Module 1

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 1 Study Guide and Deliverables

Module Topics:	• Topic 1: Introduction to Software Engineering • Topic 2: Software Processes • Topic 3: Agile Methodology • Topic 4: Software Quality Assurance, Configuration Management, and Risk Management
Readings:	• Online lecture notes • Braude, Parts I, II, and III (or related chapters in other textbooks)
Discussions:	• Introduce Yourself on the Class Discussion Board due **Wednesday, September 8 at 11:59 PM ET** • Weekly Group Meeting
Assignments:	• Pre-Class Survey due **Wednesday, September 8 at 11:59 PM ET** • Project Sign-up Sheet due **Wednesday, September 8 at 11:59 PM ET** • Project Iteration 0 (Proposal) due **Tuesday, September 14 at 6:00 AM ET** • Lab 1 due **Tuesday, September 14 at 6:00 AM ET**
Live Classrooms:	• **Tuesday, September 7 from 7:00-9:00 PM ET** • **Saturday, September 11 from 8:00-9:00 PM ET**

Learning Outcomes

By the end of this module, you are expected to do the following:

- Describe and compare major software process models and activities in the software process.
- Explain the Agile methodology and techniques.
- Form the project team and complete the project proposal.
- Identify risks in your project and define the retirement plan.
- Define the quality metrics and quality-assurance process.
- Set up the project environment, including IDE, development environment, management tool, configuration-management tool, etc.

Introduction

Software is everywhere. It pervades every aspect of our daily lives, from personal to professional, from entertainment to education. It is very important to have high-quality software that is working correctly and securely in our lives.

In this course, we will discuss techniques for the construction of reliable, efficient, and cost-effective software. We will cover everything from requirements analysis, software design, and programming methodologies to testing procedures, software-development tools, and management issues. This course also features a semester-long group project in which students will design and develop a real-world software system in groups.

Topic 1: Introduction to Software Engineering

Computer-science students usually first learn programming. Every one of you may already have written some programs. Have you developed any software to be used in the real world? Is developing software the same as writing a program? What do you think?

STOP AND CONSIDER

Software Engineering =? Programming

○ Yes ○ No

Software Disasters and Failures

As our lives now depend on software, the failure of software can trigger even near-apocalyptic disasters. Here are some famous examples of software disasters:

- **Ariane Project**—On June 4, 1996, the Ariane 5 launcher exploded about 40 seconds after it started. It was estimated to have lost about $500 million. A root cause is the integer-overflow problem when converting from 64-bit floating-point to 16-bit signed-integer value. The Ariane 5 launch is widely acknowledged as one of the most expensive software failures in history.

- **Radiation Overdose**—A radiation-therapy machine had massive overdoses of gamma rays, causing at least 5 deaths and 15 injuries, in the mid-1980s. It was partly due to a bug in the program that controlled the machine.
- **Bitcoin Hack**—The largest Japanese bitcoin exchange—Mt. Gox, launched in 2010—was hacked in June 2011, and the company stated that it lost over 850,000 bitcoin ($450 million at that time), though 200,000 bitcoin were recovered later.
- **Equifax Mega Breach**—This is one of the biggest exposures, mostly due to a poor quality-assurance culture and practices. The breach compromised the identities of about a third of the US population (145 million people).

Actually, if you search “software disasters” online, you will find a lot of examples from recent years, as well as earlier.

While some may think the software engineer is not as critical as a surgeon, the above examples show that mistakes made by software engineers can cause the loss of millions of dollars and even lives. Software engineering is very important and needs to be taken seriously! While these disasters may be rare, software failures (or software crises) are very common. Any type of failure to meet expectations is a software failure. Failure can happen in the development, maintenance, or use of the software. It can take the form of missing features, being of lower quality, performing poorly, missing deadlines, being over budget, canceling projects, not being adaptable to changed circumstances, detecting errors only after delivery, etc.

STOP AND CONSIDER

What do you think are the possible reasons leading to these failures?

- ☐ Unrealistic or unarticulated project goals
- ☐ Poor project management
- ☐ Inaccurate estimates of needed resources
- ☐ Badly defined system requirements
- ☐ Poor reporting of the project's status
- ☐ Unmanaged risks
- ☐ Poor communication among customers, developers, and users
- ☐ Inability to handle the project's complexity
- ☐ Poor software design methodology
- ☐ Poor coding skills
- ☐ Wrong or inefficient set of development tools
- ☐ Inadequate test coverage
- ☐ Inappropriate (or lack of) software process
- ☐ No security control

Download Responses

The question is, how we can avoid these? Software engineering, as a discipline, tries to answer this question. So, what is software engineering?

Software Engineering IEEE Definition

The Institution of Electrical and Electronic Engineers (IEEE) defines software engineering as “the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.”

From this definition, we can see that software engineering includes software development, software operation, and software maintenance. While we will mainly focus on software development in this course, you should be aware that software operation and maintenance are also very important.

The methods or techniques used in software engineering should be systematic, disciplined, and quantifiable. These are common engineering characteristics. However, these are usually more difficult to achieve in software than in other engineering fields.

Software Engineering is Difficult!

“The Mythical Man-month”, written by Turing Award winner Frederick P. Brooks, Jr., is a collection of classical essays on software engineering. In one of his most famous articles, “No Silver Bullet” (1987), Brooks argued that there is no single development in either technology or management techniques (i.e., no silver bullet) that, by itself, promises even one order of magnitude of improvement in productivity, reliability, or simplicity, because of both essential and accidental difficulties in software engineering. He argued that at least half of problems are essential difficulties, but most of the proposed “radical improvements” only address accidental difficulties. In particular, he listed four essential difficulties and three accidental difficulties.

Essential Difficulties

Essential difficulties are the difficulties inherent in the software. They follow:

- **Complexity**—Software entities are complex. No two parts are the same. If two pieces of code are the same, they should be refactored. There are also too many states and interactions. The complexity is increased nonlinearly, by size.
  - This also creates more difficulties in team communication and project management.
- **Inconformity**—There are no uniform interfaces. All are different, as they are designed by different people, rather than by God.
- **Changeability**—Software offers extreme flexibility. It is easy to change and subject to constant change.
- **Invisibility**—There is no ready geometric representation for software. It is very hard to visualize the whole software.

The challenge in software engineering is to deal with complexity under constant change, without uniform interfaces or visualized representation.

Accidental Difficulties

- The complexities in machine languages are mitigated by the development of high-level languages, though a high-level language can also become a burden.
- The slow turnaround of batch programming is solved by time-sharing.
- The difficulties of using programs together are attacked by the unified programming environments.

Test Yourself

This widely discussed paper on software engineering was written by Brooks in 1987. Please read the paper and think about the following questions:

1. Do you think there are still no silver bullets available nowadays?
2. In his paper, Brooks also proposed several promising solutions. Do you agree? What additional promising solutions have emerged in recent years?

Software-Engineering Concepts and Terminology

Prof. Eric Braude (Braude & Bernstein, 2010) used four Ps to describe basic software-engineering concepts in his book, *Software Engineering: Modern Approaches*. The four Ps follow:

People

People are the most important factor in software engineering. The people involved in a software project are called project stakeholders and include end users, customers, developers, project managers, business managers, technical writers, human-factor specialists, etc.

Product

The products of a software project include not only the executable software and the source code, but also related documentation, such as the following:

- Planning documents
- Requirement documents
- Development documents
- Testing documents
- Customer documents
- Evaluation documents
- Status reports
- Meeting minutes

Project

The project here refers to activities (or maybe phases, in some articles) related to software engineering. It usually includes the following:

- Planning
- Requirement analysis
- Software design
- Implementation
- Testing
- Maintenance

- Communication and management

Process

A software process is a framework for carrying out the activities of a project in an **organized and disciplined** manner. The process defines the order, frequency, deliverable, and criteria of those activities. Waterfall (or sequential) and iterative models are two common types of software process models. Software projects rarely follow a strict waterfall; instead, there are always some kind of iterations among activities. We will discuss these in more detail in the next section.

Software Engineering Ethics

Before we discuss more software-engineering concepts and techniques, we shall first understand and adhere to the ethical and professional obligations of software engineers. The IEEE Computer Society (CS) and Association for Computing Machinery (ACM) produced the Software Engineering Code of Ethics and Professional Practice in 1999, as described below:

> Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:
>
> 1. **PUBLIC**—Software engineers shall act consistently with the public interest.
> 2. **CLIENT AND EMPLOYER**—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
> 3. **PRODUCT**—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
> 4. **JUDGMENT**—Software engineers shall maintain integrity and independence in their professional judgment.
> 5. **MANAGEMENT**—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
> 6. **PROFESSION**—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
> 7. **COLLEAGUES**—Software engineers shall be fair to and supportive of their colleagues.
> 8. **SELF**—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Test Yourself

Software engineering is the same as programming.

False

True

Test Yourself

What are the essential difficulties described by Brooks?

Complexity, changeability, inconformity, invisibility

Test Yourself

The problem of IDE is an accidental difficulty described by Brooks.

✓

False

True

DE solves the accidental difficulty of using different programs together.

Test Yourself

What are the four Ps used by Braude?

People, product, project, process

Test Yourself

Please provide an example of when the code of ethics may be violated.

Topic 2: Software Process

In the previous section, we briefly introduced the concept of software-development process. A software-development process is a structure imposed on the development of a software product, which prescribes the order and frequency of activities (sometimes called phases), specifies criteria for moving from one phase to the next, and defines the deliverables of the project. When a software process is applied properly, it can have a positive effect on meeting deadlines, reducing cost and producing high-quality software.

Phases (Activities)

Here are a number of common activities or phases in the software life cycle:

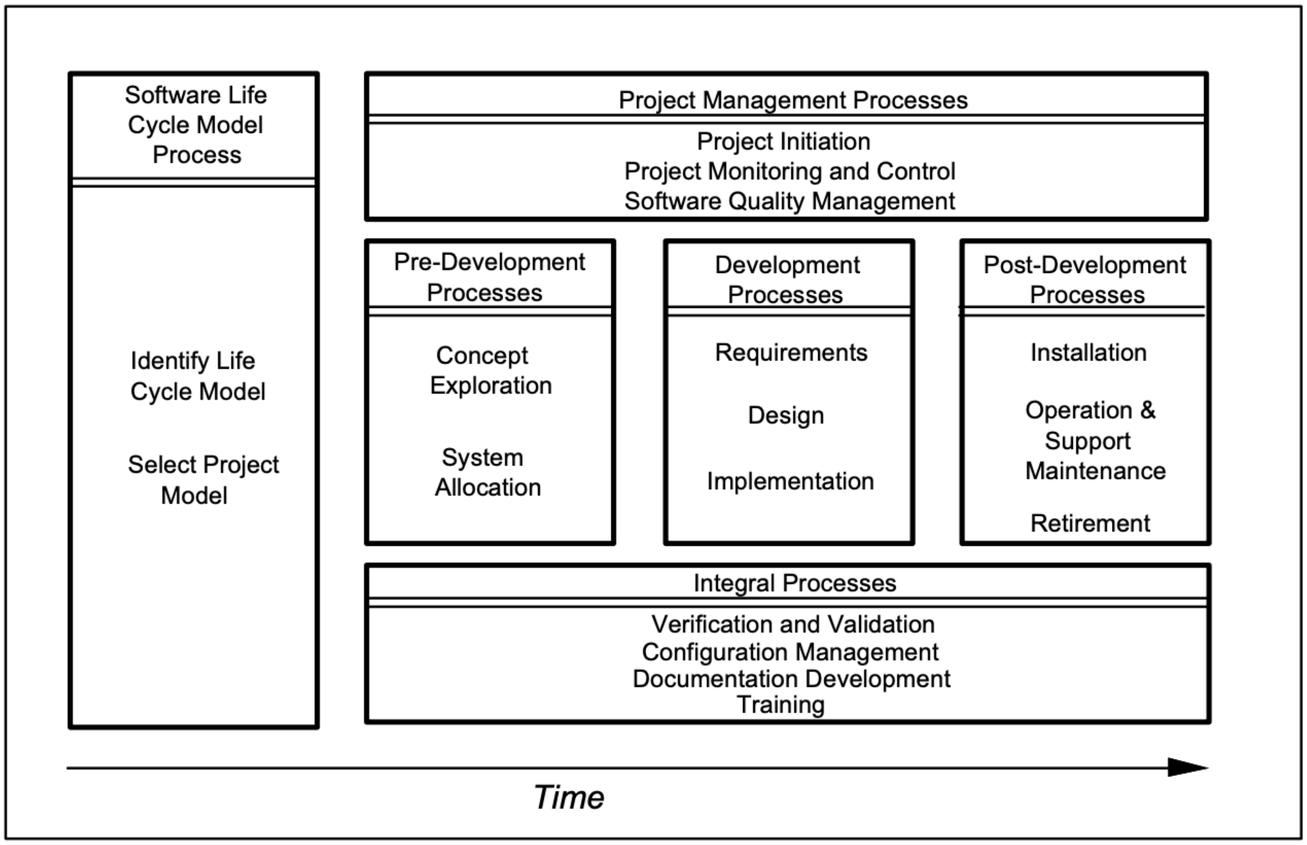
- **Inception**—Define the project scope, the major functionalities of the software, its target customers, and related systems. In this phase, the high-level idea about the software is formulated.
- **Planning**—Estimate resources, the cost, and work items needed. Set up a schedule and high-level activities. Conduct a feasibility study and make a management plan, including project management, configuration management, and quality management. The planning documents should be updated throughout the life cycle.
- **Requirements analysis**—Gather and analyze functional and nonfunctional requirements.

