Lecture 1-3:

- ➢ Configuration data helps set up the programs
- ➢ System documentation helps understand structure of the system
- ➢ User documentation explains how to use the system

What is Software Engineering?

Disciplined, consistent, and systematic effort to construct (design + build) and maintain good quality software in timely and cost-effective manner

Engineering software by remaining concerned about all aspects of software production.

Aspects of Software Production?

1. Size of the software system to be built
2. Complexity of the software system to be built
3. Need and involvement of teams
4. Technical process of developing software
5. Activities such as management of project and teams
6. Development of tools, theories, methods to support production of software

Software Engineers Job?

Adopt a systematic and organized approach, effectively, to produce high quality software.

May have to use Ad hoc approaches to develop software

- ✓ (We can use an ad-hoc approach for small problems that have changing requirements and for which we need quick solutions. So, to reduce the time to production, we start coding without analysis and a plan.)

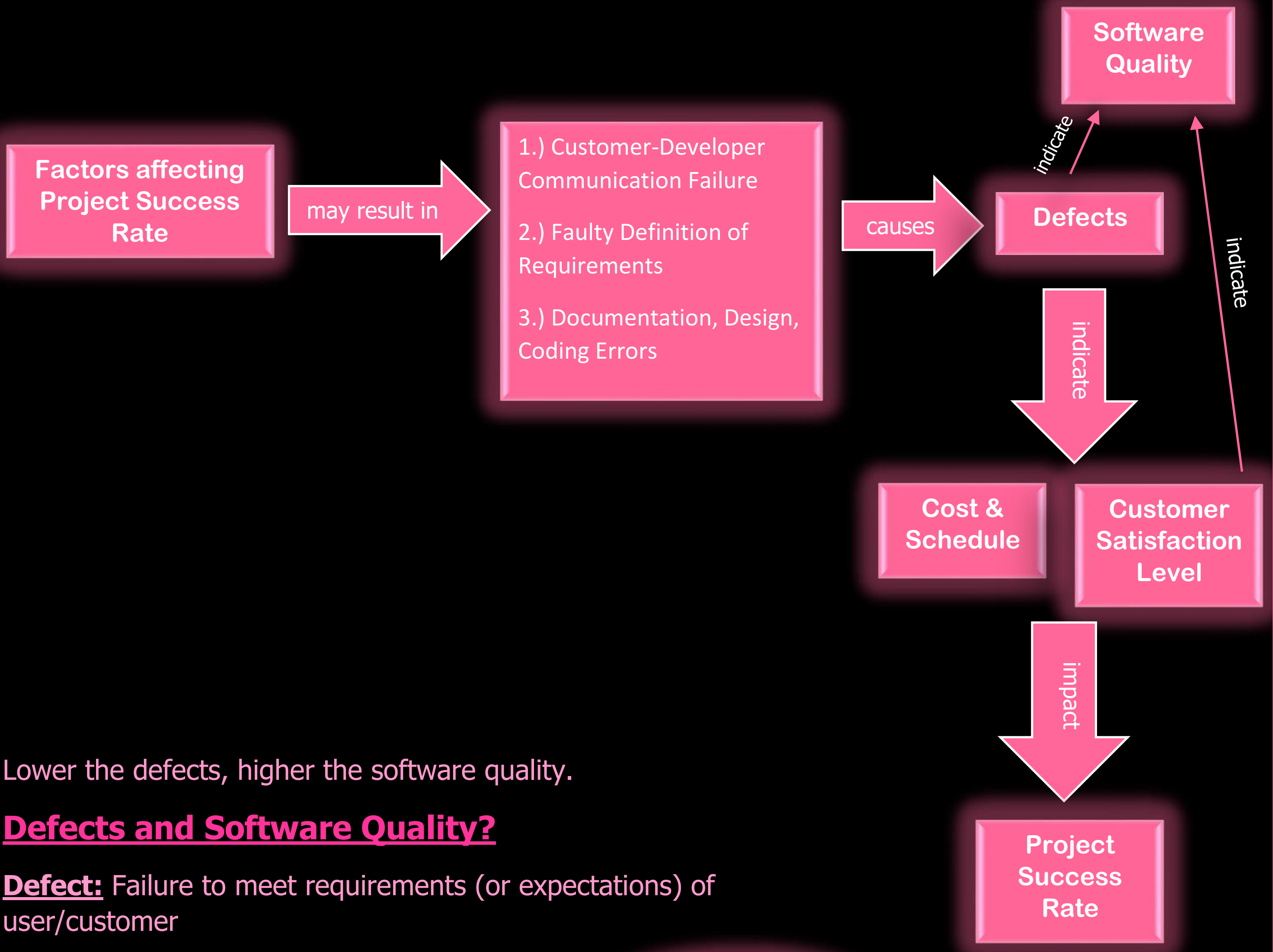
Software Lifecycle?

Phases:

1. Requirements analysis and definition
2. System (architecture) design
3. Program (detailed/procedural) design
4. Writing programs (coding/implementation)
5. Testing: unit, integration, system
6. System delivery (deployment)
7. Maintenance

Factors Affecting Project Success Rate?

1. Project Size and Complexity
2. Customer Satisfaction Level
3. Development and Budget Process
4. Skills of Developers and Managers



Factors affecting Project Success Rate

may result in

1.) Customer-Developer Communication Failure

2.) Faulty Definition of Requirements

3.) Documentation, Design, Coding Errors

causes

Defects

indicate

Software Quality

indicate

Cost & Schedule

Customer Satisfaction Level

indicate

impact

Project Success Rate

Lower the defects, higher the software quality.

