ENSE701
Contemporary Issues in Software Engineering

Lecture Notes

Contemporary Issues in Software Engineering

Date	29/10/2024
Author	Jack Darlington
Student ID	19082592
Code	ENSE701
Paper	Contemporary Issues in Software Engineering
Version	1.0

Changelog

Date	Version	Notes
29/10/2024	1.0	Initial notes document created.

Table of Contents

Lecture 1: Software Engineering and Team Process

Course Vision

Purpose of the Course

- **Aim:** To build the knowledge, capabilities, and attitudes necessary to become a good software engineer.

Key Capabilities and Attitudes

- Ability to create, deploy, and maintain high-quality software that is valuable to users.
- Collaboration skills to work effectively in a team.
- Importance of these skills in meeting industry demand.

Tony's Views on Software Engineering

Reflecting Over Time

- Contributions to ACM Inroads magazine as a columnist and associate editor.
- Reflections on software development practices and education.

Software Engineering Tensions

- **Core Questions:**
    - Difference between programming-in-the-small vs. programming-in-the-large.
    - Is programming merely the implementation of a design?
- **Opposing Forces in Software Development:**
    1. A force for change based on an evolving vision driving the software process.
    2. Commercial need for certainty in cost and outcomes.
    3. Project management's focus on delivering against targets.
    4. Professional commitment to delivering quality software.

Feature-Driven Development (FDD)

- **Principles:**
    - Delivering scheduled releases on regular cycles.
    - Improved client relationships by allowing priority adjustments.
    - Constant delivery helps estimate feature costs and control project scope.
    - Time-boxing forces tough decisions and maintains project momentum.

Documentation in Agile Methods

- **Theory Building View:**
    - Programming as developing a shared "theory of the world" within the team.
    - Code is the primary artifact encapsulating this theory.
- **Role of Documentation:**
    - Secondary to the team's internalized understanding.
    - Necessary for communication and understanding.
    - Two primary reasons for documentation:
        - To communicate.
        - To understand.

Developing as a Professional Software Engineer

- Importance of understanding software development in a global context.
- Encouraging broader thinking beyond individual coding tasks.
- Reflecting on student conceptions of the discipline.

Dispositions and Agility

Agility as a Disposition

- **Core Principle from Agile Manifesto:**
  - "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."
- **Key Aspects of Agile Development:**
  1. Customer satisfaction.
  2. Delivery of working software.
  3. Provision of value.
- **Mindset and Attitudinal Dimensions:**
  - Agility involves how people are disposed to use their abilities.
  - Raises the question of whether dispositions can be taught or are innate.

Teaching Agility

- Importance of discretion and judgment in providing value to the client.
- Teams internalizing values to decide which tasks contribute to value.
- Recognizing that not all traditional deliverables make sense in every context.

Dispositions in DevOps Roles

- Dispositions differ from knowledge and skills.
- Emphasis on flexibility, adaptability, and collaborative mindset.
- Desired attributes include customer awareness, relationship management, communication skills, and respect in collaborative relationships.

Leadership Attributes

- **Transformational Leadership Roles:**
  - Mentor, Facilitator, Innovator, Broker.
  - Contrast with transactional roles focused on tasks.
- **Expectations:**
  - Taking responsibility for team members.
  - Training and mentoring others to develop new skills and capabilities.

Problem-Based Learning

- **Approach:**
  - Addressing real-world problems for clients through software products.
  - Team-based development using good practices and tools.
  - Learning driven by planning, doing, and reflecting.
- **Learning Objectives:**
  - Understanding theory and evidence for why and how to apply certain practices.
  - Building capabilities through practice and continuous learning from mistakes.
  - Reflecting on experiences to compare theory with practice.
  - Interacting with clients/product owners.
  - Developing products incrementally and iteratively using Agile methodologies.

Course Components and Expectations

What Work Do We Need to Do?

- **Process Flow:**
  - Understand what to build by collaborating with the client.
  - Build, deploy, and maintain the software.
  - Determine values and principles guiding work, interactions, and behaviors.
  - Decide on necessary documentation and planning.

How Will This Course Help?

- **Learning Outcomes:**
  - Familiarity with software engineering language and concepts.
  - Understanding values and principles guiding behaviors and work patterns.
  - Knowledge of practices and tools to address software engineering problems.
  - Skills to use tools and practices effectively.
  - Safe environment to ask questions, make mistakes, and learn.
  - Exposure to diverse options, experiences, and people.
  - Opportunities to teach others.
  - Application of theory and knowledge in team settings.
  - Engaging work that is challenging but rewarding.

Ways of Working and Tech Stack

Methodologies and Practices

- **Options Include:**
  - SEMAT (Software Engineering Methods and Theory).
  - Agile methodologies: Scrum, XP (Extreme Programming), Kanban.
  - Code craft, Waterfall (plan-driven), DevOps, DevSecOps.
  - Test-Driven Development, Pair Programming, Mob Programming.
  - Continuous Integration and Continuous Deployment.
  - Lean principles, User stories.
- **Tools to Support WoW:**
  - Development tools: Visual Studio, Git, GitHub.
  - Testing tools: JUnit, Cucumber, Selenium.
  - CI/CD tools: GitHub Actions, Travis CI.
  - Project management tools: Asana, Jira.
  - Deployment tools: AWS, Heroku, Docker, Puppet, Ansible, Maven.
  - Code quality tools: ESLint, Prettier, SonarQube.
- **Adaptation:**
  - Most projects use hybrid development methods tailored to their needs.

Focus Technologies

- **Programming Languages:** JavaScript/TypeScript.
- **Frameworks:**
  - **Backend:** Nest.js.
  - **Frontend:** Next.js.
- **Database:** MongoDB (optional).

Supporting Tools

- **Development Environment:** Visual Studio Code.
- **Code Quality:** ESLint, Prettier.
- **Testing:** Jest for end-to-end testing.
- **Version Control and CI/CD:**
  - Git and GitHub for code management.
  - GitHub Actions for automated builds, testing, and deployment.
- **Deployment:** Vercel for cloud deployment.

Lecture 2: Ways of Working in Software Engineering

Preparing for Team Development

- **Key Questions:**
  - What decisions, actions, and plans are needed to start developing and deploying features as a team?
  - What types of work need preparation?

Ideas and Approaches

- **Traditional Waterfall Approach:**
  - Linear phases: Requirements → Design → Build → Test → Deploy → Maintain.

Types of Work in Software Development

- **Common Work Areas:**
  - Understanding the problem.
  - Planning the work.
  - Designing the solution.
  - Executing the work.
  - Delivering/deploying the solution.
  - Maintaining/updating the work.
- **SEMAT (Software Engineering Methods and Theory):**
  - Focus on key elements like opportunity, stakeholders, requirements, software system, team, way of working, and work.
- **Agile Principles:**
  - Close collaboration with users and clients.
  - Shared product goal or vision.
  - Iterative and incremental development for rapid learning and feedback.
  - Frequent delivery of small value increments.
  - Breaking down problems and solutions into manageable parts.
- **User Stories and Backlogs:**
  - Creating user stories and story maps.
  - Maintaining a product backlog of requirements.
  - Setting iteration and sprint goals.

Engaging with the Client

Talking to the Product Owner (PO)

- **Goals:**
  - Understand the problem, why it's a problem, and desired changes.
  - Develop a product goal statement.
- **Principles:**
  - Ask open-ended questions.
  - Practice active listening.
  - Stay focused on the problem space.
  - Document discussions to form user stories.
  - Break down the problem into epic user stories.

Active Listening

- **Techniques:**
    - Fully engage and pay attention.
    - Reflect and paraphrase to confirm understanding.
    - Avoid interruptions and personal biases.

Continuous Integration and Deployment

Continuous Integration (CI)

- **Process:**
    - Developers modify code locally and commit changes to a local repository.
    - Push changes to a shared GitHub repository.
    - Automated tests and code checks run before merging.
    - Continuous integration happens every few hours.
- **Tools:**
    - Git and GitHub for version control and collaboration.
    - CI servers or services to automate testing and integration.

Continuous Deployment (CD)

- **Overview:**
    - Automates the release process to deploy code to production.
    - Ensures frequent and reliable deployment of updates.

Requirements Engineering in Agile

- **Iterative Approach:**
    - Requirements are refined incrementally.
    - Emphasis on flexibility and adapting to change.

Historical Perspective

Iterative Enhancement (1975 and Earlier)

- **Concept:**
    - Start with a simple implementation of a subproblem.
    - Use a project control list to track tasks and progress.
    - Iteratively enhance the implementation until completion.

Agile Requirements Refinery

- **Application in Software Product Management:**
    - Applying Scrum principles to manage and refine requirements.
    - Emphasis on collaboration and adaptability.

DevOps as a Way of Working (WoW)

- **Values:**
    - Frequent deployment and release.
    - Team ownership of the product lifecycle.
    - Accountability for deployment and post-release performance.
- **Enablers:**
    - Test automation and CI/CD automation.
    - Infrastructure as code.

- Monitoring and alerting systems.
- Practices like rollback strategies, feature flags, and microservices.

Technology Stacks and Full-Stack Development

Importance of a Tech Stack

- **Considerations:**
  - Aligns with project goals and team expertise.
  - Impacts scalability, maintainability, and performance.

MNNN Stack Overview

- **Components:**
  - **MongoDB:** NoSQL document database.
  - **Nest.js:** Backend framework built on Node.js.
  - **Next.js:** React framework for server-rendered apps.
  - **Node.js:** JavaScript runtime environment.
- **Architecture:**
  - Layered structure separating concerns.
  - Supports scalable and maintainable applications.

Alternative Stack: MERN

- **Components:**
  - **MongoDB, Express.js, React.js, Node.js.**
  - Popular for full-stack JavaScript applications.

Agile Values and Practices

Agile Manifesto

- **Core Values:**
  - Individuals and interactions over processes and tools.
  - Working software over comprehensive documentation.
  - Customer collaboration over contract negotiation.
  - Responding to change over following a plan.
- **Principles:**
  - Emphasis on collaboration, flexibility, and delivering value.

Modern Agile

- **Key Principles:**
  - Make people awesome.
  - Deliver value continuously.
  - Experiment and learn rapidly.
  - Make safety a prerequisite.

Heart of Agile

- **Focus Areas:**
  - **Collaborate:** Work together effectively.
  - **Deliver:** Provide value frequently.
  - **Reflect:** Regularly evaluate processes and outcomes.
  - **Improve:** Continuously enhance practices and products.

Mindset and Culture

Agile Mindset

- **Approach to Uncertainty:**
  - Embrace change as an opportunity.
  - Use iterative cycles for adaptation and learning.

Growth Mindset

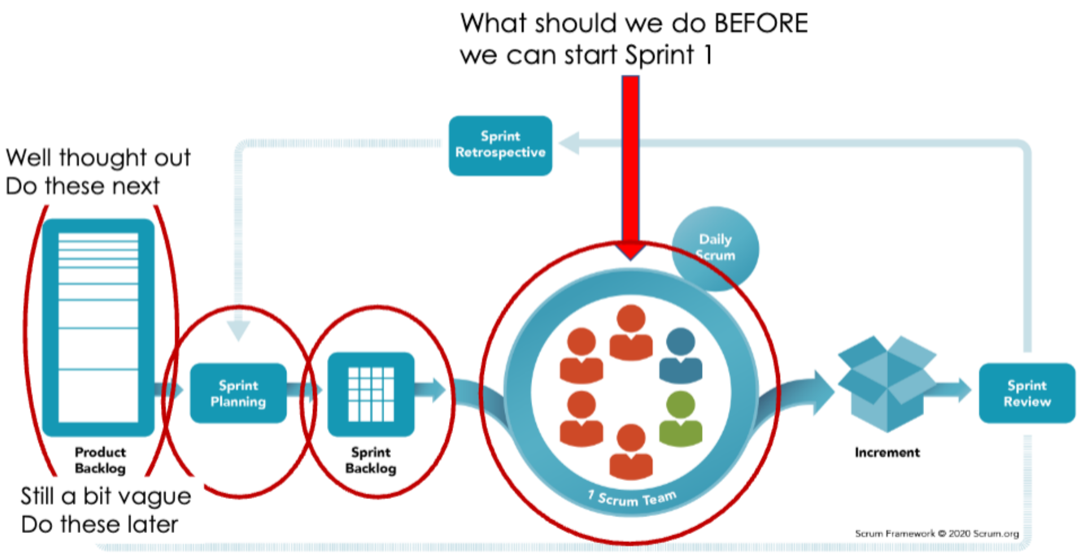
- **Concept:**
  - Belief in the ability to develop skills through effort and learning.
  - Aligns with the continuous improvement aspect of Agile.