- **Software design**—Include both high-level, architectural design and low-level, detailed design.
- **Implementation**—Write and refactor code.
- **Testing**—Include unit-testing, integration testing, system testing, etc.
- **Maintenance**—Repair defects and enhance quality.

There are also a number of umbrella activities that should be conducted throughout the software life cycle. These include project management, configuration management, quality management, risk management, etc.

IEEE published the *Guide for Developing Software Life Cycle Processes* in 1996. Figure 1 shows activities, processes, and their connections as defined in the document. You can see that the activities described above are similar to the ones defined in this document.

IEEE Guide for Developing Software Life Cycle Processes

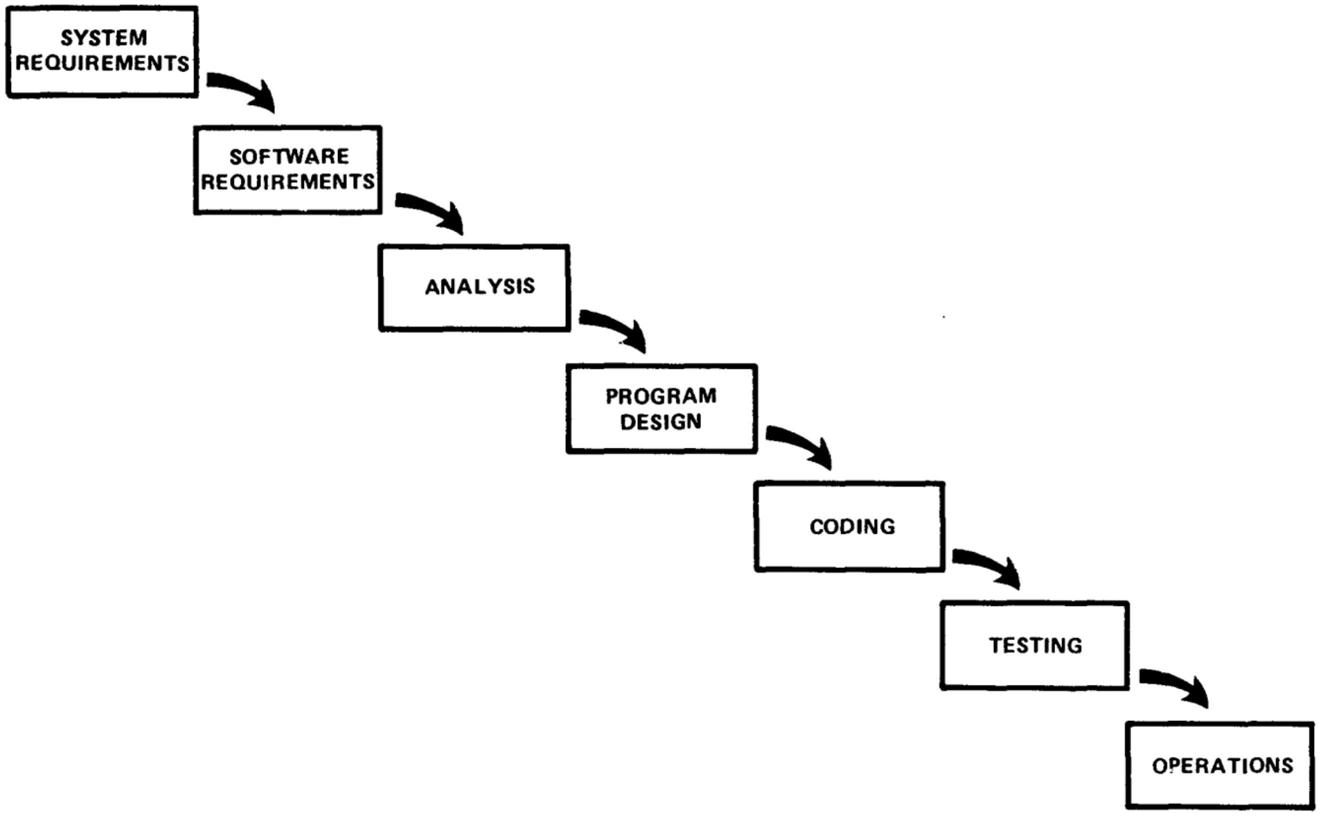

Software Life Cycle Model Process
Identify Life Cycle Model
Select Project Model
Project Management Processes
Project Initiation
Project Monitoring and Control
Software Quality Management
Pre-Development Processes
Concept Exploration
System Allocation
Development Processes
Requirements
Design
Implementation
Post-Development Processes
Installation
Operation & Support
Maintenance
Retirement
Integral Processes
Verification and Validation
Configuration Management
Documentation Development
Training
Time

Software Process Models

Software process models (or software life cycle models) define some specific software processes. There are several well-known models. The waterfall model is known as the basic sequential model. Spiral, unified process, extreme programming, and Scrum are iterative and incremental models.

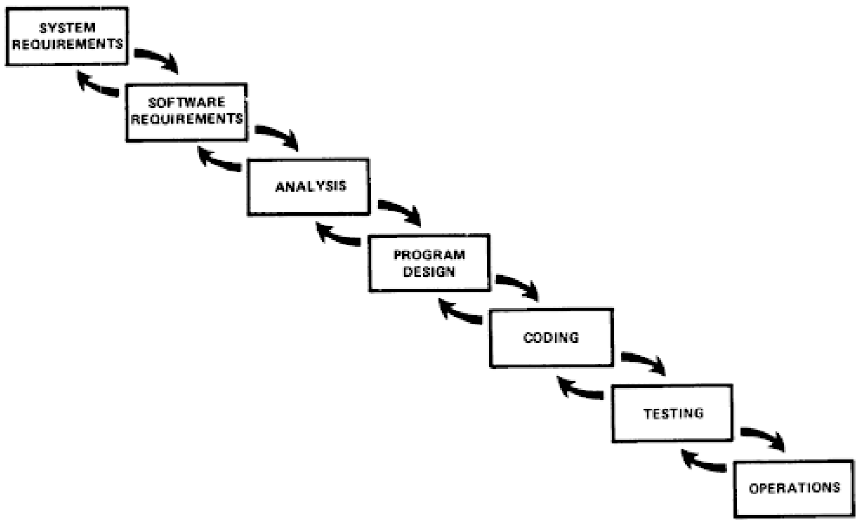
Waterfall

The waterfall (sequential) model was initially described by Winston W. Royce (1970) in his paper “Managing the Development of Large Software Systems” as an example of a flawed software-development model. That model defines a strict sequence, from requirements to analysis, design, coding, testing, and operations. Only when the previous phase is completed and verified can one move to the next phase.



Waterfall Model (Royce, 1970)

The pure waterfall model is very strict and not practical. Different modifications are proposed to add some feedback from later phases to previous phases to improve the model. For example, Royce added a feedback arrow from each phase to its predecessor.



Waterfall Model with Feedback (Royce, 1970)

Here are some pros and cons of the waterfall model or its variants.

Pros

- Simple and easy to use
- Practiced for many years
- Easy to manage
- Facilitates allocation of resources
- Works well for smaller projects, of which the requirements are very well understood

Cons

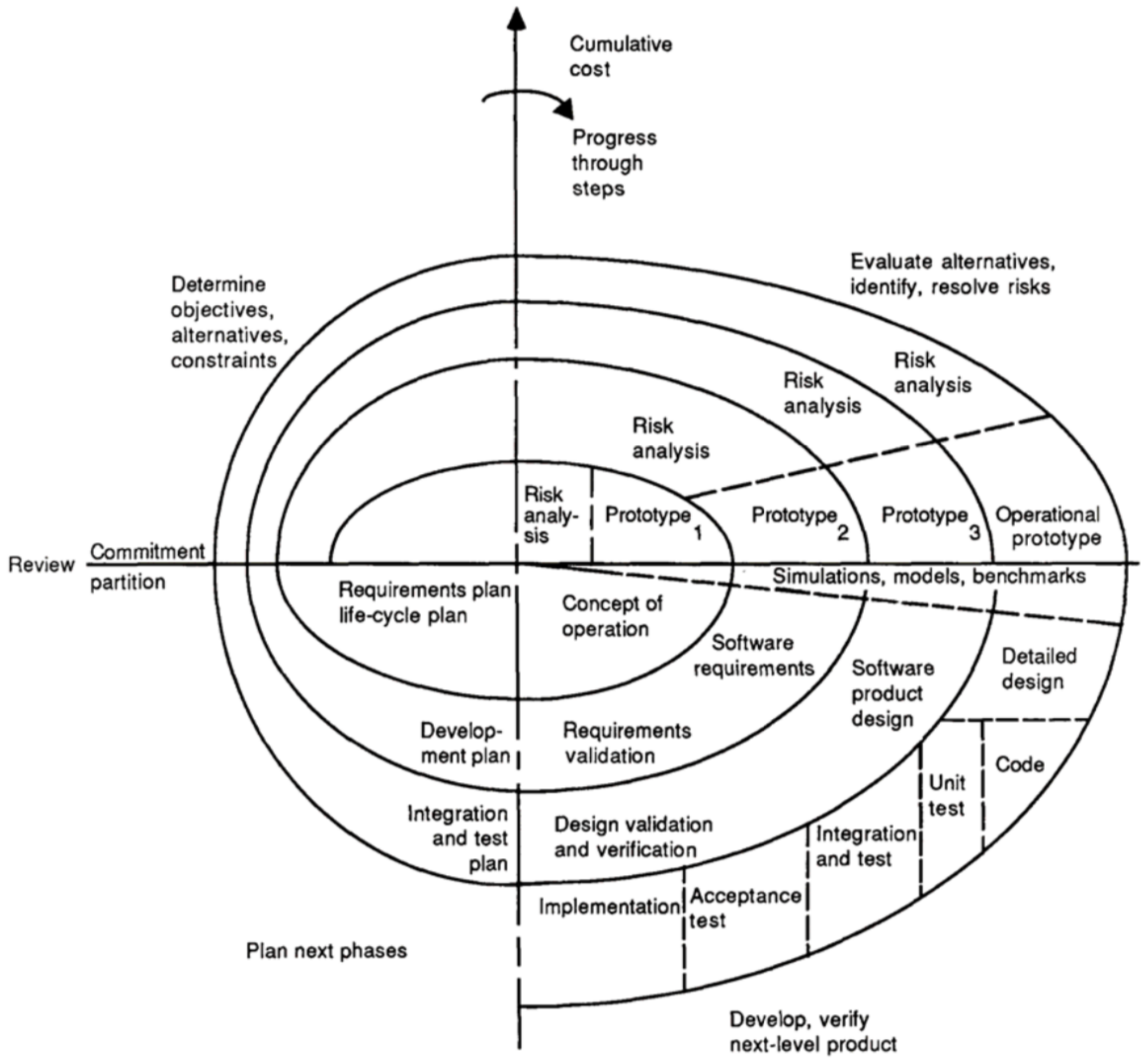
- Requirements must be known up-front
- Hard to estimate reliably
- No feedback on system from stakeholders until after testing phase
- Major problems aren't discovered until late in the process
- Lack of parallelism
- Inefficient use of resources

Spiral Model

Iterative models are proposed to address the problems in the sequential model. Though using the waterfall model is not recommended, iterative models repeatedly execute the waterfall phases, in whole or in part, resulting in refinement of the requirements, design, and implementation.

In an iterative and incremental model, an iteration is relatively small. Operational code is produced at the end of each iteration that supports a subset of the final product's functionality and features. The artifacts keep evolving after each iteration and are considered complete when the software is released.

The spiral model is one of the earliest and best known iterative process models. It was proposed by Barry Boehm (1986) in his paper "A Spiral Model of Software Development and Enhancement." The most distinct feature of the spiral model at that time was its risk-driven approach.



Spiral Model (Boehm, 1986)

A project starts at the center, and each cycle of the spiral represents one iteration. Each iteration tries to identify and resolve risks by using prototypes and other means. A prototype can be a model, a simulation, a benchmark, or a partial implementation of the target application. The goal of each cycle is to reduce the risk in the system definition and implementation. It combines the features of the prototyping and the waterfall model. When a satisfactory design is reached, it follows the waterfall model on coding, integration, testing, and release.

To better distinguish spiral models from "hazardous spiral look-alikes," Boehm lists six characteristics common to all authentic applications:

- Defines artifacts concurrently
- Performs four basic activities in every cycle
- Risk determines level of effort
- Risk determines degree of details
- Uses anchor-point milestones
- Focuses on the system and its life cycle

Its pros and cons are summarized below.