Defects and Software Quality?

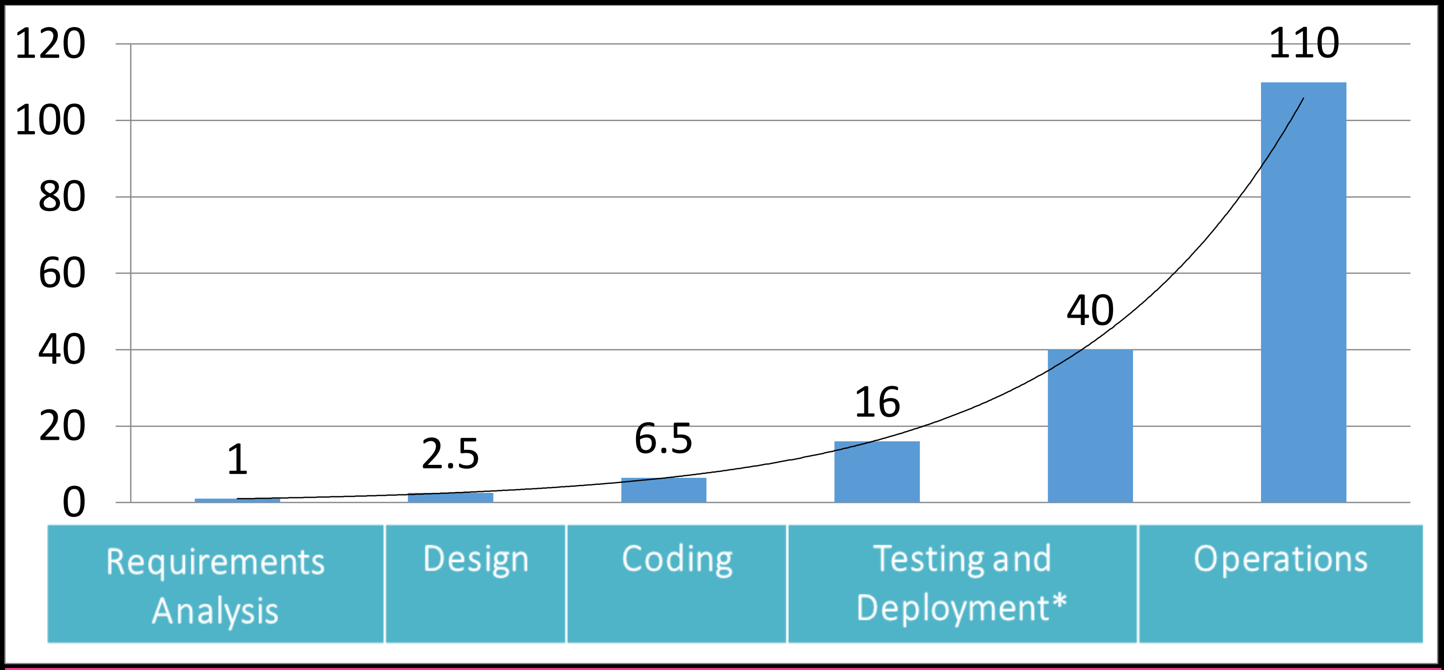
Defect: Failure to meet requirements (or expectations) of user/customer

Software Quality:

1. Fitness for Use
2. Conformance to Requirements
3. Freedom from Deficiencies
4. Lack of Defects

Removal of defects, increases quality, incurs cost.

Cost Associated with Defect Correction:



Defects prevention can reduce costs.

Software are Malleable and Human-Intensive

- ✓ (if a task or organization is people-intensive, it needs a lot of people to do the work involved, especially compared to the profit made: Success in a people-intensive business comes from hiring the right people)

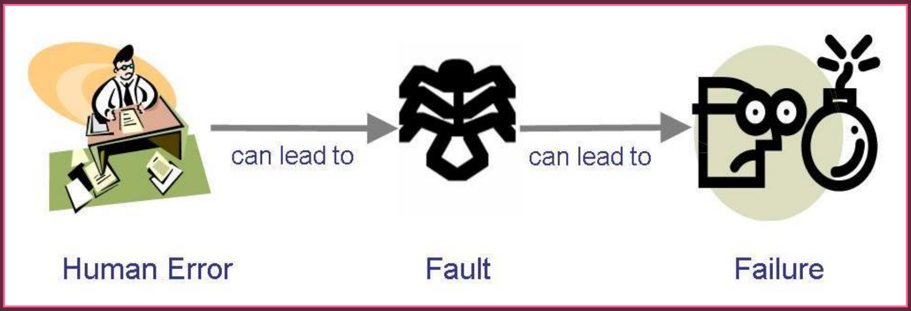
Terminology for Describing Bugs:

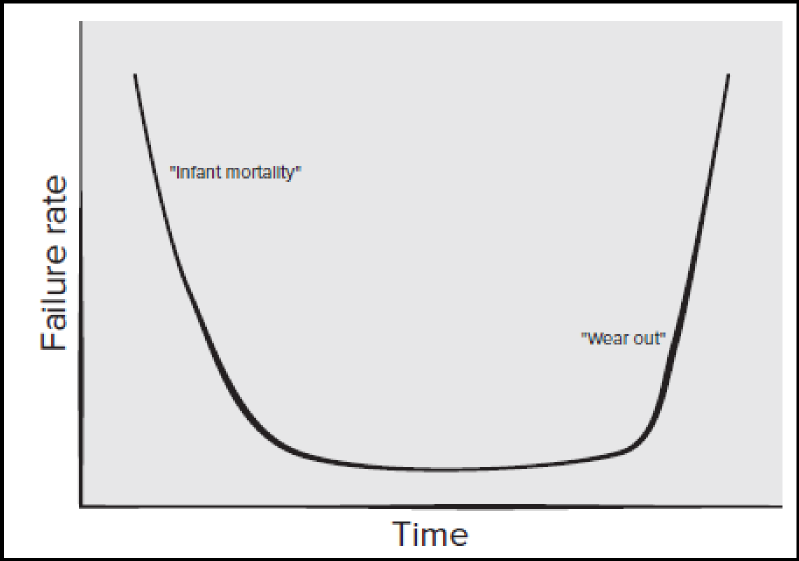
1. **Fault:**
   A fault refers to a flaw or defect in the software code or system design. It's something inherently wrong within the system that can potentially lead to errors. Faults can arise due to various reasons such as coding mistakes, logic errors, or flaws in the system architecture. When a human makes a mistake during software development, such as writing incorrect code or designing a feature improperly, it introduces a fault into the system.

2. **Error:**
   An error occurs when a human, typically a developer, makes a mistake during the execution of some software activity. This mistake could be in the form of writing incorrect code, misunderstanding requirements, or making a wrong assumption during the development process. Errors are manifestations of faults, meaning they result from the existence of faults within the system. For example, if a programmer mistakenly assigns a wrong value to a variable, it's considered an error.

3. **Failure:**
   A failure happens when the system behaves in a way that deviates from its specified or expected behavior. This departure from expected behavior could occur due to errors introduced during the development process or due to the presence of faults in the system. Failures are observable manifestations of errors in the system. For instance, if a software application crashes or produces incorrect output while running, it signifies a failure.

To summarize, faults are inherent defects in the system, errors are mistakes made by humans during development, and failures are deviations from the expected behavior of the system, often resulting from faults or errors.





Failure Rate for Hardware Devices

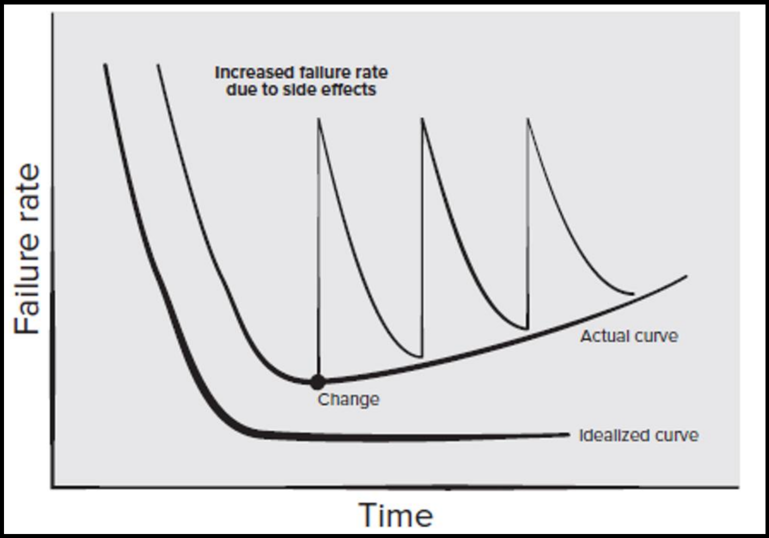
Lehman's Laws

"Continuing Change" — A system must be continually adapted or it becomes progressively less satisfactory. It happens so until it becomes economical to replace it by a new or a restructured version

"Increasing Complexity/Entropy" —Complexity/entropy of a system increases with time, unless work is done to maintain or reduce it

"Continuing Growth" — the functional content of a system must be continually increased to maintain user satisfaction over its lifetime

"Declining Quality" — the quality of a system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes



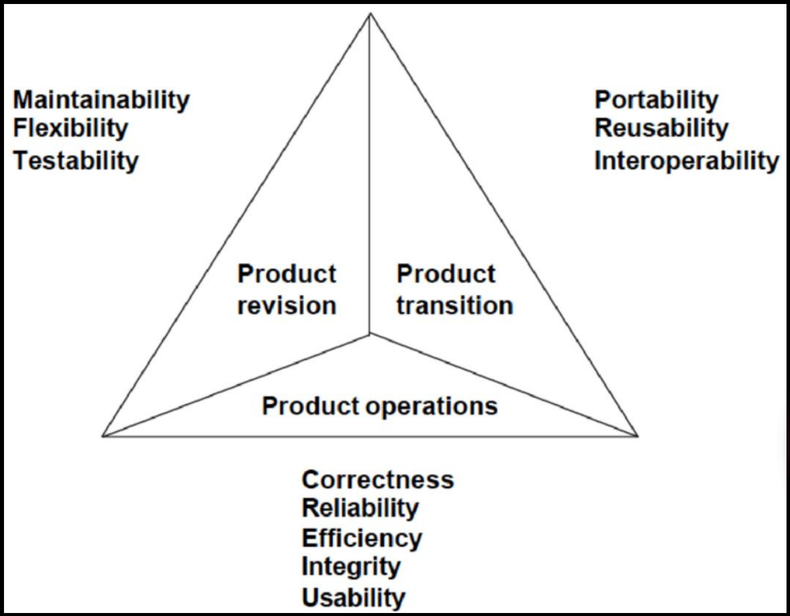
Failure Rate for Software Devices

Costs in Software Engineering:

1. Cost to change software
2. Cost to overcome a mistake (defect)
3. Cost of developing software
4. Operational cost
5. Deployment cost

What is a Good Software Product?