Evolution of Software Engineering

- **High-Level Evolution:**
  - Shift from rigid, plan-driven methods to flexible, iterative approaches.
  - Increased focus on collaboration, customer involvement, and adaptability.

Lecture 3: User Stories & Story Maps

Scrum/Iterative Workflow


What should we do BEFORE we can start Sprint 1
Well thought out Do these next
Still a bit vague Do these later
Sprint Retrospective
Daily Scrum
Sprint Planning
Sprint Backlog
Product Backlog
1 Scrum Team
Increment
Sprint Review
Scrum Framework © 2020 Scrum.org

Scrum Workflow

- **What should we do BEFORE we can start Sprint 1?**
  - Well-thought-out items: **Do these next**
  - Still a bit vague: **Do these later**

Introduction on How to Write User Stories

- Perspectives and purposes:
  - Design Lead Developer
  - Client or Product Owner
  - Sales Manager
  - Business Analyst
- There are many perspectives and purposes.

Pre-Sprint – User Requirements v1

- **What do users want to be able to do to achieve the product goal?**
- **User Stories:**
  - As a `<user type>`, I want to `<do something>` so that `<benefit>`.
- **Product Goal (Outcome wanted):**
  - Make it front of mind for the team.
- **Business Problem or Opportunity:**
  - What needs to change?
  - Why is this product worth doing?

User Stories Part 1

- **Placeholders for Conversations**

- **Significant User Type:**
  - Characteristics that distinguish from other user types that may affect the design.
  - Not just "User".
- **Capability:**
  - The new capability the user wants to reach a smaller goal (outcome).
  - In business language; no solution details.
- **Goal:**
  - What is the desired outcome of this new user capability?

Example User Story:

- As a **researcher**, I want to be able to **recommend articles to include in the evidence repository** so that **the evidence available keeps expanding**.
- **Additional Insights:**
  - I want people to be able to add new evidence so the repository gets bigger and leverage crowd sourcing.
  - As a practitioner, I want lots of evidence to be available so the evidence is convincing.
  - As the Product Owner, I want to have lots of articles in the evidence repository.
  - We should check:
    - The article is about SE with evidence (relevance).
    - The article is not already in the repository (avoid duplicates).
    - The quality of the evidence in the article (quality).
  - The submitter should be informed/thanked if the article they submitted is accepted or not (and why not).

Discovering Product Goal and User Needs

- Knowledge about the problem, user needs, and possible solutions is distributed unevenly—especially at the start.
- **There are many perspectives on the problem and user needs:**
  - Lead Developer
  - Client or Product Owner
  - Sales Manager
  - Business Analyst
- **Different worldviews and language**
- More detailed user stories with at least one acceptance criterion.
- Keep breaking down until you expect that the user story will take 1–2 ideal days and definitely less than 1 sprint.
- Work out technical tasks that need to be done on the next set of user stories to be worked on.
- Often tasks are associated with user stories.
- Avoid technical "user stories".

Order Amount	VIP Member	Number of Items in Order	Free Freight	Order Discount
<$100	Yes	>=10	No	Yes
>=$100	Yes	<10	Yes	No
<$100	No	>=10	No	No
>=$100	No	<10	No	No

Discovering User Stories (User Requirements)

- **User Story Workshops:**
  - Product stakeholders, Product Owner, Business Analyst, Developer, Tester, etc.
  - Stakeholders write them and group (and agree on order or in/out).

- **Methods:**
  - Interviewing users or the Product Owner.
  - Observation, surveys, impact mapping, customer journey mapping.
- **Approaches:**
  - Big upfront effort vs. Iterative and incremental effort.
    - Plan-driven control based on high-confidence, long-term predictions.
    - Frequent opportunity for changes based on empiricism and short learning loops.
- **Mapping = Visualizing information showing relationships and structure.**
- Helps to make sense of something and acts as an information radiator.
- **Types of Mapping:**
  - Customer Journey Maps
  - Impact Mapping
  - Value Stream Mapping
- Explore these methods yourself.

Customer Journey Maps

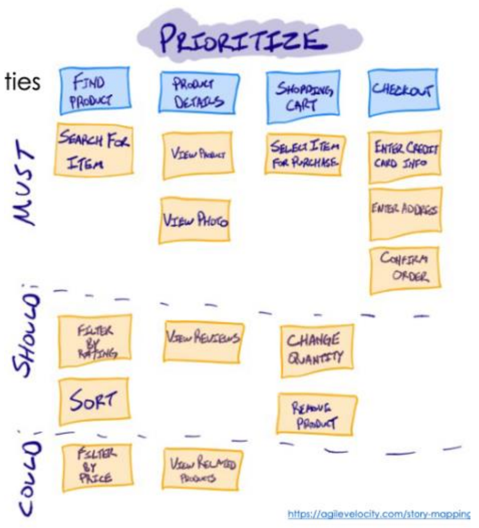
- **Definition:**
  - A visual representation of the important steps a customer goes through to achieve a goal with your company.
- **Benefits:**
  - Understand customers' motivations, needs, and pain points.
- **Quote:**
  - "A successful customer journey map will give you real insight into what your customers want and any parts of your product, brand, or process that aren't delivering." — David Weaver

Characteristics of a Good User Story and How to Write Them

- **Independent:** Minimize dependencies that require coordination.
- **Negotiable:** Not a contract; open to discussion.
- **Valuable:** Written from the user's perspective in non-technical language.
- **Estimable:** Can be estimated for effort and time.
- **Small:** Should be small enough to be completed within a sprint.
- **Testable:** Must have acceptance criteria to verify completion.

Story Maps are the New Product Backlog

- **Story Mapping:** Structuring product features (as user stories) to model the entire product.
- **Advantages Over Product Backlog:**
  - Avoids shortcomings of a prioritized list.
  - Useful for planning delivery product increments and sprint planning.
  - Supports upfront product design and work prioritization.
  - Helps share and develop understanding.



- Provides context for features to understand relationships between features, user types, and their purpose.

Using a Story Map to Slice Out a Delivery Plan or Sprint Plan

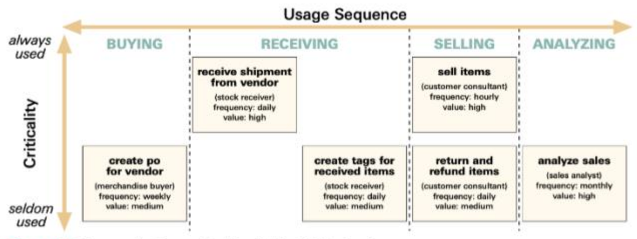

Usage Sequence
always used
Criticality
seldom used
BUYING
RECEIVING
SELLING
ANALYZING
receive shipment from vendor (stock receiver) frequency: daily value: high
sell items (customer consultant) frequency: hourly value: high
create po for vendor (merchandise buyer) frequency: weekly value: medium
create tags for received items (stock receiver) frequency: daily value: medium
return and refund items (customer consultant) frequency: daily value: medium
analyze sales (sales analyst) frequency: monthly value: high

Figure 5: **The model is vertically divided into business processes.**

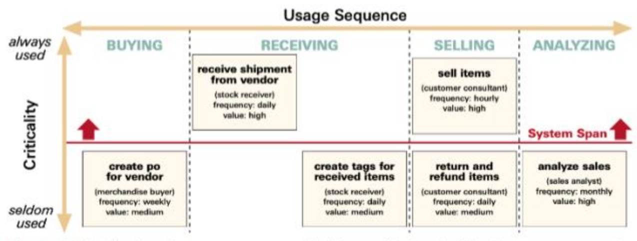
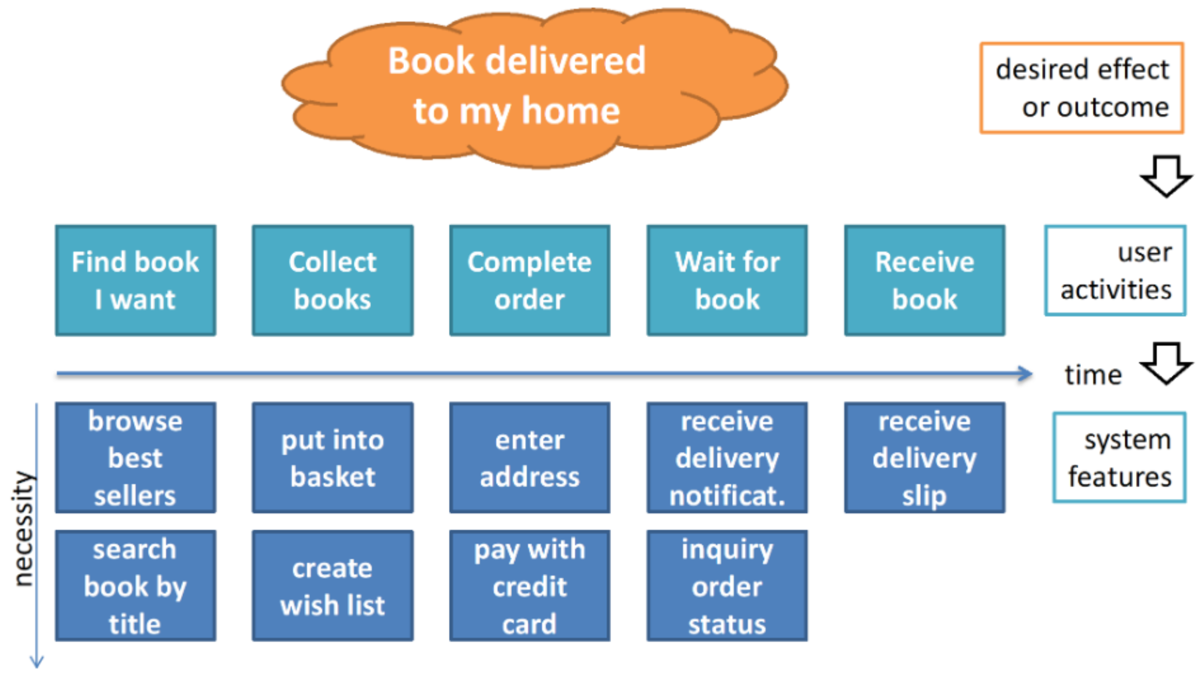

Usage Sequence
always used
Criticality
seldom used
BUYING
RECEIVING
SELLING
ANALYZING
receive shipment from vendor (stock receiver) frequency: daily value: high
sell items (customer consultant) frequency: hourly value: high
System Span
create po for vendor (merchandise buyer) frequency: weekly value: medium
create tags for received items (stock receiver) frequency: daily value: medium
return and refund items (customer consultant) frequency: daily value: medium
analyze sales (sales analyst) frequency: monthly value: high

Figure 6: **The first system span represents the smallest set of features necessary to be minimally useful in a business context.**

Anatomy of a Story Map

- **Columns:** Relate to user activities.
- **Horizontal Slices:** Group by priority, importance, or frequency of use.
- Each column has system features related to the user activity.
- Create horizontal slices to represent the scope of delivery cycles or sprint cycles.



Story Map Example

User Story Success Criteria

- A list of rules or criteria that should be met for the story to be successful.
- Also known as acceptance criteria, acceptance tests, or conditions of satisfaction.
- **Common Template:**
    - **Given** [initial context], **When** [event occurs], **Then** [outcome].
- Focus on understanding the business value from the user's perspective.
- Delays the temptation for developers to jump to solutions before understanding the problem.
- Helps the user (Product Owner) clarify what they want.
- Makes it easy to verify requirements and ensures shared understanding.

User Stories in a Nutshell

- **User Stories:**
    - Defer detail until the development team needs it.
    - Act as a reminder to have conversations and ask questions.
- **Short Feedback Cycles:**
    - Development team learns from doing.
    - Product Owner learns from seeing.
    - Requirements emerge and change.

Skills and Mindset Needed

- Writing effective user stories.
- Deferring detail appropriately.
- Asking questions to get necessary detail.
- Writing acceptance tests and conditions of satisfaction.
- Splitting user stories as needed.
- Understanding user value and creating personas.
- Managing changes and estimation.

Be Sensitive to Your User Task's "Altitude"

(Adapted from Alistair Cockburn's Writing Effective Use Cases)

- **Functional or "Sea Level":**
  - Tasks expected to be completed in a single sitting.
- **Sub-Functional or "Fish Level":**
  - Small tasks that don't mean much alone; several are done before reaching a functional goal.
- **Activity or "Kite Level":**
  - Longer-term goals with no precise ending; several functional tasks performed in this context.

Three Key Pieces of Information in User Stories

2. **Primary Actor/User Role/Persona:**
   - Who is the requirement for?
3. **Action/Feature/Goal:**
   - What is the new functionality the user wants?
4. **Reason/Benefit:**
   - Why does the user want this requirement? What is the expected benefit?