Pros

- Risks are well managed early and throughout the process
- Planning is built into the process
- Software evolves as the project progresses
- Better for large projects that have more risks and need more planning

Cons

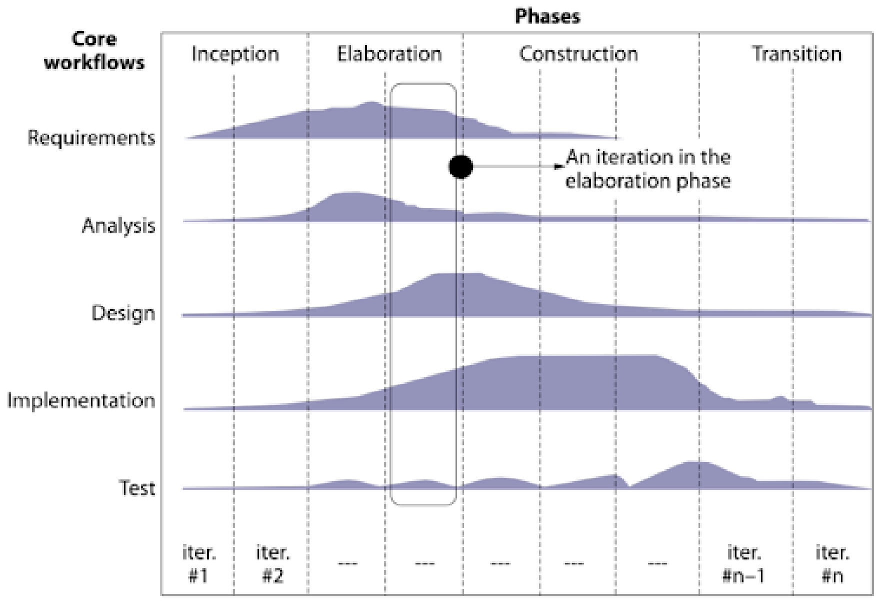
- Can be quite expensive to implement
- Requires specialist expertise, particularly in risk analysis
- End of project may be difficult to define in advance
- Not an ideal fit for smaller or low-risk projects

Unified Process

Unified process (UP) was designed by the “three amigos”—Ivar Jacobson, Grady Booch, and James Rumbaugh (1999)—as their vision of a standardized software-development process. It represents a major elaboration and refinement of the spiral model. The designers stated that UP is a generic process framework that can be specialized for a very large class of software systems, for different application areas, types of organizations, competence levels, and project sizes. The trio also developed the **Unified Modeling Language (UML)**, an integral part of the unified process. It has the following characteristics:

- **Use-case driven**—A use case defines a functional requirement. The use-case model defines all use cases, as well as their relationships with each other and with different user roles. It describes the complete functionality of the system. Use cases drive the design, implementation, and testing of the whole software system.
- **Architecture-centric**—An architecture is a view of the whole design that highlights the important structure and characteristics without much detail. It is the foundation of the whole system. It should be able to accommodate the implementation of all required use cases. Both the architecture and the use cases must evolve in parallel.
- **Iterative and incremental**—Iterations are steps in the work flow, and increments are degrees of growth in the product. In each iteration, a mini-project will go through requirements, analysis, design, implementation, and testing, in whole or in parts, to result in some increments toward the final product. These may be refinements of the existing components or additions of new components. The iteration should be controlled to better identify and reduce the risks early in the software life cycle.

The UP repeats over a series of cycles making up the life of a system. Each cycle concludes with a product release to customers. Each cycle consists of four phases. Each phase is further subdivided into iterations. Each iteration involves five core work flows in different degrees, as shown as the following diagram:



UP Work Flows (Jacobson, Booch, & Rumbaugh, 1999)

The four phases follow:

- **Inception**-Tasks in this phase include performing a feasibility study, making business cases, establishing the product vision and scope, estimating the cost and setting major milestones, assessing critical risks, and building one or more prototypes. The result of this phase is a simplified use-case model that describes the major functionalities and a tentative architecture that outlines the most crucial subsystems. The most important risks are also identified and prioritized. The next phase is planned.
- **Elaboration**—Tasks in this phase include specifying most use cases in greater detail, building the architectural baseline, implementing core architecture, refining risk assessment and resolving the highest-risk items, defining metrics, refining the project plan, and making the detailed plan for the beginning construction iterations. By the end of this phase, the use-case model, architecture, and plans are all stable.
- **Construction**—In this phase, the remaining requirements are elicited and analyzed. The architecture baseline grows to become the full-fledged system. All requirements are implemented iteratively based on the previous architecture baseline, then tested thoroughly and prepared for system deployment.
- **Transition**—In this phase, the product is released and deployed. The focus will be maintenance and support, such as performing beta tests, correcting defects, creating user manuals, training end users, providing customer support, and collecting lessons learned.

The construction phase typically needs more than 50% of the overall time, and the elaboration phase, maybe 25%. The initialization and transition phases usually only use 10% each or less.

The pros and cons of UP are summarized as follows.

Pros:

- Most aspects are accounted for
- Is mature

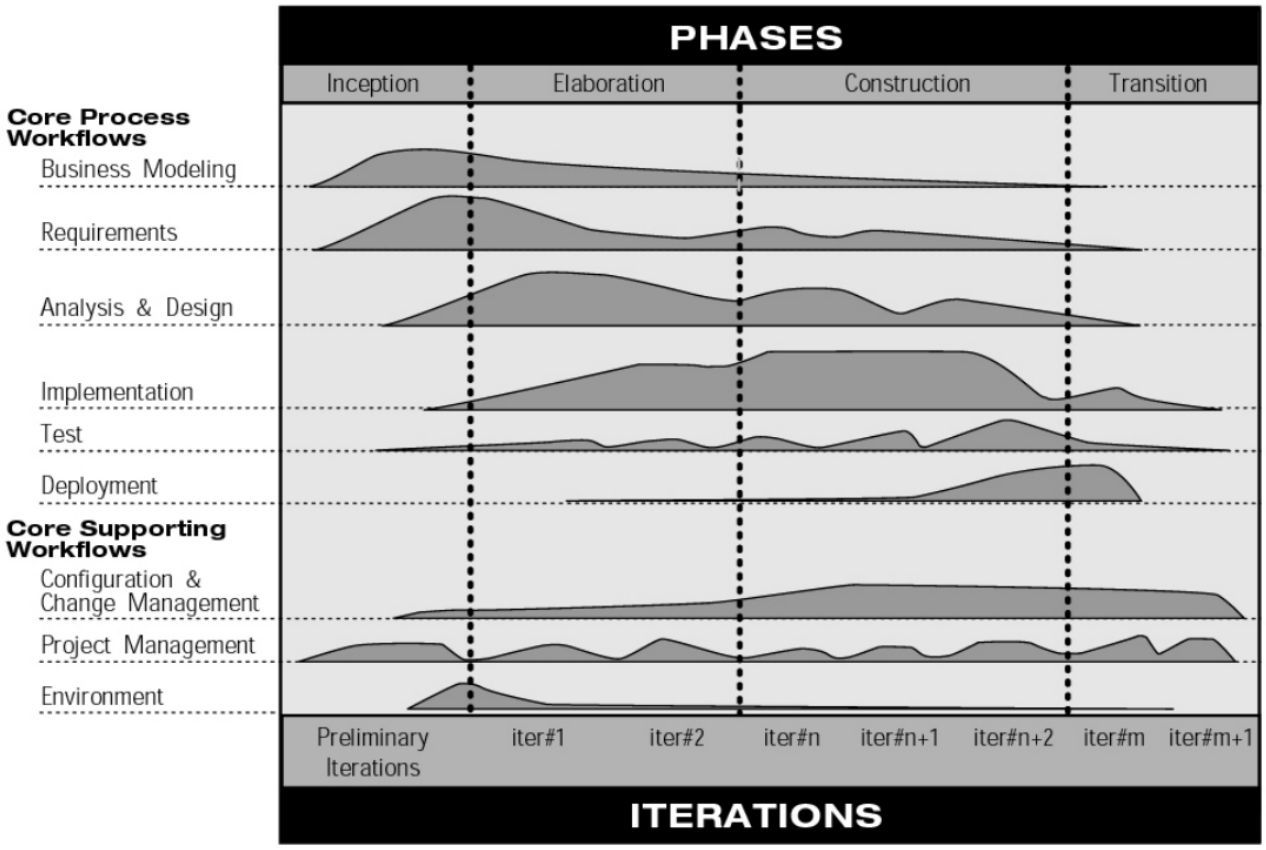
Cons

- Originally for medium-sized to large projects
- May be overkill for small projects

Rational Unified Process (RUP)

RUP is a specific implementation of UP. It is also a customizable process framework created by the Rational Software Corporation, a division of IBM since 2003. RUP promotes the use of UML and a lot of tools to automate tasks.

There are nine work flows in RUP: six core "engineering" work flows and three supporting work flows.



RUP Work Flows

Test Yourself

What are the similarities and differences between spiral and UP models?

Test Yourself

Which of these two process models (or frameworks) is better suited to your project, in your opinion, and why?

Test Yourself

The Obamacare health-insurance shopping–website project was a big failure in that only six people in the entire country got it to work well enough to select coverage on the day it launched. Many attribute this failure to poor project management. If you had to choose between UP and spiral, which one would you choose and why? What are the possible pitfalls of using that one?

Process Improvement and CMMI

A proper software process can help improve the software's productivity and quality when it is implemented correctly. Process improvement aims to proactively analyze and improve upon existing processes across a project, a division, or an entire organization, to enhance the quality of the result and/or reduce the cost and time.

Capability Maturity Model Integration (CMMI) is a classic process-improvement approach. It was developed by Carnegie Mellon University (CMU), administered by the CMMI Institute (a subsidiary of ISACA), and required by many United States Department of Defense (DoD) and U.S. government contracts. Its main focus is to help organizations measure and improve their performance. The CMMI Institute claims that CMMI is a proven set of global best practices that drives business performance by building and benchmarking key capabilities.

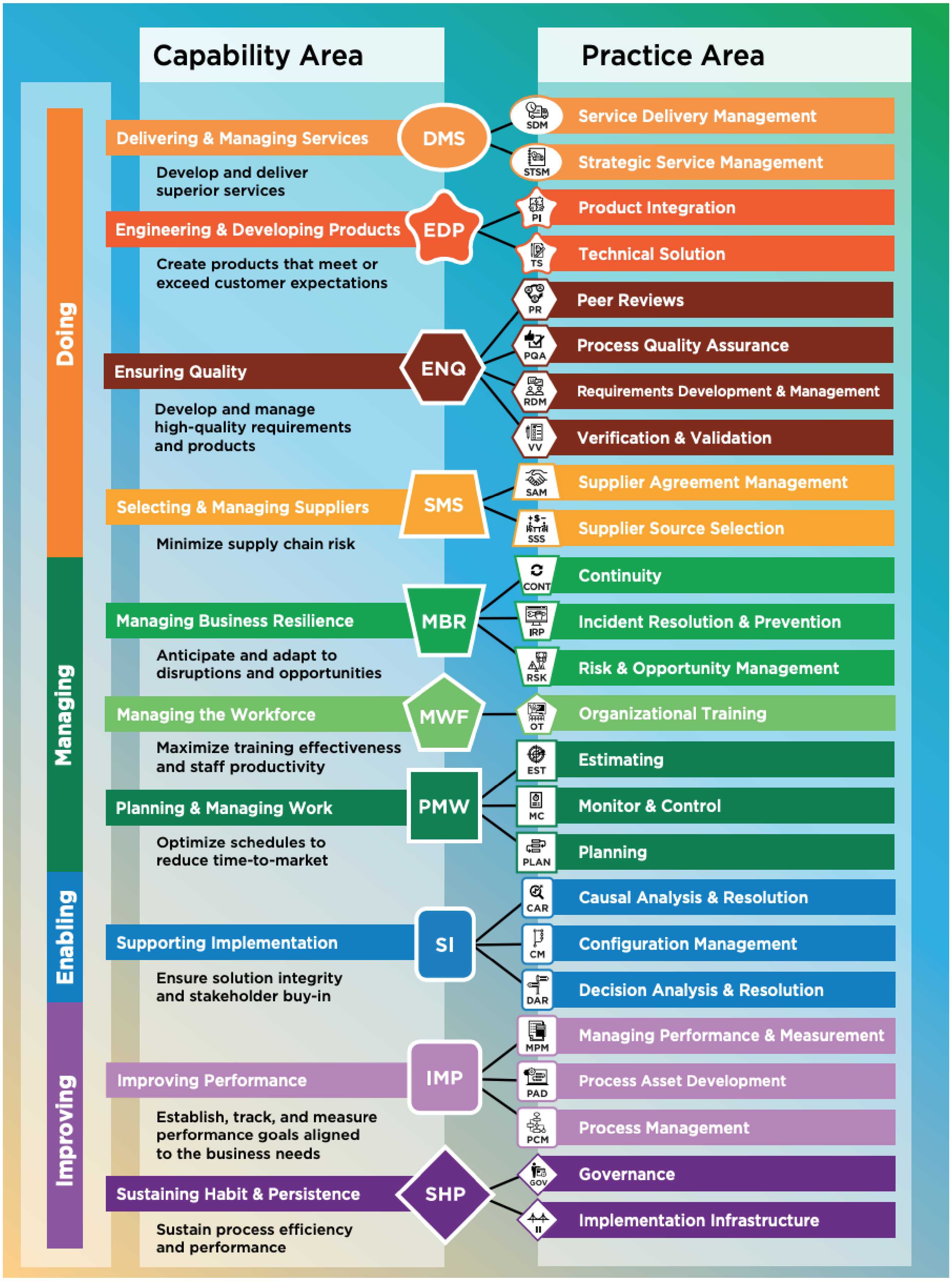
The CMMI model has been in existence since 1991. The original was published in 2000 by the CMMI Product Team. The new version, CMMI 2.0, was published in 2018. CMMI 2.0 model defines a set of practice areas that are grouped into four capability categories: doing, managing, enabling, and improving. The practices are organized into practice groups, labeled Level 0 (lowest) to Level 5 (highest), which provide an evolutionary path to performance improvement, as shown in the following table. Each level builds on the previous levels by adding new functionality or rigor, resulting in increased capability and maturity. Capability levels apply to an organization's performance and process-improvement achievements in individual practice areas. Maturity levels represent a staged path for an organization's performance and process-improvement efforts based on predefined sets of practice areas.