1. Users judge external characteristics (e.g., correct functionality, number of failures, type of failures)
2. Designers and maintainers judge internal characteristics (e.g., ease of modification)
3. Thus different stakeholders may have different criteria



McCall's Quality Model

Software Engineering Challenges:

1. Acceptable Quality

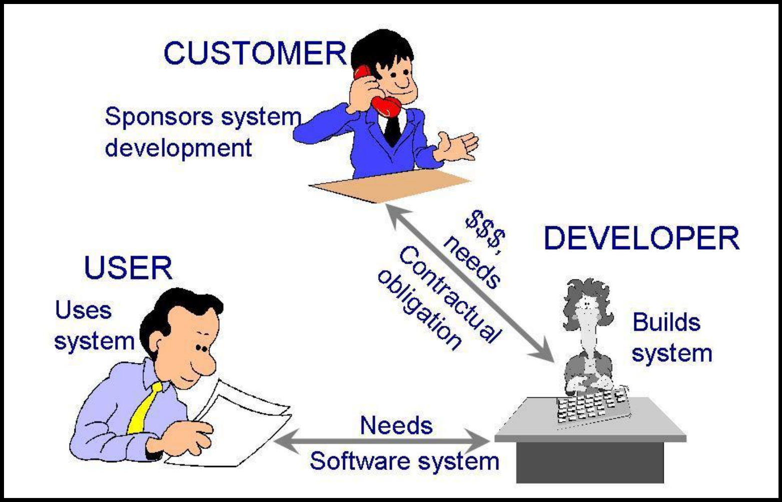
- Usability
- Security
- Reliability
- Performance

2. Cost Effectiveness

- Engineering and operational feasibility
- Limited development budget

3. Timely Completion

- Limited time



Development Team:

Requirements analysts: work with the customers to identify and document the requirements

Designers: generate a system-level description of what the system is supposed to do

Programmers: write lines of code to implement the design

Testers: catch faults

Trainers: show users how to use the system

Maintenance team: fix faults that show up later

Librarians: prepare and store documents such as software requirements

Configuration management team: maintain correspondence among various artefacts

Software Application Domains:

Broad categories of software

- **System software:** programs written to service other programs e.g. compiler, editor, operating system components
- **Application software:** stand-alone programs to solve a specific business need e.g. Information systems like leave management system, point of sales system
- **Engineering/Scientific software**: programs that can simulate things like astronomy, volcanology, molecular biology etc.
- **Embedded systems:** resides within a product or system and performs special functions for user as well as for itself e.g. system to measure temperature in a control shed, keypad control in a microwave oven
- **Product-line software:** software for a group of related products marketed under a single brand name e.g. MS Word, PowerPoint, Excel etc.
- **Web/Mobile applications:** browser based and mobile device based apps
- **Artificial intelligence software:** makes use of non-numerical algorithms to solve complex problems e.g. robotics software, expert systems to diagnose a disease, forecasting system
- **Legacy Software:** old system, difficult to change further

Software Engineering Principles:

Some principles are:

1. **Rigor and Formality**
    - Precision and exactness
    - Assessment of engineering activity and the results
2. **Separation of Concerns**
    - Flow of data and flow of control
    - Correctness and efficiency
    - Parts of system
3. **Modularity**
4. **Anticipation of Change**

Lecture 4:

Process:

- Is a series of steps involving activities, constraints, and resources to produce an intended output
- Involves a set of tools and techniques

Is a process useful?

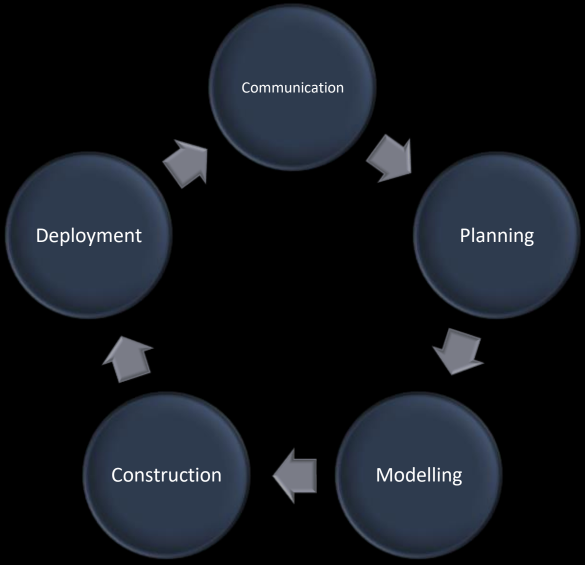
- Impose consistency and structure on a set of activities
- Guide us to understand, control, examine, and improve the activities
- Enable us to capture our experiences and pass them along

Framework Activities:

1. Communication
    - Customer, other stakeholders
2. Planning
    - Roadmap, project plan
3. Modeling
    - Understanding requirements, provide design
4. Construction
    - Code generation, testing
5. Deployment
    - Delivery to customer, feedback and evaluation

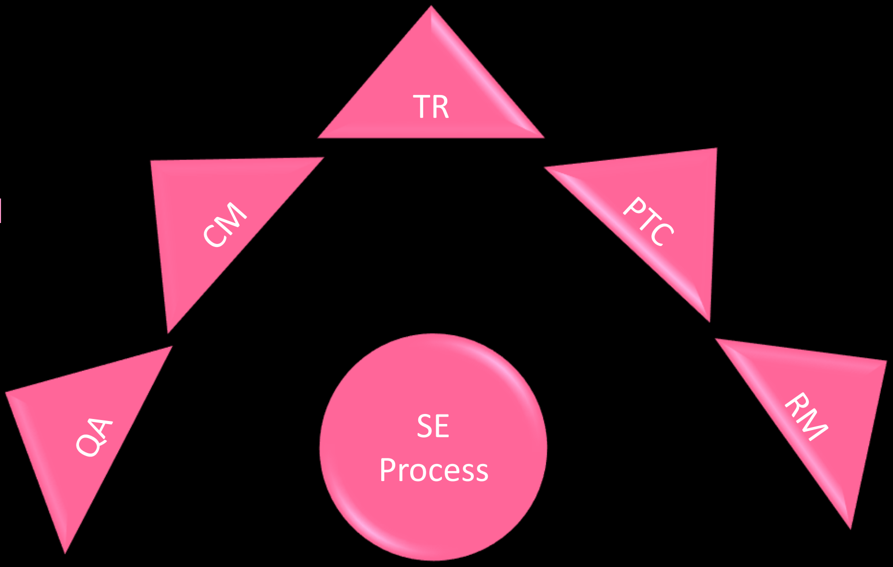
Software Engineering Practice:

- Understand the problem (communication, analysis)
- Plan a solution (modeling and design)
- Carry out the plan (code generation)
- Examine the results for accuracy (testing and QA)



Umbrella Activities:

- Quality Assurance
- Configuration Management
- Technical Reviews
- Project Tracking and Control
- Risk Management



What is a Process Model and why is it needed?

Description of a process, evolved overtime, in a certain format

1. To form a common understanding
2. To find inconsistencies, redundancies, omissions
3. To find and evaluate appropriate activities for reaching process goals
4. To tailor a general process for a particular situation in which it will be used

Prescriptive process models?

Prescriptive process models define a predefined set of process elements and a predictable process work flow. Prescriptive process models strive for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress.

We call them "prescriptive" because they prescribe a set of process elements— framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a work flow)—that is, the manner in which the process elements are interrelated to one another.

The Waterfall Method

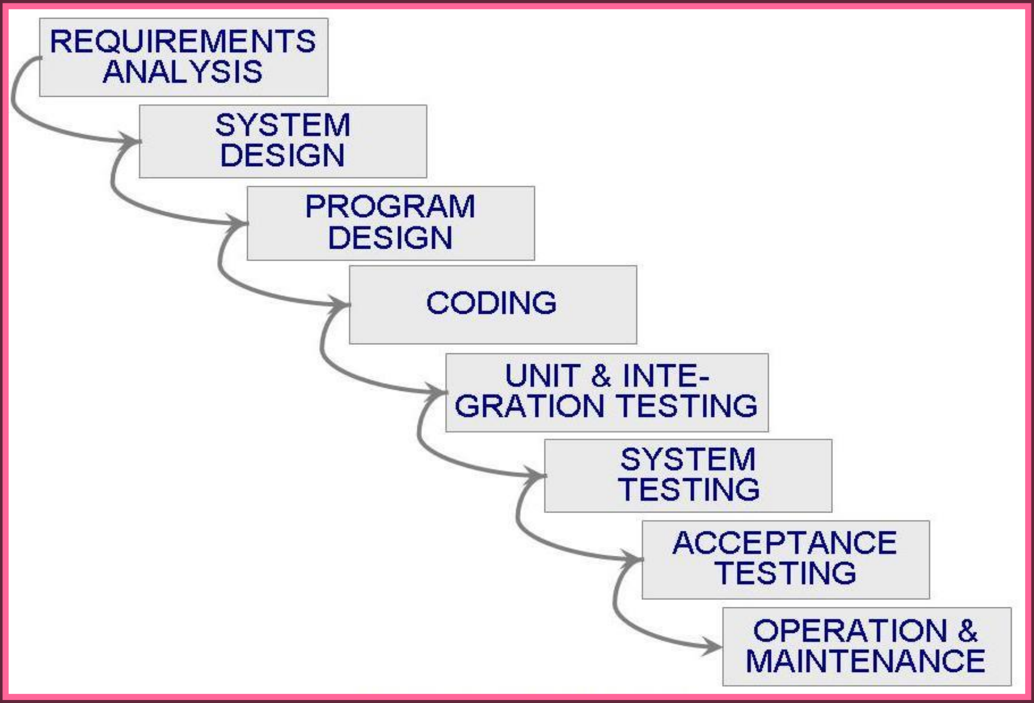
- One of the first process development models proposed
- Works for well understood problems with minimal or no changes in the requirements
- Simple and easy to explain to customers
- It presents
  - a very high-level view of the development process
  - sequence of process activities
- Each major phase is marked by milestones and deliverables (artefacts)
- Provides no guidance how to handle changes to products and activities during development (assumes requirements can be frozen)
- Views software development as manufacturing process rather than as building process

- There are no iterative activities that lead to building a final product
- Long wait before a final product
- Generates lots of documentation
- Considered suitable for large projects

The waterfall model, sometimes called the linear sequential model, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.