Examples of User Stories

- As a **registered user**, I want to **cancel my order** so that **I can change my mind**.
- As a **job seeker**, I want my **CV to be easily available to employers** so that **I maximize my chances of being offered a job**.

Software Requirements are a Communication Challenge

- User stories are placeholders for requirements.
- Analysis spans the entire lifetime of the deliverable.
- They are complete units that can be built, tested, and delivered.
- **Benefits of Getting User Stories Right:**
  - Set boundaries for each work item.
  - Unit for planning sprints and estimation.
  - Keep track of what is still to be done and in what order.
  - Basis for testing, monitoring progress, change management, and division of labor.
- **Task-Oriented:** Create a database to store job seekers' CVs.
- **User Story-Oriented:** As a **job seeker**, I want my **CV to be easily available to employers** so that **I maximize my chances of being offered a job**.

Breaking Down User Stories

- **Why?**
  - For planning, estimation, independent work splitting, and testing.
- **How?**
  - Workflow steps
  - Business rule variations
  - Major effort extraction
  - Simple vs. complex versions

Developer Tasks to Create Features

- **Why?**

- To estimate effort.
- To involve the right technical specialties.
- To plan development in the sprint.

- **Examples:**
  - Create database schema.
  - Integrate with backend services.
  - Develop frontend components.

Self-Organizing Teams in Agile

- The team agrees on:
  - Acceptance criteria (conditions of satisfaction).
  - Proposed solution approach.
  - Estimate of effort.
- **Do we need a team leader?**
  - Teams are self-organizing; leadership can be shared.

Creating User Stories

- Anyone can write user stories—team members, Product Owner, stakeholders.
- The Product Owner ensures a Product Backlog exists and is ordered.
- Everyone should be involved in discussions about user stories.
- Acceptance criteria should be written as part of the user story.
- Avoid speculating or imagining what users want.
- If you don't know who the users and customers are and why they would use the product, conduct necessary user research first (e.g., observations, interviews).
- Involve the team and stakeholders.
- Use workshops and collaborative sessions.
- Ensure shared understanding and buy-in.

Personas Lead to the Right Stories

- **Define Personas:**
  - Create detailed profiles of typical users.
- **Ask:**
  - What functionality should the product provide to meet the goals of the personas

Adding Detail to a User Story

- **When?**
  - When it is near the top of the Product Backlog.
- **How?**
  - Breaking stories down into smaller pieces.
  - Adding acceptance tests or criteria.

Acceptance Criteria as Details

- **Conditions of Satisfaction:**
  - Specific requirements that must be met for the story to be considered complete.
- **Example:**
  - As a registered user, I want to cancel my order so that I can change my mind.
    - Acceptance Criteria:

- The user can cancel an order within 24 hours.
- A confirmation email is sent upon cancellation.

Non-Functional Requirements

- Performance, security, usability, etc., that are critical to the product's success.
- Should be considered when writing user stories and acceptance criteria.

User Stories About UI

- Focus on the user's interaction with the system.
- May include mockups or wireframes to illustrate the desired interface.

Breaking Down User Stories

- **Techniques:**
  - By workflow steps.
  - By business rules.
  - By major effort areas.
  - By simple vs. complex implementations.

(Refer to Leffingwell, D. (2011). Agile Software Requirements*, p. 135–136.)*

Definition of Ready

- **Criteria for a User Story to be Ready:**
  - Shared understanding among key stakeholders.
  - Conveys the opportunity, issue, need, or value.
  - Title is short and conveys the deliverable.
  - Delivers an atomic increment in business value.
  - Can be completed in a sprint.
  - Has sufficient acceptance criteria.
  - The team can estimate the story.

INVEST Checklist for User Stories

- **Independent**
- **Negotiable**
- **Valuable**
- **Estimable**
- **Small**
- **Testable**

(From INVEST in Good Stories by Bill Wake)

Definition of Done

- **Criteria for Completion:**
  - All unit, integration, and acceptance tests are passed.
  - Code is submitted to the repository and reviewed.
  - The feature has been deployed to the appropriate environment.
  - Documentation is completed and uploaded.
  - User Acceptance Testing (UAT) is completed.

  - Any required approvals are obtained.

Lecture 4: Preparing to Iterate/Sprint

Agile Planning Levels

- **Reference:** Lal, R., & Clear, T. (2021). Three Levels of Agile Planning in a Software Vendor Environment.
- **Agile Planning:**
  - Emphasizes planning at multiple levels.
  - Adaptation based on team characteristics and project specifics.
  - Regular reflection and adjustment of working conventions.

The Agile Manifesto – Many Ways of Working (WoW)

- **Key Points:**
  - Agile is a mindset with various methodologies.
  - Teams should personalize their methodology.
  - Iterative cycles of behavior and continuous improvement.

Iterative Development Process

- **Pre-Iteration Activities:**
  - Create initial Product Backlog and Story Map (acknowledging uncertainty).
  - Develop user stories with acceptance criteria.
  - Detail understanding and design features for the next iteration only.
  - Set up architecture, tech stack, deployment, and development environment.
  - Plan for Iteration 1: Define goals and iteration backlog.
- **During Iterations:**
  - Coordinate work and maintain team alignment.
  - Regular team meetings and feedback sessions.
  - Review product increments and team processes.
  - Manage changes to requirements and risks.
  - Decide on iteration content through planning meetings.
  - Assure quality using CI/CD, pair programming, and TDD.

User Stories as Documentation

- **Purpose of User Stories:**
  - Document user requirements.
  - Serve as units of work for:
    - Understanding user needs.
    - Splitting up work.
    - Designing product features.
    - Testing product features.
    - Organizing work order.
    - Planning iterations.
    - Monitoring progress.

Lifecycle of User Stories

5. **Discover User Needs:**
   - Use various techniques (interviews, observations, workshops).
6. **Write High-Level User Stories (Epics):**

   - Keep them in the Product Backlog and User Story Map.
   - Define the Product Goal.
7. **Break Down into Smaller User Stories:**
   - Include acceptance criteria, success criteria, and Definition of Done.
8. **Decide on Work Order:**
   - Prioritize stories in the Product Backlog.
9. **Plan Iterations:**
   - Estimate team capacity and story sizes.
   - Select user stories for the next iteration.
10. **Develop and Monitor:**
    - Translate stories into design and code.
    - Monitor progress and adjust as needed.

INVEST Checklist for User Stories

- **Independent:** Can the story stand alone?
- **Negotiable:** Open to change and discussion.
- **Valuable:** Provides value to the end user.
- **Estimable:** Can estimate the size and effort.
- **Small:** Sized appropriately for completion in a sprint.
- **Testable:** Can be tested and verified.

Definition of Done

- **Criteria to Consider a User Story Complete:**
  - All tests passed (unit, integration, acceptance).
  - Code reviewed and submitted to the repository.
  - Feature deployed to the appropriate environment.
  - Documentation completed and available.
  - User Acceptance Testing (UAT) completed.
  - Any required approvals obtained.

Using Story Maps for Planning

- **Purpose:**
  - Structure product features to model the entire product.
  - Provide context and relationships between features, user types, and their purposes.
- **Anatomy of a Story Map:**
  - **Columns:** User activities.
  - **Horizontal Slices:** Priority, importance, or frequency of use.
  - Features aligned under user activities.
  - Slices can represent scopes like MVP or sprint cycles.
- **Tools:**
  - Miro and other digital platforms for creating story maps.

User Story Success Criteria

- **Acceptance Criteria:**
  - Define rules, tests, or behaviors for story success.
  - Common template: **Given** [context], **When** [event], **Then** [outcome].
- **Techniques:**

  - Behavior-Driven Development (BDD).
  - Acceptance Test-Driven Development (ATDD).

Pre-Sprint Planning

Big Picture Planning

- **Roadmap:**
  - Define sprint durations and goals.
  - In this case: 3 x 9-day sprints with specific sprint goals.
- **Release Plan:**
  - Deploy increments at the end of each sprint.
- **Focus:**
  - Delivering value to users.
  - Refining product and sprint goals.

Selecting Work for the First Sprint

- **Estimation:**
  - Estimate story sizes using relative methods (e.g., story points, T-shirt sizes).
  - Determine team capacity (velocity) for the sprint.
- **Planning:**
  - Select user stories from the top of the Product Backlog until capacity is reached.
  - Aim for small, similarly sized stories for consistency.

Software Project Estimation Issues

- **Challenges:**
  - Inconsistencies in estimation methods and findings.
  - Difficulty in providing reliable advice to practitioners.
- **Examples of Inconsistent Findings:**
  - Model-based vs. expert-based estimation studies showing conflicting results.
  - Regression vs. analogy methods yielding varied outcomes.
  - Cross-company data vs. local data in predictions.

Estimation in Agile Software Development

- **Findings:**
  - Story Points are commonly used for sizing in Agile teams.
  - Collective estimation helps in reaching team consensus and reducing biases.
- **Mobile App Development:**
  - Factors like the number of screens and supported platforms affect estimation.
  - Expert judgment and function size measurement are frequently used techniques.

Working with the Client: Useful Questions to Ask

11. What business problem are you trying to solve?
12. What's the motivation for solving this problem?
13. What would a highly successful solution do for you?
14. How can we judge the success of the solution?
15. Which business activities and events should be included or excluded?
16. Can you foresee any unexpected or adverse consequences?
17. What's a successful solution worth?
18. Who are the stakeholders that could influence or be influenced by this project?

19. Are there related projects or systems that could have an impact?

Monitoring Progress During a Sprint

- **User Story Board:**
    - Visual tool to track the status of user stories.
    - Columns represent different stages (e.g., To Do, In Progress, Testing, Done).
- **States of User Stories:**
    - Developing
    - Testing
    - Deployment
    - Done
- **Tools:**
    - Physical boards or electronic tools like Trello, Asana, Jira, Azure DevOps.

Pre-Sprint Design and Architecture

- **Non-Functional (Quality) Requirements:**
    - Consider performance, scalability, security, etc.
- **Tech Stack Example:**
    - Front-end: NEXT.js
    - Back-end: Nest.js on Node.js
    - Database: MongoDB

Working as an Individual in a Team

- **Workflow:**
    - Continuous Integration (CI) practices.
    - Adherence to coding standards and code quality.
- **Non-Functional Requirements:**
    - Focus on aspects like usability and performance.

Team Collaboration

- **Building a High-Performing Team:**
    - Build trust and share expectations.
    - Coordinate work and goals frequently.
    - Reflect on team processes and improve.
    - Foster a safe environment.
- **Practices:**
    - Break work into manageable pieces.
    - Use mob programming and collective problem-solving.
    - Regularly check progress with users or the Product Owner.

Risk Planning

- **Identifying Risks:**
    - Not meeting quality or goals.
    - Misalignment with Product Owner expectations.
    - Unused or irrelevant product features.
    - Learning new technologies taking longer than expected.
- **Mitigation Strategies:**

- Define quality and goals clearly.
- Ensure code is maintainable and robust.
- Share critical knowledge within the team.

Lecture 5: Requirements Prioritization in Scaled Agile Distributed Software Development

Understanding Requirements

- **Definition:**
    - Focus on **what** is needed, not **how** it will be implemented.
    - Address user needs to solve a problem or achieve an objective.
    - **IEEE Definition:** A condition or capability needed by a user to solve a problem or achieve an objective.
- **Importance:**
    - Requirements capture the essential needs of users.
    - Serve as a foundation for system development.

Requirements Engineering (RE)

- **Definition:**
    - A branch of Software Engineering focused on discovering, specifying, analyzing, and documenting system requirements.
- **RE Processes:**
    1. **Requirements Elicitation, Analysis, and Prioritization**
        - Gathering requirements from stakeholders.
        - Understanding and analyzing needs.
        - Prioritizing based on importance and feasibility.
    2. **Requirements Specification**
        - Documenting requirements in a clear and detailed manner.
    3. **Requirements Validation**
        - Ensuring requirements accurately reflect stakeholder needs.
    4. **Requirements Management**
        - Handling changes and maintaining consistency over time.

RE in Software Development Methods

Classic Development Methods (e.g., Waterfall)

- Requirements are fully defined and documented before development begins.
- Emphasizes thorough planning and documentation.

Agile Development (e.g., Scrum)

- Requirements evolve iteratively throughout the development process.
- Emphasizes flexibility, collaboration, and customer involvement.

Requirements Prioritization in Scaled Agile Distributed Development

- **Context:**
    - Adoption of Agile methods at the enterprise level.
    - Use of scaling Agile frameworks (e.g., Disciplined Agile Delivery (DAD), Scaled Agile Framework (SAFe)).
- **Importance:**

  - Prioritization extends beyond individual teams.
  - Involves multiple decision-making levels to align with business strategy.