Capability and Maturity Levels

Levels	**Capabilities**	**Maturity**
Level 5: Optimizing		Stable and flexible. (The organization is focused on continuous improvement and is built to pivot and respond to opportunity and change. The organization's stability provides a platform for agility and innovation.)
Level 4: Quantitative		Measured and controlled. (The organization is data driven, with quantitative performance-improvement objectives that are predictable and align with the needs of internal and external stakeholders.)
Level 3: Defined	Focuses on achieving both project and organizational performance objectives	Proactive, rather than reactive. (Organization-wide standards provide guidance across projects, programs, and portfolios.)
Level 2: Managed	Identifies and monitors progress toward project performance objectives	Managed at the project level. (Projects are planned, performed, measured, and controlled.)

Capability and Maturity Levels		
Levels	**Capabilities**	**Maturity**
Level 1: Initial	Addresses performance issues	Unpredictable and reactive (Work gets completed but is often delayed and over budget.)
Level 0: Incomplete	Inconsistent performance	Ad hoc and unknown (Work may or may not get completed)

The following diagram from the *CMMI Guide* shows the capability and practice areas:


Capability Area
Practice Area
Doing
Managing
Enabling
Improving
Delivering & Managing Services
DMS
Develop and deliver superior services
Engineering & Developing Products
EDP
Create products that meet or exceed customer expectations
Ensuring Quality
ENQ
Develop and manage high-quality requirements and products
Selecting & Managing Suppliers
SMS
Minimize supply chain risk
Managing Business Resilience
MBR
Anticipate and adapt to disruptions and opportunities
Managing the Workforce
MWF
Maximize training effectiveness and staff productivity
Planning & Managing Work
PMW
Optimize schedules to reduce time-to-market
Supporting Implementation
SI
Ensure solution integrity and stakeholder buy-in
Improving Performance
IMP
Establish, track, and measure performance goals aligned to the business needs
Sustaining Habit & Persistence
SHP
Sustain process efficiency and performance
SDM Service Delivery Management
STSM Strategic Service Management
PI Product Integration
TS Technical Solution
PR Peer Reviews
PQA Process Quality Assurance
RDM Requirements Development & Management
VV Verification & Validation
SAM Supplier Agreement Management
SSS Supplier Source Selection
CONT Continuity
IRP Incident Resolution & Prevention
RSK Risk & Opportunity Management
OT Organizational Training
EST Estimating
MC Monitor & Control
PLAN Planning
CAR Causal Analysis & Resolution
CM Configuration Management
DAR Decision Analysis & Resolution
MPM Managing Performance & Measurement
PAD Process Asset Development
PCM Process Management
GOV Governance
II Implementation Infrastructure

CMMI Capability and Practice Areas

Using the practice areas "technical solution" (TS) and "process management" (PM) as examples, the following table shows how CMMI defines the practices at different levels. Note that the total number of levels defined in each practice area varies from two to five. Here, TS defines three different levels of practices, and PM defines four.

Level	**Technical Solution (TS) Practice Summary**	**Process Management (PM) Practice Summary**
1	• Build a solution to meet requirements.	• Develop a support structure to provide process guidance, identify and fix process problems, and continuously improve processes. • Appraise the current process implementation and identify strengths and weaknesses. • Address improvement opportunities or process issues.
2	• Design and build a solution to meet requirements. • Evaluate the design and address identified issues. • Provide guidance on use of the solution.	• Identify improvements to the processes and process assets. • Develop, keep updated, and follow plans for implementing select process improvements.
3	• Develop criteria for design decisions. Develop alternative solutions for selected components. • Perform a build, buy, or reuse analysis. Select solutions based on design criteria. • Develop, keep updated, and use information needed to implement the design. • Design solution interfaces or connections using established criteria.	• Develop, keep updated, and use process-improvement objectives traceable to the business objectives. • Identify processes that are the largest contributors to meeting business objectives. Explore and evaluate potential new processes, techniques, methods, and tools to identify improvement opportunities. • Provide support for implementing, deploying, and sustaining process improvements. • Deploy organizational-standard processes and process assets. • Evaluate the effectiveness of deployed improvements in achieving process-improvement objectives.
4		• Use statistical and other quantitative techniques to validate select performance improvements against proposed improvement expectations, business objectives, or quality and process performance objectives.

Besides the model, the CMMI product suite also provides appraisal methods, training and certification, systems and tools, and adoption guidance. Their 2018 adoption report shows that appraisals increased by 16% from 2008 to 2018, and 11% of reported appraisals were of high maturity (Level 4 or 5).

The CMMI model mostly deals with what processes should be implemented, not so much with how they can be implemented exactly. Some advocate it while others criticize it. Applying CMMI will not guarantee increased performance in every organization. The CMMI can seem overwhelming and a burden to implement sometimes. It has also been used by some companies just as a way to bid on a contract without really following it. Correctly understanding CMMI and properly adopting it in your specific context may be the key to success with CMMI. Read the *CMMI V2.0 Quick Reference Guide* for more details.

Test Yourself

What are the main characteristics of the spiral model?

Iterative, risk-driven, prototyping

Test Yourself

What are the main characteristics of the UP model?

Iterative and incremental, use-case driven, architecture centric

Test Yourself

Why are iterative models better than sequential models in general?

More adaptable to changes, help divide and conquer the complexity, provide earlier feedback, enable parallelism, etc.

Test Yourself

The CMMI model defines different levels of capability and maturity. What does high maturity mean?

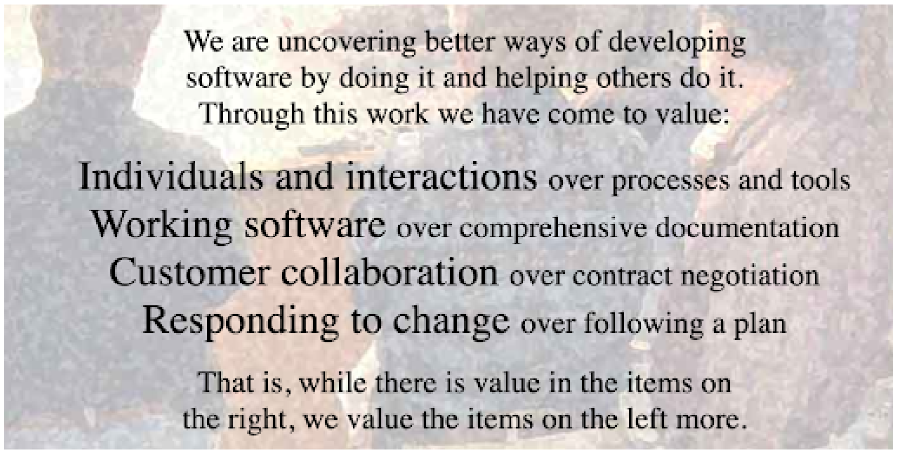
It means that the organization has consistent good performance across different projects and meets the needs of different stakeholders.

Agile Methodology

Agile software development is an iterative and incremental approach, too. It particularly attempts to address the problems in traditional, process-heavy, documentation-heavy models. The goal is to make the software-development process simpler, lighter, quicker, and more adaptive to changes.

Agile Manifesto

The term “agile” (sometimes written “Agile”) was popularized by the publishing of the Agile “Manifesto” in 2001, though the values and principles are derived from a range of software-development frameworks from the 1990s, such as RAD (rapid application development, 1991), Scrum (1995), and XP (eXtreme programming, 1996). "The Manifesto for Agile Software Development" is shown in the following picture:



We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Agile "Manifesto"

The manifesto was created by 17 software developers: Kent Beck, Ward Cunningham, and Ron Jeffries, the creators of XP; Dave Thomas and Andrew Hunt, the authors of *The Pragmatic Programmer*; Jeff Sutherland and Ken Schwaber, the creators of Scrum; Jim Highsmith, the creator of adaptive software; Martin Fowler, the author of Refactoring; Steve Mellor, the developer of Executable UML; and Alistair Cockburn, Robert C. Martin, Mike Beedle, Arie van Bennekum, James Grenning, Jon Kern, and Brian Marick.

Agile Principles

In addition, the creators stated 12 principles that are based on the Agile “Manifesto”:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

We can see here that each principle addresses one or more values proposed in the Agile “Manifesto”. These principles help guide the development of Agile frameworks, practices, and methodologies. Some of the authors formed the Agile Alliance, a nonprofit organization that promotes software development according to the manifesto's values and principles. You can find a lot of resources on the Agile Alliance website.

There are several frameworks that follow Agile values and principles, such as eXtreme programming, Scrum, and Kanban. Those frameworks define a number of practices that help achieve agility. Extreme programming mostly addresses technical practices, and Scrum and Kanban, management practices.

eXtreme Programming (XP)

Extreme programming was invented by Ward Cunningham, articulated by Kent Beck, and implemented by Ron Jeffries in 1996 for a C3 project at Chrysler. Kent argued that it is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop software. It focuses on the following values:

- Communication
- Simplicity
- Feedback
- Courage
- Respect

The original practices are listed below. You can see how these practices were originally defined on Ron Jeffries' Extreme Programming website:

1. Whole Team
2. Planning Game
3. Small Releases
4. Customer Tests
5. Simple Design
6. Pair Programming
7. Test-Driven Development
8. Design Improvement (Refactoring)
9. Collective Code Ownership
10. Continuous Integration
11. Coding Standard
12. Metaphor
13. Sustainable Pace

The XP practices have changed a bit since they were initially introduced. Kent lists 13 primary practices and 11 corollary practices in his book *Extreme Programming Explained: Embrace Change* (second edition; 1999). Don Wells, who was also involved in the C3 project, lists some of these practices in different phases on his website that explains XP. Based on their ideas, we categorize all practices as part of different phases, as in the following table:

	Primary Practices	**Corollary Practices**
Planning	1. Stories 2. Weekly cycle 3. Quarterly cycle	

	Primary Practices	Corollary Practices
Managing	1. Sit together 2. Whole team 3. Informative workspace 4. Energized work 5. Slack	1. Real customer involvement 2. Team continuity 3. Negotiated scope contract 4. Pay per use 5. Shrinking teams
Designing	1. Incremental design	1. Root-cause analysis
Coding	1. Pair programming 2. Ten-minute build	1. Shared code 2. Single code base
Testing	1. Continuous integration 2. Test-first programming	1. Code and tests
Deployment		1. Incremental deployment 2. Daily deployment

Here is an interesting anti-XP article by Yossi Kreinin. Do you agree with his comments (or some of his comments)?

Test Yourself

What XP practices do you think are good to have? Which are not?

Test Yourself

Kent argued that these practices should be together. What do you think?

