterns and organizational structure of the teams that develop it. How teams are organized and communicate can significantly influence the modularity, coupling, and overall architecture of the software.

Understanding the Impact:

- **Reflection of Communication Patterns:** If teams are siloed and communicate infrequently, the resulting software might have tightly coupled, monolithic structures. Conversely, if teams are cross-functional and communicate frequently, the software is more likely to be modular and loosely coupled.
- **Not a Deterministic Rule:** Conway's Law suggests a natural tendency, but organizations can consciously design their communication and team structures to shape the desired software architecture. It's more about alignment and intent than inevitability.

Organizational Complexity:

- **Parallel Challenges:** Just like software systems, human organizations are complex and deal with challenges such as managing dependencies (coupling) and coordinating simultaneous tasks (concurrency). Effective communication and collaboration are crucial in both contexts to manage complexity and avoid bottlenecks.

Experts at Managing Complexity in SWE

Importance of Managing Complexity

Consequences of Ignoring Complexity:

- **Big-Ball-of-Mud Systems**: Disorganized codebases with no clear architecture, making maintenance difficult.
- **Technical Debt**: Accumulated shortcuts that increase future development costs and risks.
- **High Bug Counts**: Increased defects due to poor design and unmanaged complexity.
- **Change Aversion**: Organizations may resist changes due to fear of introducing new issues in a complex system.

Need for Thoughtful Design:

- To effectively build complex systems, developers must strategically divide problems into manageable parts.
- **Effective Decision-Making**: Managing complexity depends on:
  - The nature of the problem being solved.
  - The technologies employed.
  - Team capabilities and experience.

Experts at Managing Complexity in SWE

Core Principles for Managing Complexity

1. **Modularity**:
   - Design systems as separate, interchangeable modules to reduce interdependencies.
   - **Example**: Microservices architecture allows for independent deployment and scaling of services, enhancing modularity.

2. **Cohesion**:
   - Ensure each module has a well-defined purpose and that all components within a module work towards the same goal.
   - **Example**: A module dedicated to user authentication should only handle authentication-related functions.

3. **Separation of Concerns**:
   - Divide a program into distinct sections, each handling a specific aspect of functionality to simplify development and maintenance.
   - **Example**: MVC (Model-View-Controller) architecture separates data management, user interface, and control logic.

4. **Information Hiding/Abstraction**:
   - Hide the internal workings of modules and expose only necessary interfaces, reducing the impact of changes within modules.
   - **Example**: Using APIs to interact with a database without exposing the underlying database structure.

5. **Coupling**:
   - Minimize the degree to which modules depend on each other, making the system more modular and easier to modify.
   - **Example**: Loose coupling in microservices ensures that changes in one service do not heavily impact others.

Experts at Managing Complexity in SWE

Practical Examples of Managing Complexity

Modular Monoliths:

- **Definition:** A single codebase structured into well-defined, independent modules. This approach keeps all components in one application but maintains clear boundaries between them.
- **Benefits:** Simplifies development and deployment, as everything resides in a single codebase while still maintaining modularity.
- **Use Case:** Ideal for smaller teams or projects where full microservices might be overkill, but some separation of concerns is still needed.

Microservices:

- **Definition:** An architectural style that structures an application as a collection of loosely coupled services, each handling a specific functionality.
- **Benefits:** Increases flexibility and scalability by allowing services to be developed, deployed, and scaled independently.
- **Use Case:** Suitable for larger applications where different parts of the system need to be developed and scaled independently by different teams.

Continuous Integration and Deployment (CI/CD):

- **Definition:** A set of practices that enable teams to deliver code changes more frequently and reliably through automated testing and deployment.
- **Benefits:** Reduces manual intervention, ensures quick feedback, and helps manage complexity by continuously integrating changes into the codebase.
- **Use Case:** Applicable to both monolithic and microservices architectures to ensure that code changes are tested and deployed efficiently.

Experts at Managing Complexity in SWE

Question: Can you think of a project, whether in software development or another field, where managing complexity through one of these principles was particularly effective or ineffective? Highlight the value of these principles in everyday life, not just in technical contexts.