Key Decision-Making Levels

20. **Portfolio Level**
21. **Domain/Program Level**
22. **Team Level**

Decision-Making at Different Levels

Portfolio Level

- **Objectives:**
  - Define and prioritize business goals aligned with the organization's strategy.
  - Formalize High-Level Requirements (HLRs) to achieve these goals.
  - Allocate HLRs to engineering for implementation.
- **Practices:**
  - **Inter-Iteration Prioritization:** Prioritize across multiple iterations.
  - **Portfolio Management Team:** Cross-functional team for decision-making.
  - **Continuous Decision-Making:** Regular check-ins (quarterly, monthly, fortnightly) and ad-hoc meetings.
  - **Quantitative & Qualitative Approaches:** Use data and expert judgment.

Domain/Program Level

- **Role:**
  - Acts as a bridge between portfolio and team levels.
  - Handles domain-specific requirements.
- **Responsibilities:**

  First Part:

  - Initial decision-making on HLRs from each domain.
  - Contributes to forming strategic themes.
  - Supported by **Intra-Iteration Prioritization**.

  Second Part:

  - **Inter-Iteration Prioritization** across teams.
  - Decompose HLRs into implementable tasks for short development cycles.

Team Level

- **Focus:**
  - Detailed prioritization within the team.
  - Utilize basic Agile methods for decision-making (e.g., Scrum).
- **Practices:**
  - **Intra-Iteration Prioritization:** Adjust priorities within sprints.
  - Create detailed requirements (e.g., user stories).
  - Make priority decisions collaboratively within the team.

Boundary Spanning Mechanisms

- **Definition:**
  - Activities that connect groups separated by location, hierarchy, or function.
- **Purpose:**

    - Facilitate knowledge sharing, negotiation, consensus building, and conflict resolution.
    - Essential for effective prioritization across distributed teams.
- **Mechanisms:**
    1. **Boundary Spanning Events:**
        - Meetings, workshops, and planning sessions.
    2. **Boundary Spanning Artefacts:**
        - Shared documents, models, and tools.
    3. **Boundary Spanner Roles:**
        - Individuals who act as liaisons (e.g., product owners, scrum masters).

Collaborative Technologies (CTs) in Decision-Making

- **Purpose:**
    - Enable collaboration among distributed team members.
    - Support shared understanding of requirement priorities.
- **Types of CTs:**

  Synchronous Tools:
    - Real-time communication.
    - Examples: MS Teams, video conferencing.

  Asynchronous Tools:
    - Communication over time.
    - Examples: Email, Confluence, Jira.
- **Common Tools:**
    - Spreadsheets
    - Jira
    - AHA!
    - Azure DevOps (ADO)
    - Confluence
    - PowerPoint
    - MS Teams
    - Email

Usefulness of CTs During Decision-Making

- **Benefits:**
    - Enhance collaboration across different locations and time zones.
    - Provide platforms for documenting decisions and tracking progress.
    - Support both real-time and delayed communication, catering to various needs.
- **Challenges:**
    - Ensuring all team members are proficient with the tools.
    - Managing information overload.
    - Maintaining data security and privacy.

Lecture 6: Ethics and Professionalism in Software Engineering

Why Become Aware of Ethical and Professional Responsibilities?

- **Case Study: Surveillance Technology**
  - Task: Develop software to identify unauthorized items on a desk.
  - Ethical considerations in AI and object detection (e.g., using YOLO).
- **Discussion:**
  - Ethical implications of surveillance in the workplace.
  - Reference to article: Big Tech Call Center Workers Face Pressure to Accept Home Surveillance

Software Engineering Graduates Expectations

- **SE2004 Recommendations:**
  4. Mastery of software engineering knowledge and skills.
  5. Ability to work individually or in a team to develop quality software.
  6. Making appropriate trade-offs considering constraints.
  7. Perform design integrating ethical, social, legal, and economic concerns.
  8. Understand and apply current theories and techniques.
  9. Demonstrate interpersonal, leadership, and communication skills.
  10. Learn new models, techniques, and technologies as they emerge.

Intellectual Property and Copyright with SPEED

- **Scenario:**
  - Users can suggest articles for the SPEED database.
  - Submitters provide bibliographic details only (no PDFs) due to copyright.
  - Ethical handling of intellectual property.

Lecture Outline

23. Power and Technology
24. Professional Responsibility
25. Ethics - What & Why
26. Philosophical Ethics
27. Professional Ethics
28. Codes of Ethics
29. Conflict of Professional Responsibility
30. Project Risk & Software Development Impact Statements
31. AI and Ethics
32. Further Considerations

Information Technology Perspectives

- **Multiple Discipline Perspectives:**
  - **Computational View:** Technology as algorithm (Computer Science).
  - **Tool View:** Technology as labor substitution or productivity tool (Commercial Perspective).
  - **Proxy View:** Technology as perception or diffusion (Information Systems).
  - **Ensemble View:** Technology as development project, production network, embedded system, and structure.

Power and Technology

- **Quotes:**
  - Designers have significant power in specifying systems.
  - Decisions in design can reduce or eliminate action alternatives.
- **Implications:**
  - Systems can extend managerial power and control.
  - The way technology is appropriated can be unpredictable.

Professional Responsibilities

- **Computing Professionals:**
  - Actions change the world; must reflect on wider impacts.
  - Should consistently support the public good.
- **ACM Code of Ethics:**
  - Expresses the conscience of the profession.
  - ACM Code of Ethics

What is Ethics?

- **Definitions:**
  - **Society:** Ordered community of people.
  - **Morality:** Division between right and wrong actions.
  - **Ethics:** Standards of what is believed to be right and wrong.
- **Why Study Ethics?**
  - Address new problems (cyberbullying, privacy, spam).
  - Common sense may not be adequate.

Cyberethics

- **Unique Ethical Challenges:**
  - Policy vacuums due to malleability of computers.
  - Evolving issues with technological advancements.
- **Evolution Phases:**
  - Stand-alone machines to interconnected networks.
  - Emergence of AI and autonomous systems.

Philosophical Ethics

- **Ethical Theories:**
  - **Consequentialism:** Actions judged by outcomes.
    - **Egoism:** Individual's interests above all.
    - **Utilitarianism:** Greatest good for the greatest number.
    - **Altruism:** Actions favorable to all except the actor.
  - **Deontology:** Actions judged by adherence to rules or duties.
- **Ethical Mindset:**
  - "The unexamined life is not worth living." — Socrates
  - Eudaimonism: Pursuit of a fulfilling and virtuous life.

Moral Dilemmas and AI Ethics

- **Crowd Ethics:**
  - Platforms like Moral Machine explore ethical decisions in autonomous systems.

- **Challenges:**
  - Programming ethics into AI.
  - Issues with machine learning biases.

Professional Ethics

- **What is a Profession?**
  - Requires expert knowledge, autonomy, and a code of conduct.
  - Professionals have significant social effects.
- **Codes of Ethics:**
  - Provide guidelines for professional behavior.
  - Examples: Medical (NZMA), Engineering (REA), Legal (NZLS), IT (IT Professionals NZ).

IT Professionals New Zealand (ITP) Code of Ethics

- **Eight Tenets:**
  1. Good faith
  2. Integrity
  3. Community focus
  4. Skills
  5. Continuous development
  6. Informed consent
  7. Managed conflicts of interest
  8. Competence

IEEE-CS Software Engineering Code of Ethics

- **Levels of Ethical Obligations:**
  - **Level One:** Ethical values shared by all humans.
  - **Level Two:** Obligations owed by professionals.
  - **Level Three:** Specific responsibilities unique to software engineering.
- **Principles:**
  1. Public
  2. Client and Employer
  3. Product
  4. Judgment
  5. Management
  6. Profession
  7. Colleagues
  8. Self

ACM Code of Ethics

- **General Ethical Principles:**
  1. Contribute to society and human well-being.
  2. Avoid harm.
  3. Be honest and trustworthy.
  4. Be fair and take action not to discriminate.
  5. Respect property rights.
  6. Respect privacy.
  7. Honor confidentiality.
- **Professional Responsibilities:**
  - Strive for high quality, maintain competence, respect rules, and promote public awareness.

- **Leadership Principles:**
    - Ensure public good is central, manage personnel ethically, and support policies reflecting ethical principles.

Strengths and Weaknesses of Professional Codes

- **Strengths:**
    - Inspire ethical behavior, guide decisions, educate members, discipline violations, inform the public, sensitize to issues, enhance professional image.
- **Weaknesses:**
    - Directives can be vague or conflicting, not exhaustive, may lack enforcement, and sometimes serve self-interest.

Conflict of Professional Responsibility

- **Whistle-Blowing:**
    - Exposing illegal or unethical activities within an organization.
    - Ethical dilemma between loyalty to employer and public interest.
- **Case Study:**
    - Challenger Disaster (1986)
        - Engineers aware of safety risks but were overruled.
        - Should they have gone public?
- **Guidance:**
    - Professional codes offer some direction but may be vague.
    - De George's criteria for whistle-blowing:
        - Morally permitted if harm is serious, concerns reported internally, internal channels exhausted.
        - Morally obligated if there's evidence and belief that going public will bring change.

Project Risk and Software Development Impact Statements (SoDIS)

- **Purpose:**
    - Identify and analyze potential risks in software projects, including ethical concerns.
- **Case Studies:**
    - **TRAION Project:**
        - Developing systems for a Māori tribal authority.
        - Addressed cultural sensitivities regarding genealogical information.
    - **Baptist Action Trust (BAT):**
        - Automating healthcare rostering and billing.
        - SoDIS inspection identified clusters

Lecture 7: Quality Assurance, Testing, and Test-Driven Development (TDD)

What is Quality Assurance?

- **Definition:**
  - A planned and systematic approach to evaluating software quality.
  - Ensures adherence to software product standards, processes, documentation, and procedures.
- **Focus Areas:**
  - Code design.
  - Product/Application.

Key Aspects of Quality

Perspectives on Software Quality

- **Transcendental View:** Quality is something that can be recognized but not defined.
- **User View:** Quality as fitness for purpose.
- **Manufacturing View:** Quality as conformance to specification.
- **Product View:** Quality tied to inherent characteristics of the product.
- **Value-Based View:** Quality depends on the amount a customer is willing to pay.

Quality Assurance Process

- **Quality in Learning:**
  - Learning as a transformative process.
  - Need for a quality process to ensure predictable and high-quality outcomes.
- **Roles and Responsibilities:**
  - Allocate roles to appropriately skilled team members.
- **Measurement and Standards:**
  - Use of metrics to demonstrate conformance to standards.
  - Example standards: ISO 9126 (functionality, reliability, efficiency, usability, maintainability, portability).

Metrics and Testing

- **Metrics:**
  - Provide a yardstick to demonstrate conformance or highlight lack of it.
  - Examples include standards for coding, document formatting, recording meeting minutes.
- **Testing:**
  - Considered a quality control (QC) activity, part of the production function.
  - A multi-layered testing strategy is key (unit tests, integration tests, usability tests, performance tests, etc.).

Reviews and Continuous Processes

- **Quality Reviews:**
  - Inspections, walkthroughs, formal technical reviews.
  - Code and design inspections.
- **Continuous Processes:**
  - Pair programming.
  - Joint Application Development (JAD) workshops.

  - Test-Driven Development (TDD).
  - Continuous integration, regular code builds, refactoring.
- **Continuous Improvement:**
  - Software Process and Practice Improvement (SPPI).
  - Reflect and adapt processes for future improvements.

Quality Assurance Accountability

- **Team Agreement Document:**
  - Commitments to one another.
  - Communication strategies.
  - Team roles and accountabilities (including QA for each iteration).
- **QA Accountability Reflection:**
  - Explanation of the team's emphasis on QA for the iteration.
  - Reflection on success and areas for improvement.
  - Plans for the next iteration.

Evolving Views of Software Quality

- **Debate on Software Quality:**
  - Different views and measurements over time.
  - Need for contemporary guidelines and templates.
- **Software Quality Models:**
  - Various models to guide developers.
  - Examples include the ISO standards.

Top 20 Software Quality Characteristics

- **Examples:**
  - Reliability
  - Usability
  - Maintainability
  - Efficiency
  - Portability
  - Functionality
  - Etc.

Questions About Quality and Testing

- **Key Questions:**
  - What are the quality criteria to test the code against?
  - What are the quality testing practices?
  - How to prevent low-quality issues?
  - How to recognize pass or fail?
  - Actions to take if the quality test fails.
- **Considerations:**
  - You can never be 100% sure there are no defects.
  - Testing can only prove the presence of defects, not their absence.