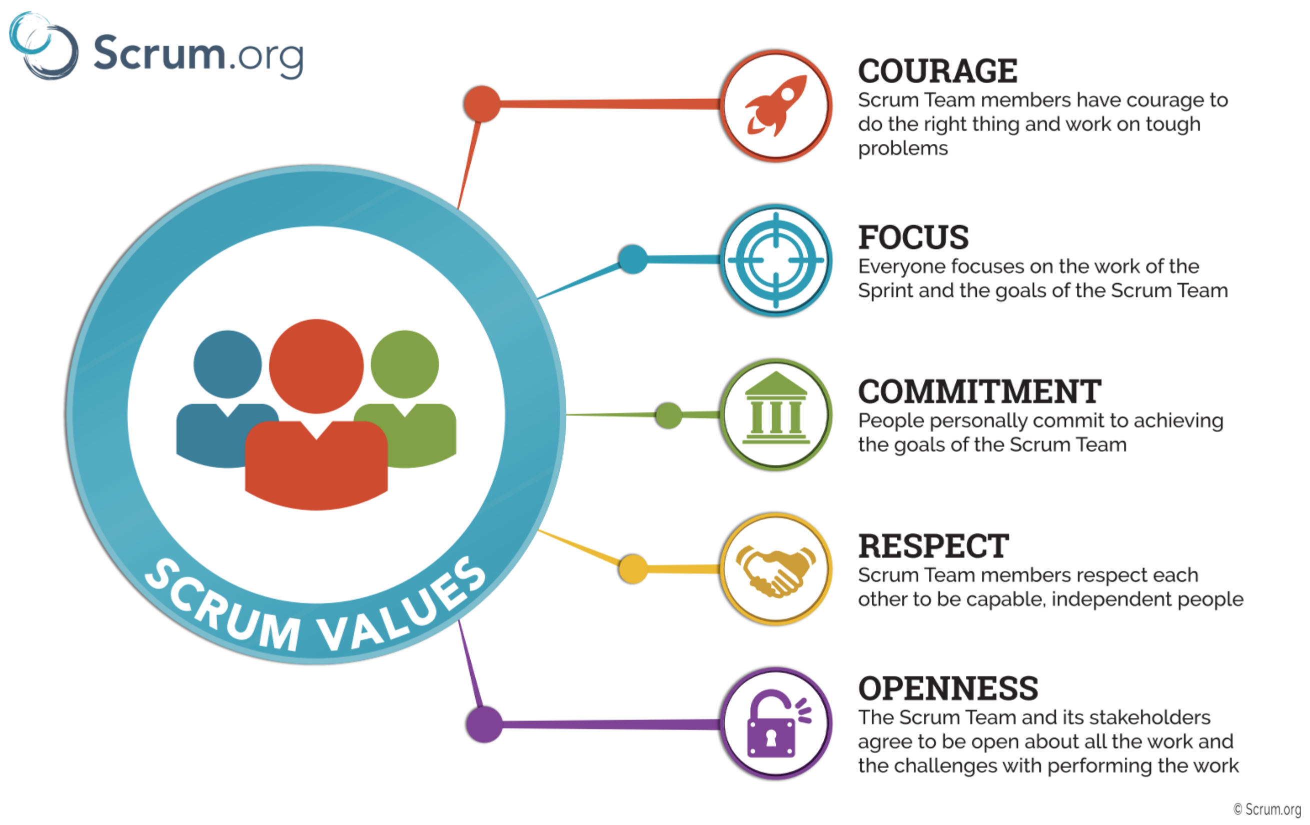
Test Yourself

What practices does your group plan to use in your group project? What are obstacles to applying these practices?

Scrum

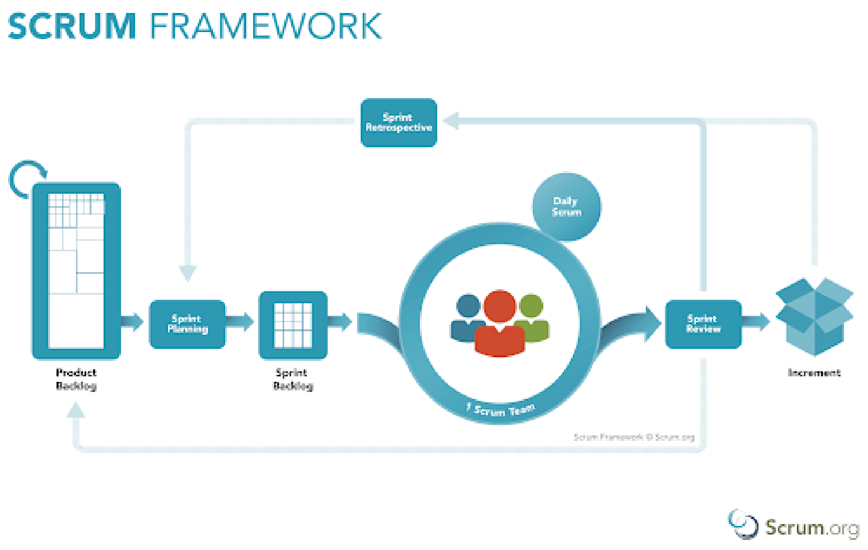
Scrum is the most widely used Agile framework nowadays. Unlike XP, it mainly focuses on the work flow and management practices. The Scrum website states that it has 12 million users—not only in software-development and IT fields, but also fields such as oil companies. Sometimes when people say “Agile,” they actually mean Scrum. Scrum is a simple framework that is designed for effective team collaboration and can adapt well to changes. It is defined by a loose set of activities and empowers the self-organized development team to define and execute tasks that are necessary for success. It was developed in the early 1990s by Ken Schwaber and Jeff Sutherland (who also signed the Agile “Manifesto”). Together, they wrote The Scrum Guide to explain Scrum clearly and succinctly. You can find the latest version of the book (2017) at Scrum Guides.

The following diagram from the Scrum website shows Scrum values, which are similar to those of XP.


Scrum.org
SCRUM VALUES
COURAGE
Scrum Team members have courage to do the right thing and work on tough problems
FOCUS
Everyone focuses on the work of the Sprint and the goals of the Scrum Team
COMMITMENT
People personally commit to achieving the goals of the Scrum Team
RESPECT
Scrum Team members respect each other to be capable, independent people
OPENNESS
The Scrum Team and its stakeholders agree to be open about all the work and the challenges with performing the work
© Scrum.org

Scrum Values

The Scrum framework is shown in the following diagram:



Scrum Framework

The events, artifacts, and roles defined in the Scrum framework are listed in the following table and explained below:

Scrum Events	Scrum Artifacts	Scrum Roles

• Sprint • Sprint Planning • Daily Scrum • Sprint Review • Sprint Retrospective	• Product Backlog • Sprint Backlog • Increment	• Scrum Master • Product Owner • Development Team

The requirements are written as user stories and stored in the **product backlog**, the single source of requirements. It keeps evolving with the product and environment. It constantly changes to identify what the product needs in order to be competitive and useful. The product owner is responsible for managing the product backlog.

A **sprint** is a time-boxed (usually one month or less) iteration during which a **product increment** is created. In each sprint, a subset of top-priority items in the product backlog are taken out as the **sprint backlog**. What items should be moved from the product backlog and what the goal of the current sprint is are discussed and decided through **sprint planning**.

The **daily scrum** is a 15-minute time-boxed event held every day of the sprint to synchronize activities and plan for the day. In this meeting, the team will inspect together the work in the sprint backlog and adapt their plan to accomplish the sprint goal and create the anticipated increment by the end of the sprint.

A **sprint review** is held at the end of the sprint to inspect the increment and adapt the product backlog if needed. The purpose is to elicit feedback and foster collaboration among the scrum team and stakeholders.

The sprint retrospective occurs after the sprint review and prior to the next sprint planning. During the sprint retrospective, the team discusses the following:

- What went well in the sprint
- What could be improved
- What they commit to improving in the next sprint

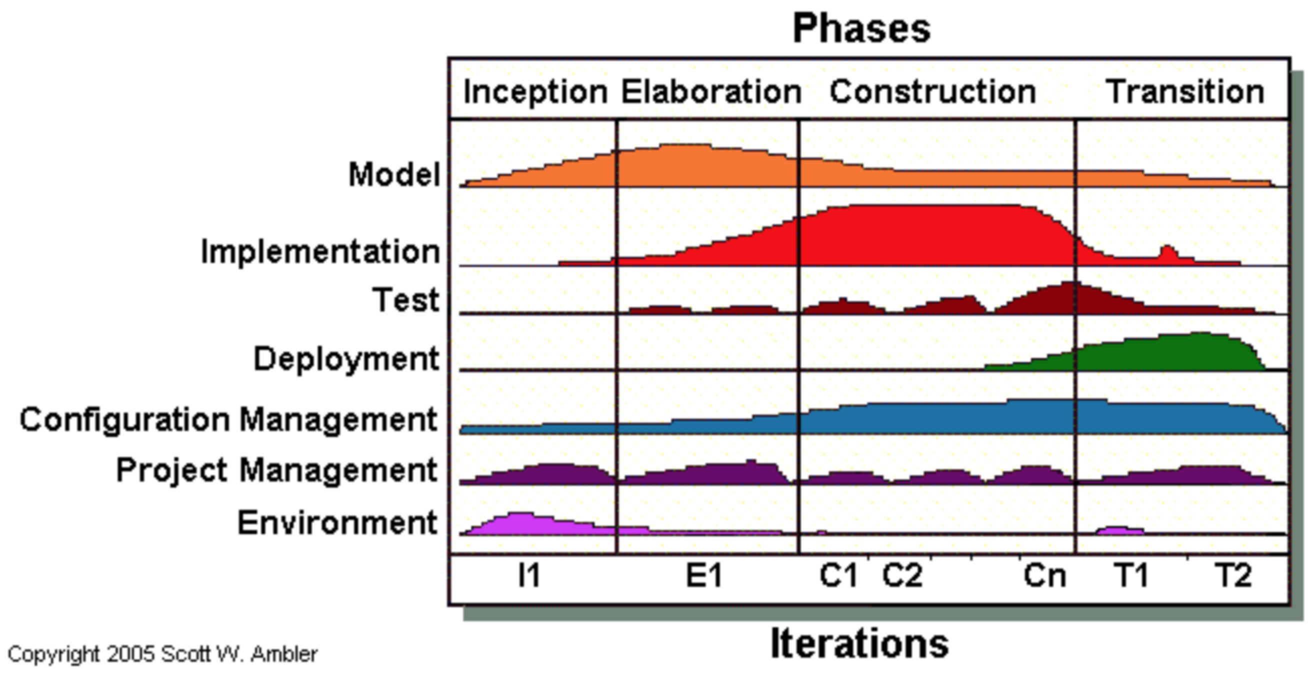
The purpose is to improve the development process and practices to make it more effective and enjoyable for the next sprint.

The scrum team consists of a product owner, a development team, and a scrum master.

- **Product owner**—The only person who is responsible for managing the product backlog. The product owner usually works with the development team and customer to decide to add, delete, edit, detail, or order the items in the backlog.
- **Scrum master**—A servant-leader for the scrum team. The scrum master facilitates communication among the team members and with outsiders to maximize the productivity and value.
- **Development team**—Should be self-organizing and cross-functional. They should have all necessary skills and can choose how to accomplish work. The whole team is accountable for the work to be completed.

Agile Unified Process (AUP)

Both UP (or RUP) and several Agile frameworks (such as XP and Scrum) were created in the 1990s as iterative and incremental process frameworks. They all try to solve the problems in the waterfall model. Compared with UP (or RUP), Agile frameworks are much lighter in process and documentation. AUP was proposed as a simplified version of RUP to add agility to UP. It is something between XP and RUP. It integrates many of the Agile practices, such as “test driven” and “refactoring,” into the simplified RUP work flows. The AUP work flow is shown in the following diagram:



AUP Work Flows

Scaling Agile

The pros and cons of Agile are summarized as below:

Pros

- Motivates developers
- Thoroughly tested, mostly
- Easier to estimate each cycle
- Responsive to customer
- Always demonstrable software

Cons

- Hard for new participants
- Sensitive to individuals
- Hard to estimate full job
- Limits team size

Pure Agile works best for small to midsized, co-located developer teams of five to eight members. Can Agile be used in large projects by multiple developer teams with distributed locations?

Narendra Shrivastava (2015) examined the principles of scaling Agile, including the concept of the scrum of scrums. The paper examines the unique features and the pros and cons of the three leading frameworks (Scaled Agile Framework, Disciplined Agile Delivery, and Large-Scale Scrum), and presents guidelines to choose a scaled solution based on the team's and project's needs.

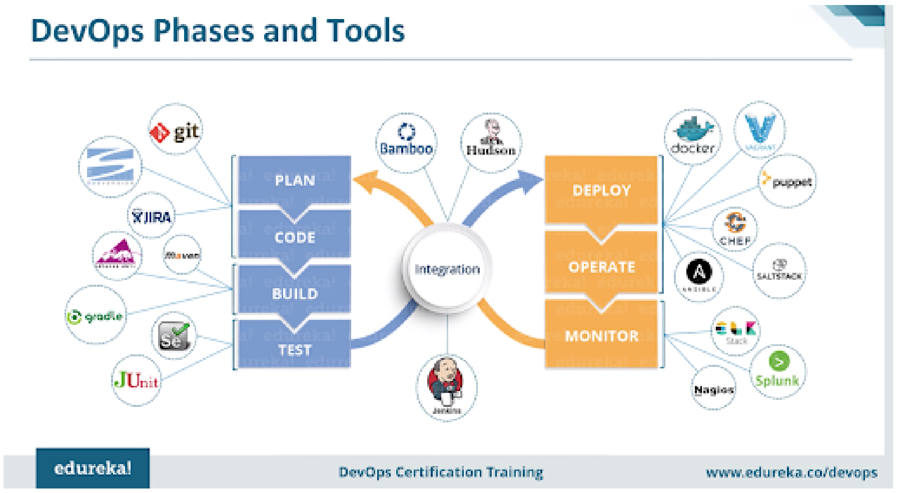
Test Yourself

Can Agile mitigate some essential or accidental difficulties?