- **Modularity:**
  - **Software Example:** A project where modularity significantly reduced interdependencies.
  - **Non-Software Example:** Planning a large event where tasks were divided into modules (e.g., catering, entertainment, logistics) to ensure smooth coordination without overlap.
- **Cohesion:**
  - **Software Example:** A project where cohesion helped maintain clear module focus and purpose.
  - **Non-Software Example:** Organizing a team for a group project in school where each team member was assigned a role closely related to their expertise, ensuring that all efforts were aligned towards the same goal.
- **Separation of Concerns:**
  - **Software Example:** A project where separation of concerns simplified maintenance and updates.
  - **Non-Software Example:** Planning a wedding where different vendors (florist, caterer, photographer) handled their specific tasks independently, making it easier to manage each aspect without confusion.
- **Information Hiding/Abstraction:**
  - **Software Example:** A project where information hiding minimized the impact of changes.
  - **Non-Software Example:** Developing a marketing campaign where sensitive budget details were kept confidential within the finance team, reducing the risk of information leakage.
- **Coupling:**
  - **Software Example:** A project where improper coupling caused major issues.
  - **Non-Software Example:** Planning a family reunion where too many tasks were dependent on one person’s availability, causing delays when they were unavailable.

Summary: The Real Tools of SWE

The True Tools of Software Engineering

Beyond Languages and Frameworks:

- While languages, tools, and frameworks are important, they are not the essence of software engineering.
- These elements evolve and change across different projects and over time.

Focus on Foundational Ideas:

- The core concepts of **effective learning** and **complexity management** are the real tools of software engineering.
- Mastering these foundational principles enables software engineers to:
  - Make better choices about languages, tools, and frameworks.
  - Apply these technologies effectively to solve complex problems.

Summary: The Real Tools of SWE

Linking Back to Core Concepts

Learning as a Core Competency:

- Adopting a mindset of continuous learning and experimentation helps engineers adapt to new challenges and technologies.
- **Examples from the Course**:
  - **Iterative Development**: Allows teams to refine solutions progressively.
  - **Empirical Decision-Making**: Using data to guide choices ensures more effective outcomes.

Managing Complexity:

- Understanding and applying principles like modularity, cohesion, and separation of concerns helps manage the inherent complexity in software systems.
- **Examples from the Course**:
  - **Modularity**: Designing systems as interchangeable modules reduces dependencies.
  - **Information Hiding**: Protects system integrity by exposing only necessary interfaces.

Summary: The Real Tools of SWE

The Yardstick for Evaluation

Evidence-Based Decision Making:

- To make informed decisions in software development, rely on evidence and data rather than trends or assumptions.
- Use metrics like **stability** (quality) and **throughput** (efficiency) as key measures.

Key Questions to Guide Decisions:

- **Quality Assessment**: "Does this choice increase the quality of the software we create?"
  - Consider if the decision enhances stability, reducing bugs and increasing reliability.
- **Efficiency Assessment**: "Does this choice increase the efficiency of our development process?"
  - Evaluate whether the decision improves throughput, speeding up delivery and productivity.

Summary: The Real Tools of SWE

Practical Application

Guidelines for Decision Making:

- If a choice improves or maintains both quality and efficiency, it is likely a good decision.
- If a choice degrades either metric, reconsider and explore alternatives.
- Use evidence-based metrics to guide decisions and ensure alignment with the goal of high-quality, efficient software development.

Choosing What Works Best:

- When a decision does not negatively impact quality or efficiency, use preference and familiarity to guide the choice.
- Always aim to align decisions with the fundamental goal of creating high-quality software efficiently.

Summary: The Real Tools of SWE

Interactive Question for Students

Question: How have you applied the concepts of stability and throughput in your decision-making processes in past projects?

Follow-Up Options:

1. Used stability to prioritize bug fixes over new features.
2. Focused on throughput to optimize delivery timelines.
3. Balanced both to ensure high-quality and efficient development.
4. Haven’t yet applied these concepts but interested in learning more.

Discussion:

- Invite students to share their experiences with using metrics like stability and throughput to guide project decisions. Discuss the challenges and benefits of making data-driven decisions and how these principles can be integrated into future projects for continuous improvement.

Questions?

Thank You.