What is Testing?

- **Definition:**

    - Verifying that the behavior of the application matches what is specified or expected.
- **Purpose:**
    - Ensure code behaves as expected.
    - Provide confidence in releasing and experimenting.

Testing Criteria and Recognizing Pass/Fail

- **Criteria:**
    - Expected output for given inputs.
    - Expected behavior of the application/system.
    - Coding standards and conventions.
- **Example of Bad Test Criteria:**
    - “The product should be user-friendly.”

Why Write Tests?

- **Documentation:**
    - Tests specify how code should work.
- **Consistency:**
    - Verify developers follow good practices and team conventions.
- **Comfort and Confidence:**
    - Strong test suite provides assurance for releases.
- **Productivity:**
    - Allows for faster shipping of code.

Types of Software Testing

- **Functional Testing**
- **Usability Testing**
- **System Testing**
- **Unit Testing**
- **Integration Testing**
- **Exploratory Testing**
- **Black Box Testing**
- **White Box Testing**
- **Regression Testing**
- **Performance Testing**
- **Stress Testing**
- **Load/Stability Testing**
- **Security Testing**
- **Regulatory and Compliance Testing**
- **Code Analysis Tools:**
    - Linters (e.g., ESLint).
    - Visual Studio Code squiggly lines.
    - SonarQube.

The Risk of No Integration Testing

- **Issues:**
    - “It works on my machine” syndrome.
    - Lack of coordination and dependencies.

- High risk when integrating code developed in isolation.

Non-Functional Requirements (Quality Requirements)

- **Examples:**
  - Performance
  - Scalability
  - Security
  - Usability
  - Reliability

Test Pyramid – Prioritizing Test Efforts

- **Concept:**
  - Focus on spending more effort on lower-level tests (unit tests) and less on higher-level tests (UI tests).
- **Mike Cohn's Agile Testing Pyramid:**
  - Base: Unit Tests
  - Middle: Service/Integration Tests
  - Top: UI/End-to-End Tests

Crispin's Agile Testing Quadrants

- **Quadrants:**
  - Different types of tests categorized based on purpose and scope.
- **Purpose:**
  - Guide teams on what tests to focus on at different stages.

Unit Tests with React

- **Tools:**
  - Jest (Test Runner)
  - React Testing Library
- **Components:**
  - Test assertion library.
  - Mocking features.
- **Testing Focus:**
  - Ensure components render correctly.
  - Simulate user interactions.

Simulating User Interaction

- **Purpose:**
  - Test how users interact with the application.
- **Methods:**
  - Simulate events like clicks, inputs, and form submissions.

Testing Resources

- **Tutorials and Videos:**
  - Unit Testing with Jest and React Testing Library.
  - Complete Guide to Component Testing with Jest.
  - Jest Crash Course.

Test Automation

- **Regression Testing:**
  - Running all tests (or prioritized ones) to ensure new changes don't break existing functionality.
- **Continuous Delivery and DevOps:**
  - Rely on test automation for rapid deployment.
- **Benefits:**
  - Ensures tests are run at the right time.
  - Frees developers and testers for more analytical testing.

Automating Integration Testing Continuously

- **Continuous Integration (CI):**
  - Integrate code changes frequently.
  - Run automated tests to catch issues early.
- **Continuous Deployment (CD):**
  - Automatically deploy code that passes tests to production or staging environments.

What is Automated Testing?

- **Definition:**
  - Using software tools to run tests automatically.
- **Purpose:**
  - Increase efficiency.
  - Reduce human error.
  - Support continuous integration and deployment.

Test Cases for Unit Test Design

- **Components of a Test Case:**
  - Test Case ID
  - Description
  - Test Steps
  - Test Data (Inputs)
  - Expected Results (Outputs)
- **Example Requirement:**
  - "If a user is a VIP customer and they order more than 10 items, they should not pay freight."
- **Acceptance Criteria Example:**
  - **Given** a logged-in VIP customer orders 11 items, **When** the order is confirmed, **Then** delivery is free.

Backend API Testing with Postman or Insomnia

- **Tools:**
  - Postman
  - Insomnia
- **Features:**
  - Send HTTP requests to APIs.
  - Write and run tests on API responses.
- **Resources:**
  - Postman's documentation and learning center.

Test-Driven Development (TDD)

- **Definition:**

  - Writing tests before writing the code to fulfill those tests.
- **Cycle:**
  - **Red:** Write a failing test.
  - **Green:** Write code to make the test pass.
  - **Refactor:** Improve the code while keeping tests passing.
- **Benefits:**
  - Improves code quality.
  - Encourages better design.
  - Provides documentation.

9 Excuses Why Developers Don't Test Their Code

33. "My code works fine—why should I even bother testing it?"
34. "This piece of code is untestable."
35. "I don't know what to test."
36. "Testing increases development time, and we're running out of time."
37. "The requirements are no good."
38. "This piece of code doesn't change."
39. "I can test this way faster if I do it manually."
40. "The client only wants to pay for deliverables."
41. "This piece of code is so small... it won't break anything."

When Test-Driven Development Goes Wrong

- **Common Issues:**
  - Not following the TDD process properly.
  - Misunderstanding the purpose of tests.
  - Writing fragile tests tightly coupled to implementation.
- **Insights:**
  - TDD is valuable even when it goes wrong.
  - It's a cornerstone of Continuous Delivery, BDD, and DevOps.

TDD = TFD + Refactoring

- **Test-First Development (TFD):**
  - Writing tests before code.
- **Refactoring:**
  - Improving code without changing its external behavior.
- **Clarification:**
  - Functional changes are not refactoring.
  - Refactoring changes the code structure, not its functionality.

Examples of TDD – Further Study

- **Resources:**
  - Videos and tutorials on TDD.
  - Step-by-step examples in various programming languages.
- **Recommendations:**
  - Lynda.com courses on TDD.
  - Exploring TDD with practical examples.

Testing Setup

- **Considerations:**
    - Use test databases or staging environments.
    - Create mock objects for dependencies.
- **Importance of Test Data:**
    - Vital for demonstrations and testing.
    - Poor test data can lead to failed demos.

Adoption of TDD in Practice

- **Reality:**
    - Few organizations strictly follow TDD.
    - The key is ensuring new code is accompanied by tests.
- **Insight from Meyer (2014):**
    - The main idea is that code must be accompanied by tests.
    - Writing code and tests simultaneously is acceptable.

TDD Empirical Study at SERL in AUT

- **Study Overview:**
    - Examined the experiences of adopting TDD in a project.
    - Compared TDD project with non-TDD projects.
- **Findings:**
    - Use of TDD was generally positive.
    - Identified benefits and challenges.

Sharing Findings with Practitioners

- **Challenges:**
    - Communicating software engineering research effectively.
- **Design Science Lens:**
    - Helps summarize and assess research contributions.
- **Visual Abstracts:**
    - Tools like VASE (Visual Abstracts for Software Engineering).
- **Resources:**
    - DSSE.org
    - GitHub repository for VASE.

Lecture 8: Review Techniques

Code Quality

Key Questions

- **Quality Criteria:**
  - What are the quality criteria to test code against?
- **Testing Practices:**
  - What code quality testing practices should be used?
- **Preventing Test Failures:**
  - What practices help ensure tests do not fail?

Best Practices

- **Clear Code Intentionality:**
  - Use good naming conventions.
  - Make code easy to understand, change, review, test, and debug.
- **Code Structure:**
  - Aim for loosely coupled and highly cohesive code units.
  - Follow Object-Oriented principles like SOLID.
  - Avoid repeating code (DRY Principle).

Improving Code Quality

- **Code Reviews:**
  - Enhance code quality through peer review.
- **Test-Driven Design:**
  - Improve code quality by writing tests first.
- **Test Automation:**
  - Automate tests to catch issues early and often.

Code Reviews

Benefits

- **Career Growth:**
  - Code reviews can help advance your career.
- **Team Dynamics:**
  - Improve collaboration and empathy among team members.
- **Quality Improvement:**
  - Identify and fix issues early.
- **Knowledge Sharing:**
  - Learn from peers and share best practices.

High-Quality Code Reviews

- **Effective Strategies:**
  - Follow structured steps for reviewing code.
  - Focus on critical issues rather than minor style problems.
- **Empathy in Code Reviews:**
  - Communicate respectfully and constructively.
  - Understand the perspective of the code author.

Resources

- Articles on improving code reviews:
    - "5 Ways Code Reviews Helped My Career"
    - "How One Code Review Rule Turned My Team into a Dream Team"
    - "How to Review Code in 7 Steps"
    - "5 Rules for Every Code Review"

Inspections

Fagan Inspections

- **Definition:**
    - A formal process for detecting defects in design and code.
- **Process:**
    - Planning, overview, preparation, inspection meeting, rework, and follow-up.
- **Comparison with Walkthroughs:**
    - Inspections are more formal and structured than walkthroughs.

Benefits

- **Error Reduction:**
    - Identify errors early in the development process.
- **Quality Assurance:**
    - Ensure adherence to standards and specifications.

Importance of Clean Code

- **Team Collaboration:**
    - Code is developed in teams; clarity is crucial.
- **Maintainability:**
    - Easier for others (and yourself) to understand and modify code later.
- **Best Practices:**
    - Think about future developers who will work on your code.
    - Be a good teammate and write code that is easy to maintain.

Pull Requests and Documentation

- **Commit Messages:**
    - Include clear and descriptive commit messages.
- **Documentation:**
    - Provide context and reasoning behind code changes.
- **Best Practices:**
    - Follow patterns for writing effective commit messages.

Integration of Code - Workflow

Steps for Developers

42. **Pull Latest Code:**
    - Update local repository with the latest code from the shared branch.
43. **Work Locally:**
    - Write tests and functional code.
    - Run tests locally to ensure code works.

44. **Commit Frequently:**
    - Make frequent commits with informative messages.
45. **Push and Merge:**
    - Push code to the working branch.
    - Merge after passing local tests.
46. **Pull Requests:**
    - Submit pull requests to merge with develop or main branches.
47. **Code Review:**
    - Collaborators review, discuss, and run integration tests.
48. **Continuous Integration:**
    - Run integration tests on merged code.

Continuous Integration (CI)

- **Definition:**
    - Practice of merging code changes frequently.
- **Benefits:**
    - Detect integration issues early.
    - Maintain a consistent codebase.

Reflective Practice and Reviews

- **Reflective Practitioner:**
    - Not just doing but thinking about how and why you're doing it.
- **Importance:**
    - Increases professional effectiveness.
    - Helps in continuous learning and improvement.
- **Resources:**
    - Donald Schön's work on reflective practice.

Automating Continuous Integration and Deployment

- **CI/CD Pipelines:**
    - Automate the process of building, testing, and deploying code.
- **Automated Checks:**
    - Build code, run linters, and execute unit tests automatically.
- **Deployment:**
    - Automatically deploy code to production or staging environments after passing tests.

CI Servers and Tools

- **Examples:**
    - GitHub Actions.
- **Configuration:**
    - Use YAML files to define workflows.
- **Resources:**
    - GitHub Actions starter workflows.
    - Articles on using GitHub Actions effectively.

Collaborative Programming Practices

Pair Programming

- **Definition:**
    - Two developers work together at one workstation.
- **Benefits:**
    - Improved code quality.
    - Enhanced team communication.
    - Knowledge sharing.

Mob Programming

- **Definition:**
    - Whole team works on the same thing, at the same time, in the same space.
- **Benefits:**
    - Collective code ownership.
    - Broader knowledge across the team.
    - Improved design decisions.
- **Challenges:**
    - Requires suitable workspace.
    - Potential for interpersonal conflicts.
    - May slow down initial coding pace.

Retrospectives

- **Purpose:**
    - Reflect on the last sprint to learn and improve.
- **Process:**
    - Use structured methods to gather insights.
    - Encourage honest and candid participation.
- **Techniques:**
    - Happiness Histogram.
    - Sailboat Exercise.
    - Mad-Sad-Glad.
- **Outcome:**
    - Identify actionable items to implement in the next sprint.
    - Focus on strengthening the team.

Tools for Online Retrospectives

- **Examples:**
    - Padlet
    - Retrium
    - RetroTool.io

Software Process Improvement (SPI)

Agile vs. Traditional SPI

- **Traditional SPI:**
    - Top-down approach.
    - Prescribes norms for operations.
- **Agile SPI:**
    - Bottom-up approach.
    - Focuses on practices evolving dynamically with the team.
- **Iterative Improvement:**

    - Agile methods emphasize continuous adaptation within ongoing projects.