Problems?

1. Real projects rarely follow the sequential work flow that the model proposes.
2. It is often difficult for the customer to state all requirements explicitly at the beginning of most projects.
3. The customer must have patience because a working version of the program(s) will not be available until late in the project time span.
4. Major blunders may not be detected until the working program is reviewed.

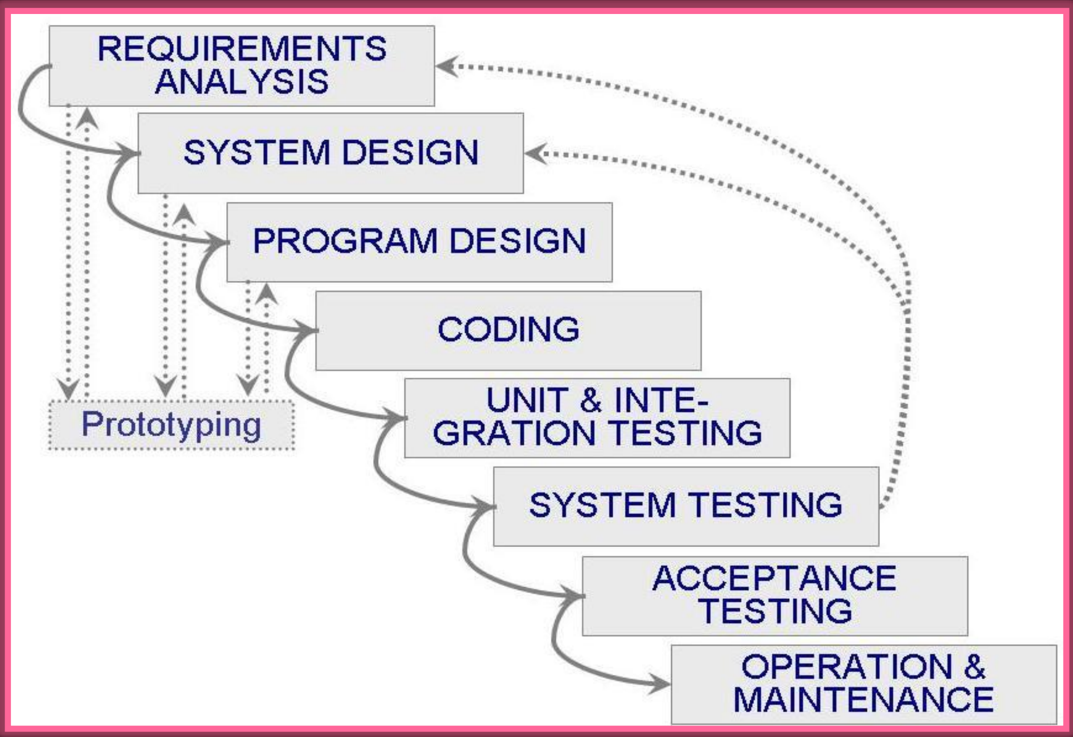


The Waterfall Method with Prototyping

A prototype is a partially developed product

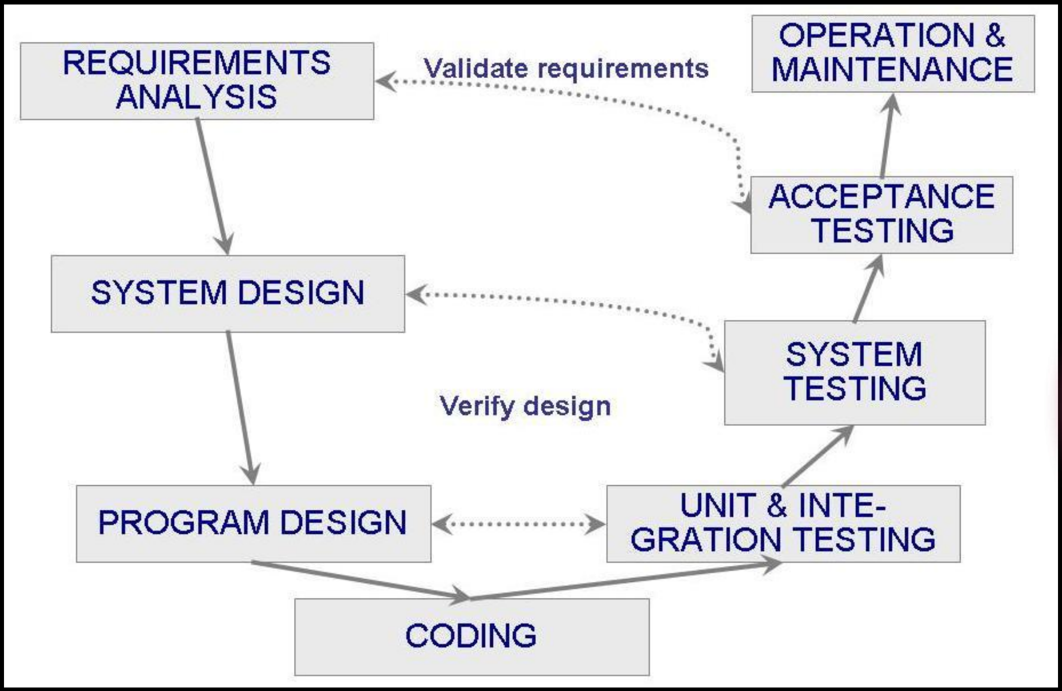
Prototyping helps:

1. developers assess alternative design strategies (design prototype)
2. users understand what the system will be like (user interface prototype)



The V Model

- A variant of the waterfall model
- Uses unit testing to verify procedural design
- Uses integration testing to verify architectural (system) design
- Uses acceptance testing to validate the requirements
- If problems are found during verification and validation, the left side of the V can be re-executed before testing on the right side is re-enacted



Verification: Each function works correctly

Validation: All requirements have been implemented and each functionality can be traced back to a particular requirement

Prototyping Process Model

- Allows repeated investigation of the requirements or design
- Reduces risk and uncertainty in the development

Usage:

Often, a customer defines a set of general objectives for software but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

The prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy

Methodology:

The prototyping paradigm begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).

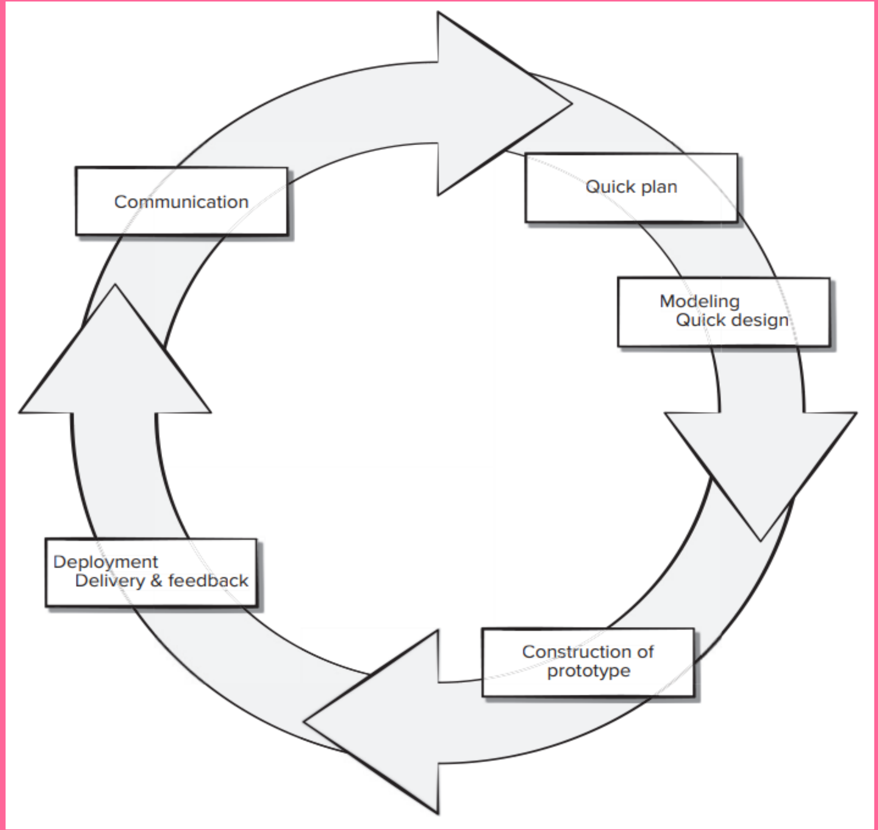
The quick design leads to the construction of a prototype.

The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.

Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Problems:

1. Stakeholders see what appears to be a working version of the software. They may be unaware that the prototype architecture (program structure) is also evolving. This means that the developers may not have considered the overall software quality or long-term maintainability.
2. As a software engineer, you may be tempted to make implementation compromises to get a prototype working quickly. If you are not careful, these less than-ideal choices have now become an integral part of the evolving system


Communication
Quick plan
Modeling
Quick design
Construction of
prototype
Deployment
Delivery & feedback

Lecture 5:

The Spiral Model:

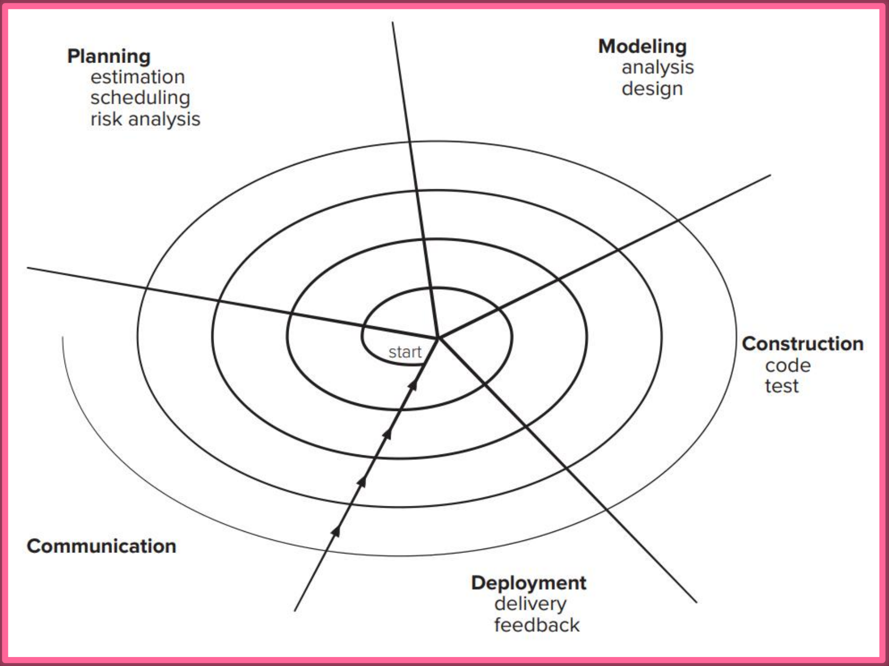
- Suggested by Boehm (1988)
- Combines development activities with risk management to minimize and control risks
- The model is presented as a spiral in which each iteration is represented by a circuit around four major activities
    - Plan
    - Determine goals, alternatives and constraints
    - Evaluate alternatives and risks
    - Develop and test