DevOps

“DevOps” has become another buzzy word in recent years. It is a term that combines software development (Dev) and information technology operations (Ops) to represent a closer relationship between these two. It usually refers to a set of practices, roles, or processes to improve the communication between developers and IT operators in order to shorten the entire life cycle—including both development and deployment—when delivering new features, fixes, or updates. DevOps promotes the use of tools to automate infrastructure, automate work flow, and continuously monitor and measure application performance. The idea of DevOps is to make the development environment the same as the product environment so that it can increase the frequency and speed of deployment and, thus, improve the business.

The following diagram from Edureka shows DevOps phase and tools. We can see it integrate the phases of development—such as plan, code, build, and test—with the operation phases, such as deploy, operate, and monitor. Each phase is empowered by automation tools to improve the speed and quality.



DevOps Phases and Tools

Here is a brief description of each tool included in diagram:

Git	A distributed version-control system used to manage the changes of source codes and documents.
Subversion(SVN)	A centralized version-control system.
JIRA	A bug-tracking and Agile management system developed by Atlassian.
Ant	A Java library and command-line tool mainly to help build Java applications. It can also be used to build non-Java applications.
Maven	Another dependency-management and build-automation tool, primarily used for Java applications. Less flexible than Ant, but easier to use.

Gradle	Another build tool, which was built upon the concepts of Ant and Maven.
Junit	A unit-testing framework for Java applications.
Selenium	An automated testing suite for web applications across different browsers and platforms. It focuses on automating browsers.
Jenkins	An automation server that enables continuous integration (CI) and continuous delivery (CD). It helps automatically build, test, and deploy the software.
Hudson	Jenkins was previously known as Hudson. Oracle gained control of the Hudson project in 2010. In early 2011, the community renamed the code to Jenkins and continued developing the tool, leaving Oracle in control of the original Hudson code. Oracle now owns the Hudson trademark, but has licensed it under the Eclipse Public License (EPL).
Bamboo	Another CI and CD automation tool. Unlike Jenkins, it is a commercial tool by Atlassian.
Docker	A popular container platform used to securely build, share and run applications.
Vagrant	A tool for building and managing virtual machine environments in a single workflow.
Puppet	A configuration-management tool to help development and operations teams manage applications and infrastructure. It is built with Ruby and provides puppet DSL (domain-specific language) to write configuration files.
Chef	A configuration-management tool. It is very similar to Puppet.
Ansible	Currently the most popular configuration-management tool. It runs on many Unix-like systems and can configure both these and Windows. It automates the configuration and deployment of remote nodes. It is written in Python and uses YAML, a very simple script language for configuration files. It is easier to use than the other configuration-management tools. Unlike Puppet, it is agentless and uses a push system.
Saltstack	Another configuration-management tool. It is similar to Ansible.
ELK Stack	A very popular log-management platform. It is the acronym for three open-source projects: Elasticsearch, Logstash, and Kibana.
Splunk	A tool to analyze, search and visualize machine data such as logs.
Nagios	A network-, server-, and log-monitoring system.

While most Agile software-development teams also use automation tools for development—such as automated build, test automation, CI, and CD—DevOps teams often use all of those tools and more, including configuration management, metrics and monitoring schemes, virtualization, and cloud computing. They also address different issues. Agile focuses on the small, iterative, incremental development process, while DevOps focuses on integrating development and operation. They are often used together to improve software quality and business productivity.

Test Yourself

What are the values written in the Agile “Manifesto”?

(Please see the Agile “Manifesto.”)

Test Yourself

What are similarities and differences between XP and Scrum?

Similarities: Both have similar values, promote small iterations, use user stories, and promote agility and light process.

Differences: Most XP practices focus on the technical side of software development, such as refactoring, test-driven, pair programming, and continuous integration. Scrum focuses on the management side and defines the work flow. It can also be used in other industries.

Test Yourself

What are the responsibilities of a scrum master?

To ensure that the team understands and follows Scrum process and practices, and ensure good communication among all members of the team and external stakeholders.

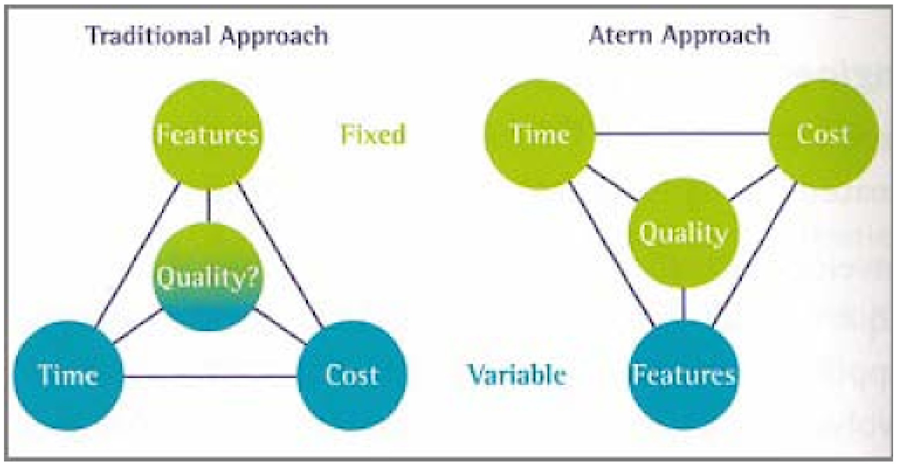
Test Yourself

How does DevOps differ from traditional software-development processes?

DevOps also emphasize the operation (e.g., deployment and configuration) and integrates it with the development so that they can communicate and serve each other better.

Topic 4: Software Quality Assurance, Configuration Management, and Risk Management

We always want to produce high-quality products with all features with less time and cost. However, there are always trade-offs among these factors: cost (e.g., the number of man hours), time (e.g., the number of weeks before delivery), quality (e.g., the number of defects), and functionality (the number of features completed). In the traditional approach (particularly the waterfall model), a contract usually defines all features that need to be completed. The time and cost may vary and be negotiable. The quality may be sacrificed to a certain degree to reduce time and cost. In the alternative approach (Agile), each iteration has a fixed time and cost. The quality cannot be compromised. However, the number of features completed may be negotiable. Quality is one of the most important goals of the Agile process.



Project Factors TradeOff

Software Quality

The question is how to define software quality. What does high quality mean? A number of attributes are usually used to describe high software quality, such as performance, usability, reliability, scalability, and security. Steve McConnell's *Code Complete* (2004) divided quality attributes into two categories: external and internal. External attributes are those parts of a product that face its users, and internal attributes are those that do not.

- External quality characteristics: correctness, usability, efficiency, reliability, integrity, adaptability, accuracy, and robustness
- Internal quality characteristics: maintainability, flexibility, portability, reusability, readability, testability, and understandability

CISQ (Consortium for IT Software Quality) has defined five major, desirable characteristics that a piece of software needs to provide business value: reliability, efficiency, security, maintainability, and size. While size is not really a quality attribute per se, it impacts maintainability. It can be used to measure the complexity, and assess the workload and cost, as well as productivity.

Be aware that these quality attributes are perceptual and subjective, and can be interpreted differently by different people. For example, a system admin may love the command line Terminal, but a general user may think it is very user-unfriendly. Moreover, it is generally hard to achieve all. There are always trade-offs between these quality attributes. Which quality attributes are more important is decided by the project stakeholders and will vary from projects to projects.

Test Yourself

What quality attributes are most important in your project? How do you plan to evaluate it?

To properly define and evaluate the quality of a piece of software, we need to do the following:

- Understand the customer's needs and intentions. The nonfunctional requirements usually define the quality attributes that the customer cares about.
- Anticipate changes and think about how the software can more easily evolve.
- Define quantifiable and testable metrics to measure and evaluate the quality.

Defect Management

While there are many different quality attributes, the more closely a software product meets its specified requirements (both explicit and implicit), and those requirements meet the wants and needs of customers, the higher its quality. Software-defect metrics are often used to measure how close the software product is to its requirements. A software defect is any deviation from requirements. Common defects include missed or misunderstood requirements, design flaws, and coding errors. Software-defect management is an important element of quality assurance.

Software defects can be categorized based on the following:

- Severity—Major, medium, trivial
- Priority—High, medium, low
- Type—Omission, unnecessary inclusion, inconsistency, unclassified inclusion
- Source—In which phase was it injected?

Defects need to be tracked and managed during the whole life cycle. The following metrics are usually used:

- The number of unrepaired defects in each category
- The number of repaired defects in each category
- Defect density (the number of defects per document or per thousand lines of code [KLoC])
- Repair rate (the number of defects fixed per week)

Understand that the cost to repair defects increases the later a defect is discovered. Therefore, the following are important quality goals:

- Remove as many defects as is reasonably possible before the project is completed.
- Remove as many of these defects as early as possible in the development process.

A spreadsheet like the one below may be used to track and manage the defects manually.

Defect Tracking

Name	**Description**	**Discovering Enginr.**	**Responsible Enginr.**	**Date Opened**	**Source**	**Severity**	**Type**	**Status**
Checkout flicker	Checkout screen 4 flickers when old DVDs are checked out by hitting the “Checkout” button.	Kent Bain	Fannie Croft	1/4/04	Integration	Med	GUI	Being worked, begun 2/10/04
Bad fine	Fine not correct for first-run DVDs checked out for 2 weeks, as displayed on screen 7.	Fannie Croft	April Breen	1/4/06	Requirements	High	Math	Not worked yet
---	---	---	---	---	---	---	---	Tested with
---	---	---	---	---	---	---	---	Resolved

---	---	---	---	---	---	---	---	---

Nowadays, there are a lot of tools that can help manage defects or bugs. For example, Bugzilla is an issue-tracking tool built by the team behind Mozilla Firefox since 1998. Jira is a popular Agile software-development tool developed by Atlassian to track issues, bugs, and tasks related to software-development projects. It is highly configurable and has a lot of add-ons available. It is also a great project-management tool. Pivotal Tracker is another Agile project-management tool that supports the bug-tracking feature. GitHub also provides support to track defects.

Test Yourself

Can you provide some examples of software defects?

Other Related Quality Metrics

Besides defect metrics, there are other useful metrics that can help measure product and process quality and should be collected and analyzed throughout the software project.

Product Metrics:

- Mean time to failure (MTTF), to measure reliability
- Customer problems and customer satisfaction, to measure usability and operability.
- Size (e.g., the number of KLoC)
- Complexity

Process Metrics:

- Project effort (the number of person hours)

By defining proper quality metrics, we can measure software quality quantitatively, which also helps us estimate project schedules, track progress, determine project cost, and improve the process. To improve the process, such metric data on multiple projects should be reviewed and analyzed to understand which practices worked and which didn’t.

Quality Assurance Process

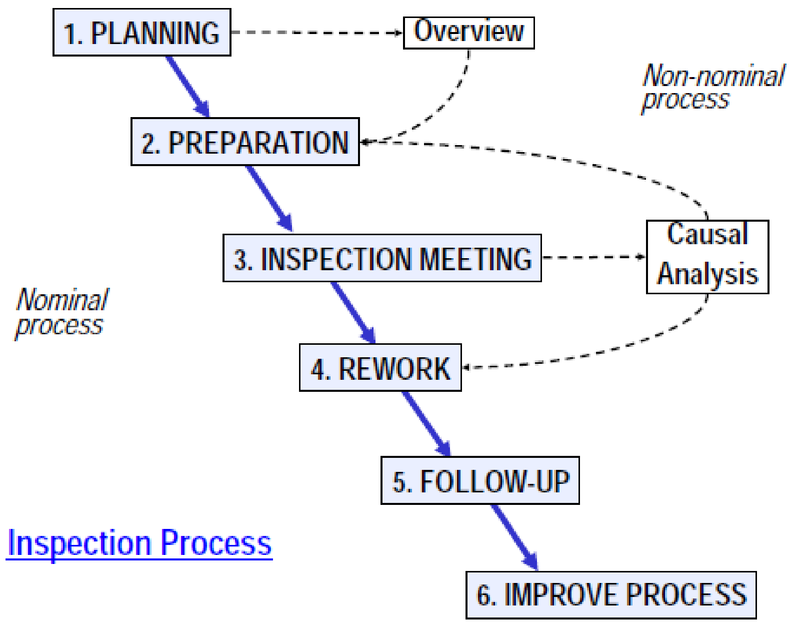
Quality is not something that can be added on. Quality must be built into the software. It is important to focus continuously on quality. To improve the quality and reduce the cost, we want to find and repair defects as early in the development process as possible. In addition, a proper quality assurance process should be defined and followed.