SPI in Agile Teams

- **Meetings for Improvement:**
    1. **Daily Stand-ups:**
        - Coordinate work and solve immediate problems.
    2. **Sprint Retrospectives:**
        - Reflect on what went well and what can be improved.
- **Challenges:**
    - Process improvement doesn't happen automatically.
    - Requires active effort and experimentation.

Diagnosis and Planning

- **Team Assessment:**
    - Evaluate teamwork factors like leadership, team orientation, and learning.
- **Action Planning:**
    - Based on diagnosis, teams plan concrete measures to improve.
- **Examples of Actions:**
    - Reintroducing agile practices like retrospectives and daily stand-ups.
    - Implementing code reviews.
    - Pair programming.
    - Collocating the team.

Study Findings

- **Importance of Effort:**
    - Process improvement requires deliberate effort.
- **Learning from Diagnosis:**
    - Teams benefit from discussing and addressing identified issues.
- **Continuous Improvement:**
    - Agile teams need to actively engage in process improvement activities.

Lecture 9: Code Craft and Code Quality

Overview

- **Quality of Code:**
  - Easy to change.
  - Behaves as expected (no bugs).
- **Quality of Product:**
  - Solves user problems.
  - Easy to understand and modify.
  - Predictable changes with limited impact.
  - Clear code intention.

Techniques to Improve Code Quality

- **Test-Driven Development (TDD)**
- **Continuous Integration/Continuous Deployment (CI/CD)**
- **Pair/Mob Programming**
- **Code Reviews**
- **Coding Standards and Static Code Analysis**
  - Linters, SonarQube, etc.
- **Proper Naming and Structure**
- **Small Code Structures**

Coding as a Craft

- **Iterative Development:**
  - Start with a sketch.
  - Add detail iteratively.
  - Revise, extend, and refine.
- **Continuous Improvement:**
  - Software is never truly finished.
- **Team Collaboration:**
  - Others need to read, understand, and modify your code.
- **Maintainability:**
  - Easier to debug and extend.
- **Future Self:**
  - You may revisit your code months later.
- **Professionalism:**
  - "Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live." — Martin Golding
- **Readability:**
  - Code is read more often than it is written.

Writing Small and Focused Code

Functions

- **Keep Functions Small:**
  - Rarely more than 20 lines; ideally less than 10.
- **Limit Arguments:**
  - Preferably no arguments.
- **Single Responsibility:**
  - Do one thing and do it well.
- **Example:**
  - A function that fetches, manipulates, and stores data should be split into three smaller functions.

Classes

- **Single Responsibility Principle (SRP):**
  - Classes should have one responsibility.
- **Warning:**
  - Avoid too many tiny classes that complicate understanding and changes.

Making Code Self-Documenting

- **Clear Intention:**
  - Code should be readable and its purpose immediately clear.
- **Avoid Excessive Comments:**
  - “Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments.” — Robert C. Martin
- **Eliminate Magic Numbers:**
  - Use named constants instead of hard-coded values.
- **Use Descriptive Names:**
  - Methods and variables should reflect their purpose.

Example Refactoring

Before:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

After:

```
if (employee.isEligibleForFullBenefits())
```

Lecture 10: Software Risks and Scale

Software and Risk - Definitions

Traditional Focus on Project Risks

- **Primary Focus in Literature:**
  - Risk analysis and mitigation often center on project development aspects like budget and schedule overruns.
  - Less emphasis on satisfying the customer by meeting technical requirements.

Definitions of Project Risk

- **Henry (2004):**
  - *"An event, development, or state in a software project that causes damage, loss, or delay."*
- **Schwalbe (2004):**
  - *"Problems that might occur on the project and how they might impede project success."*

Limitations of Traditional Risk Analysis

- **Narrow Focus:**
  - Emphasis on quantifiable risks related to project development (time, budget).
  - Often excludes qualitative risks and broader stakeholder considerations.
- **Result:**
  - Risks to extra-project stakeholders may be underestimated or ignored.
  - Tunnel vision can lead to overlooking key stakeholders, affecting requirements and implementation.

Software and Risk – Traffic Control Scenario

Overview of the Project

- **Objective:**
  - Develop traffic control software to direct approaching traffic into the least congested lanes on a multi-lane bridge.
  - Aim to facilitate maximum and continuous traffic flow, especially during rush hours.

Identified Stakeholders

- **Vehicle Drivers:**
  - Those traversing the bridge.
- **Bridge Maintenance Personnel:**
  - Responsible for the upkeep and safety of the bridge.
- **City Traffic Authority:**
  - Manages traffic flow and infrastructure.

Success Criteria Defined

- **System Performance:**
  - Works well in its context.
  - Does not promote vehicle accidents.

- **Project Management:**
    - Delivered on time.
    - Within budget.
    - Accurate cost/benefit analysis showing reasonable return on investment.

Unexpected Outcome

- **Despite meeting all the above conditions, the system was judged a failure.**
- **Question:**
    - Why was the system considered a failure despite meeting its success criteria?

Software and Risk - Traffic Control Gone Wrong

System Failure During Emergency

- **Scenario:**
    - An emergency nuclear disaster evacuation exercise required the system to manage heavy traffic loads continuously for 8 hours.
- **System Behavior:**
    - In the eighth hour, the software changed lane directions for lanes already filled with cars.
- **Consequences:**
    - Misdirection and accidents occurred.
    - The bridge was clogged for almost 20 hours.
    - Significant delays and safety hazards were introduced.

Technical Cause of Failure

- **Crystal Clock Issue:**
    - The clock used for timing decisions would gradually go out of synchronization after 7 or more hours of continuous use.
    - This led to incorrect timing decisions by the software.
- **Developer Awareness:**
    - The developer was fully aware of this problem.
    - It was a known issue that required attention.

Software and Risk – Cold Reboot Needed?

Developer’s Mitigation Strategy

- **User Manual Warning:**
    - To reset the clock and avoid synchronization issues, the developer specified in the user manual that the software should be briefly stopped and restarted after 6 hours of continuous heavy traffic loads.
- **Assumption:**
    - The users (traffic authority personnel) would read the manual and follow the instructions to reboot the system as needed.

Decision Rationale

- **Meeting Constraints:**

- To meet schedule and budget constraints, the developers opted for a documentation fix rather than implementing a software solution to address the clock issue.

- **Focus on Project Goals:**
  - Emphasized delivering the system on time, within budget, and satisfying the immediate customer requirements.
  - Prioritized project development risks over operational risks.

Software and Risk – Documentation Fix?

Narrowing of Risk Focus

- **Project Risk Analysis:**
  - Focused primarily on risks impacting development goals (time, budget, customer satisfaction).
- **Exclusion of Broader Risks:**
  - Did not adequately consider operational risks and the potential impact on end-users and other stakeholders during extended use.
- **Emphasis in Literature:**
  - Many software development textbooks and risk management articles reinforce this narrow focus on development risks.

Implications

- **Underestimation of Risks:**
  - Ignoring qualitative risks and the needs of extra-project stakeholders can lead to significant failures, as seen in the traffic control scenario.
- **Ethical Considerations:**
  - Responsibility to consider the safety and well-being of all stakeholders, not just project delivery metrics.

Software and Risk – All Stakeholders?

Need for Broader Risk Analysis

- **Stakeholder Identification:**
  - Essential to identify all stakeholders, including those indirectly affected by the software.
- **Quotation from Schmidt et al. (2001):**
  - *"Failure to identify all stakeholders: Tunnel vision leads project management to ignore some of the key stakeholders in the project, affecting requirements, and implementation, etc."*
- **Recommendation:**
  - Project risk analysis must expand beyond traditional methods to include a broader scope of risks and stakeholders.
  - Consider both quantitative and qualitative risks.
- **Ethical Responsibility:**
  - Developers have a duty to foresee potential negative impacts on all stakeholders and address them appropriately.

Software and Risk – SoDIS (Software Development Impact Statement)

Introduction to SoDIS

- **Purpose:**
  - A practical mechanism for ethical risk assessment in software development.
  - Helps identify and mitigate risks that traditional risk analysis might overlook.
- **Components:**
  - Systematically identifies stakeholders and potential impacts.
  - Evaluates risks associated with software decisions.
  - Encourages ethical considerations in risk management.

Stakeholder Analysis

- **Identifying All Stakeholders:**
  - Direct Stakeholders:
    - Users, customers, developers.
  - Indirect Stakeholders:
    - Those affected by the software's operation, such as the general public, regulatory bodies, etc.
- **Understanding Stakeholder Concerns:**
  - Elicit concerns related to safety, privacy, security, usability, and other factors.

SoDIS Inspection Process

49. **Define the Project:**
    - Clearly understand the project's goals, scope, and context.
50. **Identify Stakeholders:**
    - Create a comprehensive list of all stakeholders.
51. **Elicit Stakeholder Concerns:**
    - Engage with stakeholders to understand their needs and potential impacts.
52. **Identify Ethical Issues:**
    - Analyze the project for potential ethical dilemmas or conflicts.
53. **Assess Risks:**
    - Evaluate the likelihood and impact of identified risks on each stakeholder.
54. **Develop Mitigation Strategies:**
    - Propose solutions to minimize or eliminate risks.
    - Consider alternative designs or additional features.
55. **Document Findings:**
    - Create a Software Development Impact Statement outlining the analysis and decisions.
56. **Review and Iterate:**
    - Revisit the SoDIS throughout the project as new risks or stakeholders emerge.

Benefits of SoDIS

- **Comprehensive Risk Management:**
  - Ensures all potential risks are considered.
- **Ethical Decision-Making:**
  - Incorporates ethical considerations into project planning.
- **Stakeholder Engagement:**

    - Promotes open communication with stakeholders.
- **Improved Project Outcomes:**
    - Reduces the likelihood of project failures due to unforeseen risks.

Software and Risk – Security Risk Lifecycle

Lifecycle Phases

57. **Risk Identification:**
    - Recognize potential security risks early in the development process.
58. **Risk Assessment:**
    - Evaluate risks in terms of likelihood and potential impact.
    - Prioritize risks based on severity.
59. **Risk Mitigation:**
    - Implement strategies to reduce or eliminate risks.
    - Includes designing secure code, implementing security controls, etc.
60. **Risk Monitoring:**
    - Continuously observe risks throughout the project lifecycle.
    - Update risk assessments as new threats emerge.
61. **Risk Communication:**
    - Keep all stakeholders informed about risks and mitigation efforts.
62. **Risk Review:**
    - Regularly review the effectiveness of risk management strategies.
    - Learn from incidents to improve future practices.

Importance in Software Development

- **Proactive Approach:**
    - Anticipate and address risks before they become issues.
- **Regulatory Compliance:**
    - Meet legal and industry standards for security and risk management.
- **Trust and Reputation:**
    - Protect the organization's reputation by preventing security breaches.

Concluding: Software Projects – Student Projects & Risks

Recent Research on Student Projects and Risk Management

- **Studies by New Zealand Researchers (Kirk et al., 2022; 2024):**
    - Focused on incorporating risk management into student group projects.
- **Purpose:**
    - To understand how risk management affects project outcomes in educational settings.
    - To develop low-overhead risk management frameworks suitable for student projects.

Findings

- **Impact of Poor Risk Management:**

    - Teams with inadequate risk management strategies tend to perform worse.
    - Higher likelihood of encountering significant issues within the team.
- **Instructor Intervention:**
    - Teams with poor risk practices more frequently required instructor assistance to address problems.
- **Benefits of Risk Frameworks:**
    - Improved team performance.
    - Better project outcomes.
    - Enhanced learning experiences for students.

Identified Risk Categories

63. **Communication Risks:**
    - Miscommunication or lack of communication among team members.
    - Language barriers or differing communication styles.
64. **Technical Risks:**
    - Insufficient technical skills or experience.
    - Challenges with new technologies or tools.
65. **Scheduling Risks:**
    - Poor time management.
    - Conflicting commitments (e.g., other courses, work).
66. **Scope Risks:**
    - Unclear project requirements.
    - Scope creep or changes in project objectives.
67. **Stakeholder Risks:**
    - Not adequately considering client needs or expectations.
    - Lack of feedback from supervisors or clients.
68. **Team Dynamics Risks:**
    - Conflicts within the team.
    - Uneven workload distribution.
69. **Resource Risks:**
    - Limited access to necessary resources (software, hardware).

Conclusions and Recommendations

- **Importance of Risk Management Education:**
    - Teaching students about risk identification, assessment, and mitigation prepares them for real-world projects.
- **Implementing Risk Frameworks:**
    - Even low-overhead risk management processes can significantly improve project outcomes.
- **Active Monitoring:**
    - Regular check-ins and updates on risk status help prevent issues from escalating.
- **Instructor Support:**
    - Educators should guide students in risk management practices and intervene when necessary.

Lecture 10a: Software Ecosystems

Overview

This lecture explores the concept of software ecosystems, their definitions, maturity models, focus areas, and the new competencies demanded by such ecosystems. It delves into how software ecosystems function, their configurations, and the implications for software development and innovation.

Software Ecosystems - Definitions

General Definition

- **Software Ecosystem:** An entity where an ecosystem owner provides not just a software product but an underpinning platform. This platform offers a set of APIs through which external developers can connect and build applications.

Examples of Software Ecosystems

- **Apple App Store**
- **Google Play Store**
- **Xero's App Marketplace** in New Zealand

Goals of Software Ecosystems

- **Innovative Continuous Software Business:**
  - Aim for growth and sustainability.
  - Build a healthy network of partners around the business.
- **Staying Power (Cusumano, 2010):**
  - Minimize risk.
  - Increase innovation.
  - Increase revenue.
  - Create a robust network of partners.

Formal Definition

- **Jansen, Finkelstein, and Brinkkemper (2009):**
  - *"A set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently underpinned by a common technological platform or market and operate through the exchange of information, resources, and artifacts."*

Software Ecosystems – Maturity Models

Focus Area Maturity Model

- **Purpose:** To assess and guide the development of software ecosystems.
- **Components:**
  - **Focus Areas:** Domains or practices essential for ecosystem maturity.
  - **Capabilities:** Specific abilities within each focus area.
  - **Practices:** Actions or processes within capabilities.
  - **Maturity Levels:** Stages of development or proficiency.
  - **Functional Domain Capabilities:** The overall competencies achieved.

Implementation

- **Levels of Achievement (Maturity):**
  - Practices are implemented progressively to achieve higher maturity levels.
- **Institutionalization:**
  - Practices become ingrained in the organizational context.

Software Ecosystems – Focus Areas

The maturity model includes several focus areas critical for the development and sustainability of software ecosystems.

1. Associate Models

- **Definition:**
  - Practices related to the management and coordination of partners within the ecosystem.
- **Key Practices:**
  - **Creation of Partnership Models:** Developing frameworks for collaboration.
  - **Partner Training:** Educating partners on the platform and tools.
  - **Consultancy and Sales Partner Support:** Assisting partners in marketing and sales efforts.
- **Technical Aspects:**
  - **Communication Systems:** Building systems that enable partners to interact with end-users.
    - Example: Approval systems in app stores.
    - **Customer Partner Connection Centers:** Platforms like SAP's that allow partners to share ticketing systems with customers and SAP.

2. Ecosystem Health

- **Definition:**
  - Viewing the ecosystem as a living entity that can be analyzed and compared with other ecosystems.
- **Key Practices:**
  - **Partner Health Analysis:** Assessing the well-being and performance of partners.
  - **Sharing Market Data:** Providing insights and data to partners.
  - **Strategic Choices Regarding Competing Ecosystems:** Making informed decisions to maintain competitiveness.

3. Open Markets

- **Definition:**
  - Practices focused on creating an open market for services and applications within the ecosystem.
- **Key Practices:**
  - **Extension Approval:** Processes for approving partner-developed extensions or apps.
  - **Extension Marketing:** Promoting partner extensions to users.
  - **Business Model Innovation:** Developing new ways to generate revenue and value.
  - **App Delivery:** Ensuring efficient and reliable distribution of apps.
- **Characteristics:**
  - **Balance Between Management and Technical Boundaries:** Practices cover both managerial and technical aspects.

4. Open Platforms

- **Definition:**
  - Practices related to building a stable, solid, and open platform that supports the ecosystem.
- **Key Practices:**
  - **Platform Creation:** Developing the core platform infrastructure.
  - **Platform Security:** Ensuring the platform is secure from threats.
  - **Extension Capabilities:** Allowing for extensibility and customization.
  - **Documentation:** Providing clear and comprehensive documentation for developers.

5. Intellectual Property

- **Definition:**
  - Managing patents and intellectual property within the ecosystem.
- **Key Practices:**
  - **Innovation Sharing:** Encouraging the sharing of innovations across the ecosystem.
  - **Patent Management:** Handling patents effectively.
  - **Licenses:** Establishing licensing agreements that benefit all parties.
  - **Co-Creation Stimulation:** Promoting collaborative innovation to enhance ecosystem health.

6. Open Innovation

- **Definition:**
  - Sharing knowledge across the ecosystem to enable external developers to improve and innovate.
- **Key Practices:**
  - **Development Practice Sharing:** Distributing best practices among partners.
  - **Innovation with Partners:** Collaborating on new developments.
  - **Ecosystem Standards:** Establishing standards that facilitate compatibility and interoperability.

7. Software Development Governance

- **Definition:**
  - Observing, supporting, and enabling software developers within the ecosystem.
- **Key Practices:**
  - **Testing:** Ensuring quality through rigorous testing procedures.
  - **Roadmapping:** Planning future developments and updates.
  - **Shared Requirements:** Collaborating on requirements across the ecosystem.
  - **Developer Enablement:** Providing tools and resources to facilitate third-party development.
  - **Data Collection:** Gathering data on software operation to inform improvements.

Software Ecosystem Configuration

Conceptual Framework

- **Ecosystem Configuration:**
  - The structure and arrangement of components within a software ecosystem.
  - Includes the relationships and interactions among businesses, platforms, developers, and users.

Key Components

70. **Platform Owner (Ecosystem Owner):**
    - Provides the core platform and APIs.

  - Sets the rules and standards for the ecosystem.

71. **Complementors (Partners/Developers):**
    - Build applications or services on the platform.
    - Contribute to the ecosystem's diversity and innovation.
72. **End-Users:**
    - Consumers of the applications and services.
    - Provide feedback and demand that drive the ecosystem.
73. **Regulatory Bodies:**
    - May influence the ecosystem through laws and regulations.

Dynamics

- **Interactions:**
  - Continuous exchange of information, resources, and artifacts.
- **Value Creation:**
  - Collaboration leads to innovative solutions and mutual benefits.
- **Ecosystem Health:**
  - Dependent on the balance and health of relationships among components.

Software Ecosystems & Open Innovation

Integration of Open Innovation

- **Definition:**
  - Open innovation involves leveraging external ideas and paths to market in conjunction with internal ones.
- **In Software Ecosystems:**
  - Encourages external developers to contribute to the platform.
  - Promotes collaborative development and shared success.

Challenges

- **Requirements Management:**
  - Managing diverse requirements from various stakeholders can be complex.
- **Communication:**
  - Ensuring clear communication between the platform owner and partners.
- **Alignment of Goals:**
  - Balancing the platform owner's objectives with those of the partners.

Strategies for Managing Requirements Selection

- **Transparency:**
  - Openly sharing roadmaps and development plans.
- **Community Engagement:**
  - Involving partners and users in decision-making processes.
- **Prioritization Frameworks:**
  - Establishing criteria to prioritize requirements based on impact and feasibility.

Software Ecosystems & Requirements Flow

Requirements Flow in Ecosystems

- **Upstream and Downstream Flow:**
    - Requirements can flow from end-users to complementors and then to the platform owner, and vice versa.
- **Complexity:**
    - Multiple stakeholders with varying needs increase the complexity of managing requirements.

Key Considerations

74. **Stakeholder Alignment:**
    - Ensuring that all parties have a shared understanding of goals and expectations.
75. **Traceability:**
    - Keeping track of requirements as they evolve and move through the ecosystem.
76. **Conflict Resolution:**
    - Addressing conflicting requirements or interests among stakeholders.

Strategies for Effective Requirements Flow

- **Centralized Communication Channels:**
    - Platforms for sharing information and updates among stakeholders.
- **Modular Architecture:**
    - Designing the platform to accommodate diverse requirements without extensive rework.
- **Feedback Mechanisms:**
    - Regularly collecting and integrating feedback from users and partners.

Software Ecosystems – New Competencies Demanded

Evolving Skill Sets

- **Technical Skills:**
    - Proficiency in platform development and API design.
    - Understanding of security, scalability, and extensibility.
- **Business Skills:**
    - Ability to develop and manage partnership models.
    - Knowledge of intellectual property management and licensing.
- **Communication Skills:**
    - Effective collaboration with a diverse range of partners and stakeholders.
- **Strategic Thinking:**
    - Navigating competitive ecosystems.
    - Making informed decisions to foster ecosystem health.

Importance of Adaptability

- **Continuous Learning:**

    - Staying updated with emerging technologies and market trends.
- **Innovation Mindset:**
    - Embracing open innovation practices.
    - Encouraging creativity and new ideas within the ecosystem.

Emphasis on Governance

- **Software Development Governance:**
    - Establishing policies and practices that support developer engagement and quality assurance.
- **Ecosystem Health Monitoring:**
    - Regularly assessing the ecosystem's performance and making necessary adjustments.

Lecture 11: Agile is Eating the World

Overview

This lecture explores the concept of Agile and its profound impact on the software development industry and beyond. We delve into the meaning of Agile, its core principles, the reasons behind its widespread adoption, and how it influences individuals, teams, and organizations. We also address common misconceptions and emphasize the importance of focusing on delivering value to customers.

Introduction: Agile is Eating the World

Why and What Does It Even Mean?

- **Agile's Ubiquity:**
    - Agile methodologies have become pervasive in software development and are extending into other industries.
    - Publications like **Forbes Magazine** and **The Wall Street Journal** highlight Agile's growing influence.
- **Confusion Around Agile:**
    - The term "Agile" has become a buzzword, leading to misunderstandings about its true meaning.
    - With over **70 different Agile practices**, it's easy to lose sight of the core principles.

Diverse Agile Practices

- **Examples of Agile Methods:**
    - **Scrum**
    - **Kanban**
    - **Extreme Programming (XP)**
    - **Mob Programming**
    - **DevSecOps**
    - **Fast Agile**
    - **Shape Up**
- **Challenge:**
    - The abundance of practices can overwhelm practitioners, causing them to focus on methods rather than principles.

The Agile Manifesto

Core Values and Principles

- **Agile Manifesto (2001):**
    - A declaration of four fundamental values and twelve principles to guide Agile development.
- **Key Values:**
    1. **Individuals and Interactions** over Processes and Tools
    2. **Working Software** over Comprehensive Documentation
    3. **Customer Collaboration** over Contract Negotiation
    4. **Responding to Change** over Following a Plan
- **Agile as a Mindset:**

- Emphasizes beliefs and principles guiding behavior, interactions, and process design.
- Focuses on software development but has broader applications.

Simplifying Agile: The Heart of Agile

- **Alistair Cockburn's Perspective:**
    - **Heart of Agile** reduces Agile to four words:
        - **Collaborate**
        - **Deliver**
        - **Reflect**
        - **Improve**
- **Purpose:**
    - Simplify Agile to its essence, making it accessible and applicable.
- **Resources:**
    - The Heart of Agile Technical Report
    - Heart of Agile Website

Modern Agile Principles

Four Guiding Principles

77. **Make People Awesome**
    - **Focus on Users and Customers:**
        - Understand their context, pain points, and aspirations.
        - Aim to enhance their capabilities and experiences.
    - **Team Empowerment:**
        - Support team members to grow and excel.
78. **Make Safety a Prerequisite**
    - **Psychological Safety:**
        - Create an environment where team members feel safe to express ideas and take risks.
    - **Operational Safety:**
        - Protect people's time, information, reputation, money, health, and relationships.
        - Ensure systems are resilient and secure.
79. **Experiment and Learn Rapidly**
    - **Continuous Learning:**
        - Embrace experimentation as a path to learning.
        - Conduct "safe-to-fail" experiments to reduce fear of failure.
    - **Adaptability:**
        - Use feedback from experiments to adjust and improve.
80. **Deliver Value Continuously**
    - **Incremental Delivery:**
        - Break down large tasks into smaller, deliverable pieces.
        - Deliver value to customers as early and often as possible.
    - **Customer Feedback:**
        - Use customer input to guide future development.

The Goal of Agile Mindset and Practices

Beyond Processes and Tools

- **Not Just About Making Money:**
  - Financial success is a byproduct, not the primary goal.
- **Not About Being Agile for Its Own Sake:**
  - Adopting Agile methodologies isn’t the end goal.
- **Not Only About Working Software:**
  - While important, software is a means to an end.

The True Goal: Delivering Customer Value

- **Delighting Customers:**
  - Create products or services that exceed customer expectations.
  - Focus on what makes the customer’s life easier, faster, or more competitive.
- **Continuous Value Stream:**
  - Provide an ongoing stream of value to customers.
  - Deliver value sooner rather than later.
- **Quote:**
  - *“Providing a continuous stream of additional value to customers and delivering it sooner.”* — Denning

Reflective Questions

- **What is the Value You Create?**
  - Understand and articulate the value your work brings to customers.
  - Ensure alignment between team efforts and customer needs.
- **Do You Care About Customer Value?**
  - Cultivate a genuine concern for the customer’s success and satisfaction.