Validation and verification (V&V) are two important means of assuring quality. **Verification** ensures that each artifact is built in accordance with its specifications. It attempts to answer the question, “Are we building the product correctly?” It focuses on the process and is mostly done through **inspections and reviews**. **Validation** checks that each completed artifact satisfies its specifications. It targets the question, “Are we building the right product?” It focuses on the product and is mostly done through **testing**.

Inspections and Reviews

Code review is a very important method of identifying bugs in the code. Besides the source code, documents may also contain defects. Inspection is an approach beyond the code review.

Fagan) inspection is a quality technique that focuses on reviewing the details of a project artifact, which include both documents and code. The purpose of the inspection is to seek defects and assure the correctness of all artifacts. It requires a formal process and can be costly.



Inspection Process (Braude, 2010)

A number of metrics should be collected from the inspection process, such as the following:

- Number of defects discovered, by severity and type
- • Number of defects discovered, by category of stakeholder inspecting the artifact
- Number of defects per page reviewed
- Review rate (number of pages per hour)

Test Yourself

What practices in inspection or review are suitable for your group project, and why?

Testing

No software product can be released without being tested first. There are different types of testing, such as unit testing, modular testing, integration testing, functional testing, performance testing, stress testing, smoke testing, and acceptance testing. The testing may be done by different people, including developers, quality-assurance professionals (QAs), and even third-party teams. Different techniques may be applied to help detect defects. Testing should not just be done at the end of the software-development life cycle. It should be performed throughout the whole life cycle. Test-driven development is a technique promoted by XP that particularly emphasizes the concept of testing early and often. We will discuss more details about testing in the later modules.

Test Yourself

What types of testing will your group use, and who will do the testing?

Test Yourself

Do you know TDD (test-driven development)? If yes, what are the pros and cons? Do you plan to use TDD in your project?

Several documents may be produced in the software quality-assurance process, including the following:

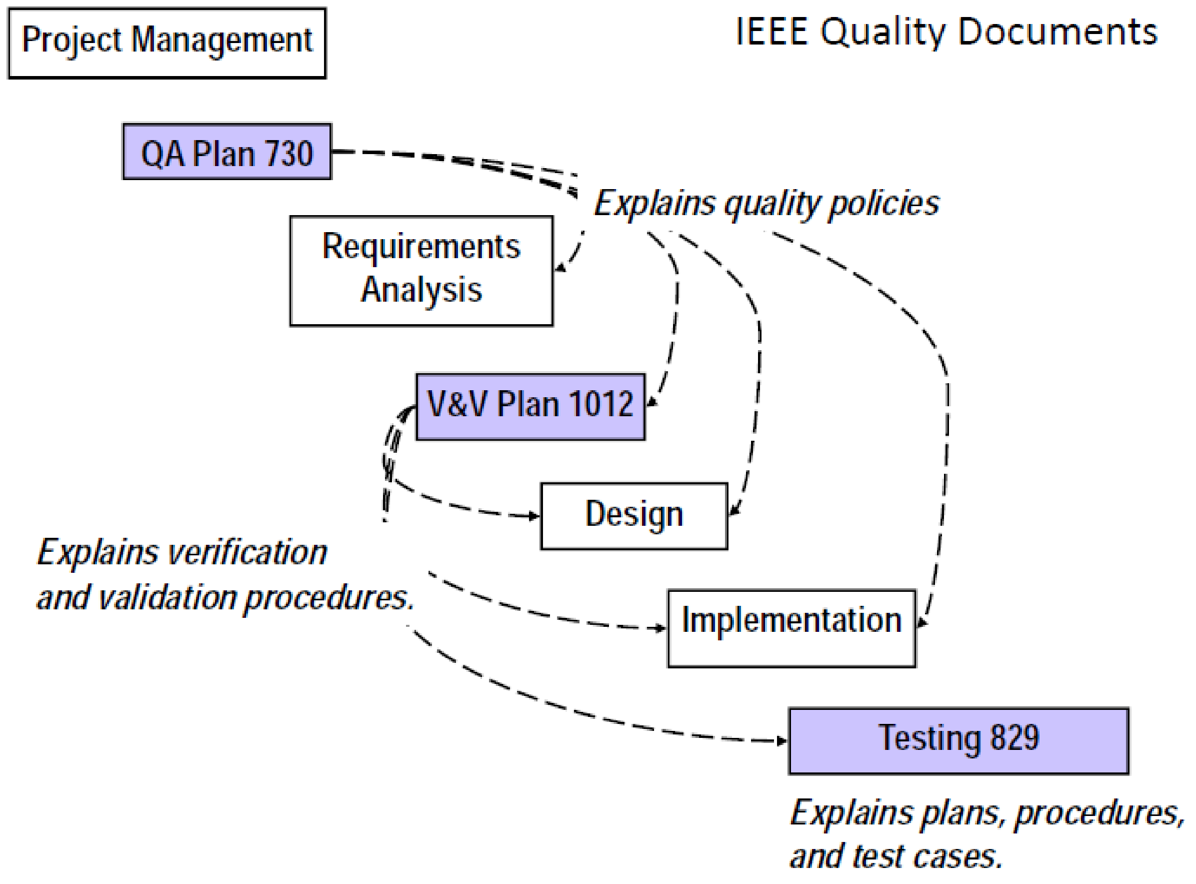
1. SQAP: software quality-assurance plan
2. SVVP: software verification and validation plan
3. STD: software testing document

Understand what inspections and reviews are. The quality-assurance process needs to define the who, when, and how of conducting inspections and reviews in each project. How does this affect configuration management?

For example, the following questions should be answered in the SQAP document:

- Who is responsible?
- What documentation will be generated?
- What standards will be used?
- What metrics will be used?
- What procedures will be used?
- What testing will be performed?
- What techniques will be used?
- How will defects be handled

The following diagram shows the relationships between these IEEE quality documents:


Project Management
IEEE Quality Documents
QA Plan 730
Explains quality policies
Requirements Analysis
V&V Plan 1012
Design
Explains verification and validation procedures.
Implementation
Testing 829
Explains plans, procedures, and test cases.

IEEE Quality Documents (Braude, 2015)

Software Configuration Management

In software engineering, software configuration management (SCM) basically refers to the management of any items that can be configured and are important for the success of the project. Configuration items include source code, property files, documentation, binaries, and other entities in the software life cycle. SCM helps in identifying, tracking, managing, and controlling changes of these configuration items. The goal is to increase productivity with minimal cost.

Changeability is an inherent characteristic of software engineering. All software is subject to constant changes during each phase of its life cycle, such as planning, design, development, and deployment. For example, bugs are constantly discovered and need to be fixed. The source code and the executable program are of different versions. The requirements document is updated. These problems occur frequently and are difficult to fix without SCM. SCM plays an important role in the software life cycle by ensuring that what was designed is the same as what is built and deployed.

Version control is the main part of the traditional SCM. It is particularly useful in managing the change of source code. Sometimes people refer to source-code management or version control alone as configuration management. However, version control focuses on the management of different versions, while SCM means that you can build, deploy, and test any particular version, particularly when it comes to DevOps.

SCM Process

The SCM process can comprise the following:

- Identification of configuration items, such as source-code modules, test case, and requirements specification.
- Baseline management. A baseline is a formally accepted version of a software configuration item. It is designated and fixed at a specific time while conducting the SCM process. It can only be changed through formal change-control procedures. Simply, “baseline” means ready for release. Widely used baselines include functional, developmental, and product baselines.
- Controlling change is a procedural method that ensures quality and consistency when changes are made in the configuration object. In this step, the change request is submitted to the software-configuration manager, who controls ad hoc change to build a stable software-development environment. Changes are committed to the repository. The request will be checked based on the technical merit, possible side effects, and overall impact on other configuration objects. This function involves managing changes and making configuration items available during the software life cycle.
- Configuration status accounting. Track what each release version has and the changes that led to this version. Keep a record of all the changes made to the previous baseline to reach a new baseline.
- Auditing and reproducibility. Auditors check that defined processes are being followed and the SCM goals are satisfied. This task is to verify that all of the software product satisfies the baseline needs.

In DevOps, two new concepts, infrastructure as code (IaC) and configuration as code (CaC), are used to manage not only the changes of the source code, but also the infrastructure and configuration, making the building, deploying, and configuring of each new version much easier.

Traditionally, infrastructure provision and configuration between environments are managed manually with each release. System admins need to manually run the script (maybe modified) for each release on each node. IaC and CaC treat infrastructure and configuration as code and thus as subject to version control so that they can be traced, rolled back, and deployed automatically, using configuration-management tools such as Ansible, Puppet, Chef, and SaltStack—introduced in the previous DevOps section.

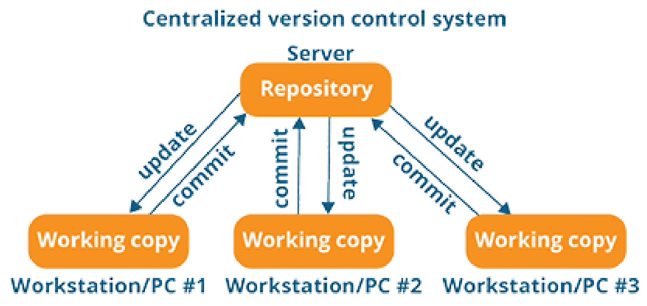
Version-Control System (VCS)

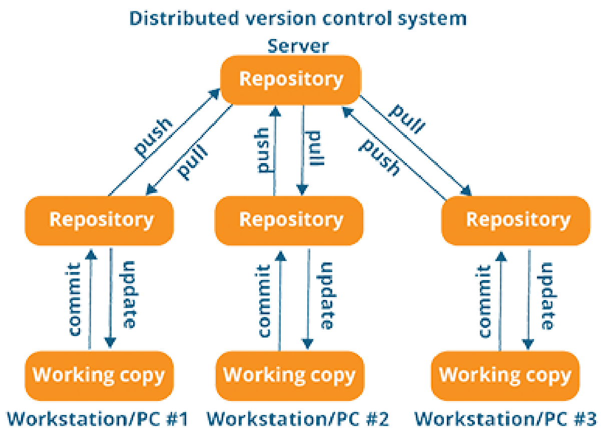
VCS is a way to manage changes to documents, source code, and other information. It can track any changes and roll back to any version. It also supports parallel changes and resolves conflicts among developers’ codes.

There are two types of VCS:

- **Distributed**—Each user clones the project repository on his or her local machine and makes changes to the local copy, committing locally first (thus quite fast). Once the user is confident in all the changes that have been made, he or she can then push it to the server to make the changes available for everyone else to pull. Git is currently the most popular distributed VCS. GitHub is the largest web-based hosting service using Git. Bitbucket is another popular host that supports Git. It also supports Mercurial, another distributed VCS.
- **Centralized**—Every update is made directly on a single remote server, where developers will clone and check out the source code and then push back their revision. CVS (Concurrent Versions System) and SVN (Subversion) are two popular centralized VCSs.

The following figure shows the difference between distributed and centralized VCSs.





Distributed and Centralized VCS

Here we provide brief explanations of commonly used terms in VCS:

- **Repository**—Stores a set of files, as well as the history of changes made to those files, often on a server.
- **Initialize**—To create a new, empty repository.
- **Baseline**—An approved revision of a document or source file from which subsequent changes can be made. Sometimes, it refers to a unique line of development that is ready for release. In this case, it may also be called a mainline or master.
- **Branch**—A set of files under version control may be duplicated at a time point, and then the copy will develop independently from the original one from that time forward. Branches are used when activities could destabilize the main line or considerably slow down development. Usually, an active development branch is used to allow teams to develop and release changes more frequently, with reduced risk of breaking the main line. Furthermore, additional branches may be forked from the development branch, such as a private personal branch, test branch, or integration branch. These branches will be merged into the active development branch and eventually into the master. A set of rules and guidelines are needed for branch creation and management.
- **Stream**—A container for branched files that has a known relationship to other such containers. Streams form a hierarchy; each stream can inherit various properties (e.g., versions, namespace, work-flow rules, subscribers) from its parent stream.
- **Clone**—Creating a repository containing the revisions from another repository. This is equivalent to pushing or pulling into an empty (newly initialized) repository. In a distributed VCS, such as Git, a local repository is first created by cloning the remote repository.
- **Change (or diff, delta)**—Represents a specific modification to a file under version control.
- **Checkout (or co)**—To create a local working copy from the repository. A user may specify a revision or obtain the latest.
- **Commit (or check in, ci)**—To write or merge the changes made in the working copy back to the repository.
- **Head (or tip)**—The most recent commit, either to the trunk or a branch. The trunk and each branch have their own head, though “HEAD” is sometimes loosely used to refer to the master.
- **Conflict**—Occurs when different parties make changes to the same file, and the system is unable to reconcile the changes automatically. A user must resolve the conflict by combining the changes or by rejecting one change in favor of the other.
- **Merge (or integration)**—An operation in which two sets of changes are applied to a file or set of files.
- **Pull, push**—Copy revisions from one repository into another. Pull is initiated by the receiving repository, while push is initiated by the source. “Fetch” is sometimes used as a synonym for pull, or to mean a pull followed by an update.
- **Forward integration**—The process of merging changes made in the main trunk into a development (feature or team) branch.
- **Reverse integration**—The process of merging different team branches into the main trunk of the versioning system.