From Mindset to Process

Transitioning to Agile Processes

- **Agile Values Lead to Agile Practices:**
  - Values and principles inform the design of processes like Scrum.
- **Empirical Process Control:**
  - Scrum is an empirical framework that relies on transparency, inspection, and adaptation.
- **Focus on Goals:**
  - Emphasize understanding and progressing towards the Product Goal and Sprint Goals.
  - Goals should be centered around delivering customer value.

Scrum Values and Principles

The Five Scrum Values

81. **Commitment**
    - Team members personally commit to achieving team goals.
82. **Courage**

- The team has the courage to do the right thing and work on tough problems.

83. **Focus**
    - Everyone focuses on the work of the Sprint and the goals of the Scrum Team.
84. **Openness**
    - The team and stakeholders agree to be open about all the work and challenges.
85. **Respect**
    - Team members respect each other to be capable, independent people.

Empirical Process Control in Scrum

- **Transparency:**
  - Significant aspects of the process must be visible to those responsible for the outcome.
- **Inspection:**
  - Regularly inspect Scrum artifacts and progress toward the Sprint Goal.
- **Adaptation:**
  - Adjust processes or materials when deviations are detected.

Responsibility and Risks

- **Shared Responsibility:**
  - The team collectively owns the process and outcomes.
- **Risk Management:**
  - Continuously identify and mitigate risks through iterative development.

The Need to Be Agile

Adapting to Complexity and Change

- **Complex Work Requires Agility:**
  - Best practices may not exist for complex problems.
  - Agile methods help navigate uncertainty and change.
- **Rapid Technological Advancements:**
  - Technology and potential solutions evolve faster than traditional processes can handle.

Individual, Team, and Organizational Agility

- **Personal Agility:**
  - Be open to learning and adapting.
  - Embrace new ideas and approaches.
- **Team Agility:**
  - Collaborate effectively.
  - Share knowledge and skills.
- **Organizational Agility:**
  - Create structures that support Agile values.
  - Foster a culture of continuous improvement.

Breaking Free from Method Prison

Guided Continuous Improvement

- **Beyond Rigid Methodologies:**
    - Avoid being confined by strict adherence to specific methods.
- **Optimize for Flow and Benefits:**
    - Focus on delivering continuous value rather than meeting fixed constraints.
- **Embrace Uncertainty:**
    - Accept that late requirements and changes can be beneficial.

The #NoProjects Movement

- **Shift from Projects to Products:**
    - View work as an ongoing stream of value rather than finite projects.
- **Continuous Delivery:**
    - Deliver value incrementally with no predetermined end date.
- **Double Loop Learning:**
    - Reflect on both actions and underlying assumptions to drive improvement.

The Laws of Agile

1. The Law of the Customer

- **Customer Obsession:**
    - Deliver value to customers continuously.
    - Everyone has a clear line of sight to the ultimate customer.
- **Eliminate Non-Value Activities:**
    - Remove processes and tasks that don't add value to the customer.
- **Adjust Goals and Systems:**
    - Align everything—goals, values, principles, processes—with delivering customer value.

2. The Law of the Team

- **Small, Cross-Functional Teams:**
    - Work is done by small teams with diverse skills.
- **Autonomy and Empowerment:**
    - Teams have the authority to make decisions about their work.
- **Short Cycles and Feedback:**
    - Work in short iterations with continuous feedback from customers or end-users.

3. The Law of the Network

- **Collaboration Beyond Teams:**
    - Teams collaborate across the organization, breaking down silos.
- **Network of Teams:**

  - The organization functions as a network rather than a hierarchy.
- **Agility at Scale:**
  - Apply Agile principles at the organizational level to respond swiftly to changes

Addressing Bureaucracy and Silos

Challenges in Traditional Organizations

- **Bureaucratic Obstacles:**
  - Rigid structures impede collaboration and responsiveness.
- **Silo Mentality:**
  - Departments operate in isolation, hindering information flow.

Overcoming Barriers

- **Create Cross-Functional Teams:**
  - Assemble teams with members from different departments.
- **Foster Open Communication:**
  - Encourage transparency and information sharing.
- **Leadership Support:**
  - Leaders must support and model Agile values.

Lecture 12: Course Review and Empirical Software Engineering

Course Review

Key Topics Covered

- **Software Development Tools:**
    - Setting up individual development environments.
    - Collaborative tools for team projects.
- **Tech Stacks:**
    - Understanding and working with chosen front-end and back-end technologies.
    - Roadmaps for current tech stacks.
- **DevOps and Scrum Practices:**
    - Implementing DevOps methodologies.
    - Scrum framework for iterative development.
- **Code Craft and Quality:**
    - Writing clean, maintainable code.
    - Importance of code reviews and testing.
- **Continuous Integration and Deployment:**
    - Setting up CI/CD pipelines.
    - Automating testing and deployment processes.
- **Empirical Software Engineering:**
    - Understanding what works under different circumstances and why.
    - Research and analysis of software engineering practices.

Review Strategies

- **Course Schedule and Modules:**
    - Revisit the course schedule to recall topics and activities.
- **Assessment Overview:**
    - Understand the weight and focus of each assignment.
- **Example Test:**
    - Review the example test on Canvas to familiarize with the format.
- **Mind Maps and Roadmaps:**
    - Use Jim's mind map and technology roadmaps to visualize connections between topics.
- **Development Process and Practices:**
    - Reflect on key events in the development process.
    - Consider the practices adopted during the team project.
- **Collaborative Experience:**
    - Digest insights from collaborative software engineering experiences.

Roadmaps and Resources

roadmap.sh

- **Community Effort:**
    - A platform providing roadmaps, guides, and educational content for developers.
- **Skill-Based Roadmaps:**

- React Roadmap
- JavaScript Roadmap
- TypeScript Roadmap

- **Role-Based Roadmaps:**
  - Frontend Roadmap
  - Backend Roadmap
- **Architectural Patterns:**
  - Understanding 12-Factor Apps.
  - 12-Factor App Video

Iterative Development Process

Agile Iterations

- **Initial Planning:**
  - Create an initial product backlog and story map.
  - Identify user stories with acceptance criteria.
- **Iteration Planning:**
  - Detail understanding and design of features for the next iteration only.
  - Set goals and create an iteration backlog.
- **Development Phases:**
  - Iteration of design, code, and test processes.
  - Regular team meetings during iterations.
  - Continuous integration and continuous delivery (CI/CD).
- **Coordination and Feedback:**
  - Coordinate work within the team.
  - Obtain feedback on the product from users and clients.
- **Quality Assurance:**
  - Implement practices like Test-Driven Development (TDD).
  - Utilize pair and mob programming.

Sprints and Product Increments

- **Sprint 1:**
  - Weeks 5 and 6.
  - Focus on setting up the development environment and initial feature development.
- **Sprint 2:**
  - Weeks 7 and 8.
  - Build upon the initial product increment, add new features.
- **Sprint 3:**
  - Weeks 9 and 10.
  - Finalize the product, polish features, and prepare for deployment.

Empirical Software Engineering Research

Understanding What Works and Why

- **Research Questions:**
  - How can technical debt be measured?
  - What are the benefits and challenges of planning poker for estimation?
  - How are requirements changes managed in globally distributed teams?
  - How is DevOps implemented, and what are the benefits and challenges?

  - When is it better to use mob, pair, or solo programming?
- **Methods:**
  - Interviews and surveys.
  - Case studies.
  - Content analysis of transcripts.
- **Areas of Interest:**
  - Team health monitoring and diagnosis.
  - Coordination of distributed teams.
  - Teaching and learning programming and software engineering effectively.
  - Emerging roles in agile teams and work division.

Case Study: Implementing DevOps

- **Research Objective:**
  - Investigate the enablers and challenges in implementing DevOps in practice.
- **Approach:**
  - Conduct interviews with multiple organizations at different stages of DevOps adoption.
  - Gather views from different roles, including developers, testers, operations, QA release managers, and tech leads.
- **Insights:**
  - Deep understanding of DevOps implementation in real contexts.
  - Identification of benefits, challenges, and success factors.

The DevOps Way of Working

Introduction to DevOps

- **Definition:**
  - A set of practices that combines software development (Dev) and IT operations (Ops).
  - Aims to shorten the system development life cycle and provide continuous delivery.
- **Key Concepts:**
  - Continuous integration and continuous deployment (CI/CD).
  - Infrastructure as code.
  - Monitoring and logging.
- **Cultural Shift:**
  - Emphasis on collaboration between development and operations teams.
  - Shared responsibility for the product from development to deployment.

Case Study: DevOps Implementation

- **Company Profile:**
  - Medium-sized SaaS product company experiencing fast growth.
  - Agile way of working (Scrum) with cross-functional teams.
- **Team Structures:**
  - Teams organized by product functional modules.
  - Roles include developers, testers, operations, QA release managers, and infrastructure leads.
- **Adoption of DevOps:**
  - Transition from monolithic applications to microservices.
  - Shift towards infrastructure as code and automation.
  - Embedding operations within development teams.

Shared Meaning of DevOps

- **Team Perspectives:**

  - Developers appreciate having control over infrastructure and deployment.
  - Operations personnel value collaboration with development teams.
  - QA and release managers see benefits in faster and more reliable releases.
- **Common Themes:**
  - Increased collaboration and communication.
  - Shared responsibility for deployment and monitoring.
  - Emphasis on automation and continuous delivery.

Benefits of DevOps Adoption

- **Organizational Benefits:**
  - Faster time-to-market with more frequent deployments.
  - Enhanced feedback loops leading to continuous improvement.
  - Increased stability and reduced risk of errors.
- **Team Benefits:**
  - Better code quality due to awareness of deployment impacts.
  - Greater team ownership and responsibility.
  - Improved knowledge sharing and problem-solving capabilities.

Enablers and Success Factors

- **Cultural Enablers:**
  - Trust among team members and management.
  - Open communication and willingness to share knowledge.
- **Technical Enablers:**
  - Automation tools for testing, building, and deployment.
  - Adoption of microservices architecture.
  - Use of cloud technologies and infrastructure as code.
- **Organizational Support:**
  - Cross-training and upskilling team members.
  - Support from leadership for the DevOps transformation.

Challenges and Barriers

- **Cultural Challenges:**
  - Resistance to change from traditional roles.
  - Shifting mindsets to embrace shared responsibility.
- **Technical Challenges:**
  - Complexity in changing technology stacks.
  - Keeping up with new tools and practices.
- **Resource Challenges:**
  - Staffing issues, including hiring and retaining skilled personnel.
  - Upskilling existing team members to handle new responsibilities.

DevOps Principles and Practices

CALMS Framework

- **Culture:**
  - Fostering a collaborative and open environment.
- **Automation:**
  - Automating repetitive tasks to increase efficiency.
- **Lean:**

  - Streamlining processes to reduce waste.
- **Measurement:**
  - Tracking performance metrics to inform improvements.
- **Sharing:**
  - Encouraging knowledge sharing and transparency.

Technology and Tools

- **Development and Build Tooling:**
  - Version control systems (e.g., Git).
  - Build automation tools (e.g., Jenkins).
- **Automated Testing Tools:**
  - Unit testing frameworks (e.g., JUnit).
  - Integration testing tools (e.g., Cucumber).
- **Deployment Tooling:**
  - Containerization (e.g., Docker).
  - Orchestration (e.g., Kubernetes).
- **Monitoring and Logging:**
  - Tools for tracking application performance and errors.
- **Collaboration Tools:**
  - Communication platforms (e.g., Slack).
  - Project management tools (e.g., Jira).

Research and Case Studies

Empirical Studies on DevOps

- **"DevOps Capabilities, Practices, and Challenges: Insights from a Case Study"**
  - By M. Senapathi, J. Buchan, and H. Osman.
  - Explores the practical implementation of DevOps in organizations.
- **"Emerging Trends for Global DevOps: A New Zealand Perspective"**
  - By W. Hussain, T. Clear, and S. MacDonell.
  - Discusses the adoption and impact of DevOps in the New Zealand context.

Key Findings

- **Shared Understanding:**
  - Importance of having a common definition and goals for DevOps within the organization.
- **Drivers for Adoption:**
  - Need for faster delivery and response to customer needs.
  - Desire to reduce bottlenecks and improve collaboration.
- **Benefits Realized:**
  - Enhanced agility and customer experience.
  - Improved team dynamics and job satisfaction.
- **Challenges Faced:**
  - Cultural resistance and change management.
  - Technical hurdles in transitioning to new architectures and tools.