Nowadays, every software-development team uses some type of VCS, and web services such as GitHub make it easier than ever. Simply having a VCS in place is not enough. A proper branching strategy and commit policy are the keys to using a VCS effectively. Read this article to get some insights on how to create an effective branching strategy: Successful Branching Strategies and Commit Policies for SVN/GIT.

Test Yourself

Have you used a VCS tool, such as Git or SVN? If yes, what was the biggest issue you had when using it?

Test Yourself

What kind of branching strategy and commit policy do you propose for your group project?

Risk Management

Risks are uncertain factors that may impact the project, the product, or the business negatively. Uncertainty is inherent in software engineering. Therefore, it is very important to identify these risks and minimize their effects; that is risk management.

Risks can be categorized based on what is affected:

- Project risks threaten the project plan. Examples include potential budgetary, schedule, personnel, resource, customer, or requirement problems; project complexity, size, and structure; etc.
- Technical risks threaten the quality and timeliness of software products. Examples include specification ambiguity, technical uncertainty, technical obsolescence, leading-edge technology, and potential design, implementation, interfacing, verification, and maintenance problems.
- Business risks threaten the viability of software to be built and include market risk, strategic risk, sell risk, management risk, and budget risk.

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.

Risk	Affects	Description
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Project competition	Business	A competitive product is marketed before the system is completed.

Possible Risks (Sommerville, 2015)

It may be categorized by the root cause of the risk. Sommerville also gave some examples:

Risk Type	Possible Risks
Estimation	• The time required to develop the software is underestimated. (12) • The rate of defect repair is underestimated. (13) • The size of the software is underestimated. (14)
Organizational	• The organization is restructured so that different management are responsible for the project. (6) • Organizational financial problems force reductions in the project budget. (7)
People	• It is impossible to recruit staff with the skills required. (3) • Key staff are ill and unavailable at critical times. (4) • Required training for staff is not available. (5)
Requirements	• Changes to requirements that necessitate major design rework are proposed. (10) • Customers fail to understand the impact of requirement changes. (11)
Technology	• The database used in the system cannot process as many transactions per second as expected. (1) • Reusable software components contain defects that mean they cannot be reused as planned. (2)
Tools	• The code-generated by software code generation tools is inefficient. (8) • Software tools cannot work together in an integrated way. (9)

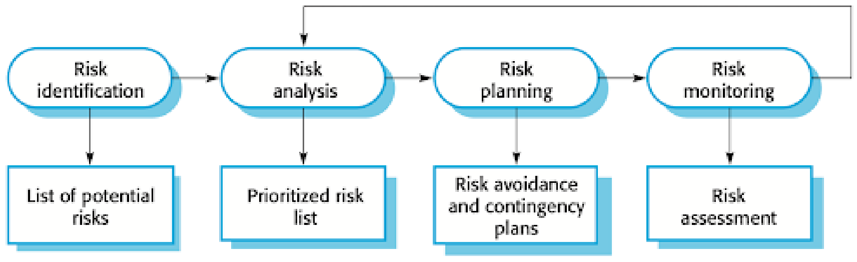
Risk Examples (Sommerville, 2015)

They can also be categorized as either generic risk or project/product–specific risks.

- Generic risks include the following:
    - Product size
    - Business impact
    - Customer characteristics
    - Process definition

  - Development environment
  - Technology to be built
  - Staff size and experience.

Risk-Management Process



Risk-Management Process (Sommerville, 2015)

The goal of risk management is either to identify and retire risks or fail as early as possible. The risk-management process involves four tasks:

- Identify risks
  - We may use a risk checklist. We want to evaluate all worst-case scenarios. This task may last for approximately the first third of the project.
- Analyze risks
  - A risk has two characteristics: uncertainty and loss. To quantify risks, we need to evaluate the level of uncertainty and degree of loss.
  - For each risk, we need to analyze its probability of occurrence, its potential impact, and the cost to retire it. The impact may be performance degradation, cost overrun, support difficulty, and/or schedule slippage.
  - Then we can perform a quantitative comparison of the above factors and prioritize them. For example, risks with high likelihood, severe impact, and low cost to repair should have higher priority.
- Mitigate risks
  - Risks can be retired through either conquering or avoiding them. For example, the risks of using a new technology that no one on the team is familiar with can be avoided by choosing a different technology, or they can be conquered through a dedicated training session. A detailed plan is needed for each identified risk. You may simply accept some risks without any changes. In this case, project-management approval is mandatory. The risks may also be transferred to other stakeholders.
- Continuously monitor risks:
  - Assess each identified risk regularly to decide whether or not it is becoming less or more probable, and whether its effects have changed. Pay particular attention to whether the impact of the risk is increasing.

This process should be repeated in each iteration to keep track of risks, particularly high risks. The following table provides a template for risk management. You can download the template here: Google Docs link. The instructor will give you access to this link.

Top Risk Lists

What are commonly identified risks? Here we give several lists identified by researchers in software engineering fields.

Boehm lists the top 10 risk items in his paper early in 1991, as shown below.

Figure 3. A Top Ten List of Software Risk Items

RISK ITEM	RISK MANAGEMENT TECHNIQUES
1. Personnel shortfall	Staffing with top talent; job matching; team building; morale building; cross-training; prescheduling key people
2. Unrealistic schedules and budgets	Detailed multisource cost and schedule estimation; design to cost; incremental development; software reuse; requirements scrubbing
3. Developing the wrong software functions	Organization analysis; mission analysis; ops-concept formulation; user surveys; prototyping; eary users' manuals
4. Developing the wrong user interface	Prototyping; scenarios; task analysis
5. Gold plating	Requirements scrubbing; prototyping; cost-benefit analysis; design to cost
6. Continuing stream of requirements changes	High change threshold; information hiding; incremental development (defer changes to later increments)
7. Shortfalls in externally performed tasks	Benchmarking; inspections; reference checking; compatibility analysis
8. Shortfalls in externally performed tasks	Reference checking; pre-award audits; award-fee conacts; competitive design or prototyping; teambuilding
9 Real-time performance shortfalls	Simulation; benchmarking; modeling; prototyping; instrumentation; tuning
10. Straining computer science capabilities	Technical analysis; cost-benefit analysis; prototyping; reference checking
Top Ten Risk Items (Boehm, 1991)	

Tharwon Arnuphaptrairong (2011) compiled a number of top-10 lists reviewed in the literature. The following table shows the risk items and their frequency in the 8 top-10 lists he compiled.

NO.	RISK ITEM	FREQ.
1	Misunderstanding of requirements (R)	5
1	Lack of top management’s commitment and support (Org)	5

NO.	RISK ITEM	FREQ.
3	Lack of adequate user involvement (U)	4
4	Failure to gain user commitment (U)	3
5	Failure to manage end-user expectation (U)	3
6	Changes to requirements (R)	3
7	Lack of an effective project-management methodology (P&C)	3

The following table lists the top 10 risks in students' projects, as identified by Laurie Williams and listed in her 2004 paper, "Risk Management."

Risk Item	Risk Management Technique
Overriding other people's work, not having the latest versions of code.	Use a configuration management tool effectively.
Lack of exposure to and/or experience with technologies	Take time to learn tools and technologies, seek help from teaching staff.
Being overwhelmed by work in other classes.	Have a project management plan with deadlines and ownership, update the project management plan frequently.
Requirements understanding	Meet with, e-mail, or phone customer.
Lack of communication	Set up a group Web page, group e-mail accounts, trade instant messaging IDs, meet regularly.
Project organization	Assign each team member a role, break down work in project management plan.
Loss of a team member	Assure files are uploaded and integrated consistently, use knowledge management strategies such as pair programming to understand each other's work.
Difficulty integrating work	Increase communication, integrate often.
Planning taking up too much time, not enough time to work on product.	Don't get more detailed than necessary with the planning.
Top 10 Student Projects Risk Items (Williams 2004)	

Software Risk Factor Rating (Asif, Ahmed, & Hannan, 2014) collected 20 risk factors from the literature review and then conducted an exploratory survey to prioritize them. A sample of 200 respondents was selected, out of which 100 were university students and 100 were the IT professionals of a well-reputed software house.

Rating	Software Risk Factor	Frequency In Percentage
1	Improper feasibility report	90%
2	Higher management's decisions	88%
3	Understanding problems of customers	85%
4	Understanding problem of developers	82%
5	Improper planning	80%
6	Improper scope definition	78%
7	Lack of experience of project manager	76%
8	Government factors	75%
9	Implementation	72%
10	Cultural diversity	70%
11	Lack of motivation	68%
12	Personnel hiring	67%
13	Unrealistic deadline	60%
14	Inappropriate design	55%
15	Improper budget	52%
16	Inappropriate technology	49%
17	Lack of resources	44%
18	Size of the project	40%
19	Improper marketing techniques	35%
20	Market demand obsolete	30%
(Asif, Ahmed, & Hannan, 2014)		

Please assign two numeric values, from 0 to 5 (5: most likely), for each of the following risk factors, based on your own opinion:

Risk Factors

Please provide your rating of the following risk factors (5: most likely, 1: least likely) in two different scanior. One is the likelihood in a real world software project in general. The other is the likelihood in your group project this semester.

aduffy6144@gmail.com Switch account

* Required

Email *

Your email

Your name

Your answer

Test Yourself

Briefly describe the difference between reliability and robustness of a software system.

Reliability is the ability to work as intended and be resilient upon encountering errors. Robustness is the ability to cope with errors or attacks.

Test Yourself

Provide an example in which improving maintainability may affect efficiency.

For example, applying design patterns will add additional classes, method calls, interfaces, which can worsen efficiency.

Test Yourself

What is a defect? Can you provide a product metric related to defects? Can you propose a process metric related to defects?

A defect is a derivation from the explicit or implicit requirements. The number of defects per kilo lines of code (KLoC) is a product metric. The repair rate (e.g., the number of defects solved per week) is a process metric.

Test Yourself

What are the commonalities and differences between Git and SVN?

They are both version-control systems. SVN is centralized and Git is distributed.

Test Yourself

Which types of risks identify potential design, implementation, verification, and maintenance problems?

Technical risks

Test Yourself

What are two main characteristics of risks?

Loss and probability

Conclusion

In this module, we introduced the basic terminology and concepts in software engineering. We gained a basic understanding of different activities in the software-development life cycle and several process models. We also provided an overview of how to perform quality assurance, risk management, and configuration management.

Bibliography

- Brooks, F. P., Jr. (1975). *The mythical man-month.* Addison-Wesley.
- Brooks, Fred P. Jr. (1987). No Silver Bullet. IEEE Computer.
- Braude, E. J., & Bernstein, M. E. (2010). *Software engineering: Modern approaches* (2nd ed.). Wiley.
- IEEE. (1996). *IEEE guide for developing software life cycle processes*. IEEE Std 1074.1-1995.
- Royce, W. (1970). *Managing the development of large software systems*. ICSE '87: Proceedings of the 9th international conference on Software Engineering,
- Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5), 61–72. doi: 10.1109/2.59
- Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified software development process*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.
- Scott, K. (2001). Unified process explained. Addison-Wesley Professional. Retrieved from http://www.informit.com/articles/article.aspx?p=24671&seqNum=3
- Beck, K. (1999). Extreme programming explained: Embrace change. Addison-Wesley Professional.
- Beck, K., & Andres C. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Addison-Wesley Professional.
- McConnell, S. (2004). Code complete: *A practical handbook of software construction* (2nd ed.). Microsoft Press.
- Boehm, B. (1991). Software risk management: Principles and practices. *IEEE Softw.*, 8(1), 32–41. Retrieved from http://dx.doi.org/10.1109/52.62930

- Addison, T. (2003). E-commerce project development risks: Evidence from a Delphi survey. *International Journal of Information Management, 23(1), 25–40*.
- Asif, M., Ahmed, J., & Hannan, A. (2014). Software risk factors: A survey and software risk mitigation intelligent decision network using rule based technique. *Proceedings of the International MultiConference of Engineers and Computer Scientists 2014 Vol. I, IMECS 2014, March 12–14, 2014, Hong Kong*.
- Arnuphaptrairong, T. (2011). Top ten lists of software project risks: Evidence from the literature survey. *Proceedings of the International MultiConference of Engineers and Computer Scientists 2011 Vol. 1, IMECS 2011, March 16–18, 2011, Hong Kong.*
- Scientists 2011 Vol 1, IMECS 2011, March 16-18, 2011, Hong Kong.
- Shrivastava, N. K. (2015). Scaling Agile. *Paper presented at PMI® Global Congress 2015—North America, Orlando, FL. Newtown Square, PA: Project Management Institute*.
- Sommerville, I. (2015). *Software engineering* (10th ed.). Pearson.

Boston University Metropolitan College