1. The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software.
2. Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.
3. A spiral model is divided into a set of framework activities defined by the software engineering team.
4. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is made. Anchor point milestones—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.
5. The first circuit around the spiral (beginning at the inside streamline nearest the center) might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
6. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.
7. Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. The spiral model is a realistic approach to the development of large-scale systems and software. It uses prototyping as a risk reduction mechanism.

Problems:

1. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.
2. It may be difficult to convince customers that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success.

3. If a major risk is not uncovered and managed, problems will undoubtedly occur. We have already noted that modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer-user satisfaction.
4. In many cases, time to market is the most important management requirement. If a market window is missed, the software project itself may be meaningless.
5. The intent of evolutionary models is to develop high-quality software in an iterative or incremental manner. However, it is possible to use an evolutionary process to emphasize flexibility, extensibility, and speed of development.



The Unified Process Model:

It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

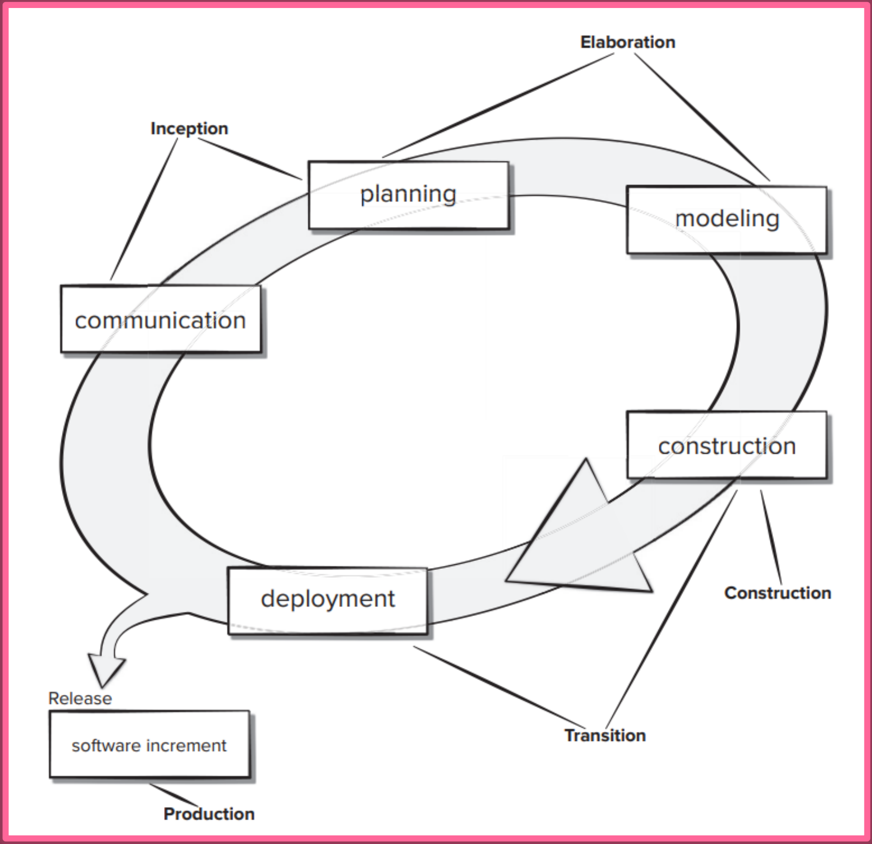
- UP should be tailored to organizational and project needs
- Highly iterative life cycle
- Project will be use-case driven and modeled using UML

1. **UP life cycle:**
    - Includes four phases which consist of iterations
    - Iterations are "mini-projects"

- **Four Phases:**

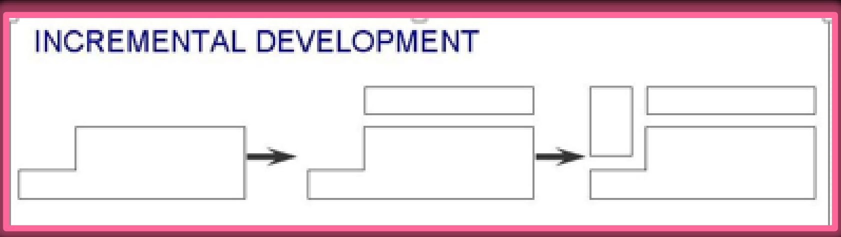
- Inception – develop and refine system vision
- Elaboration – define requirements and design and implement core architecture
- Construction – continue design and implementation of routine, less risky parts
- Transition – move the system into operational mode

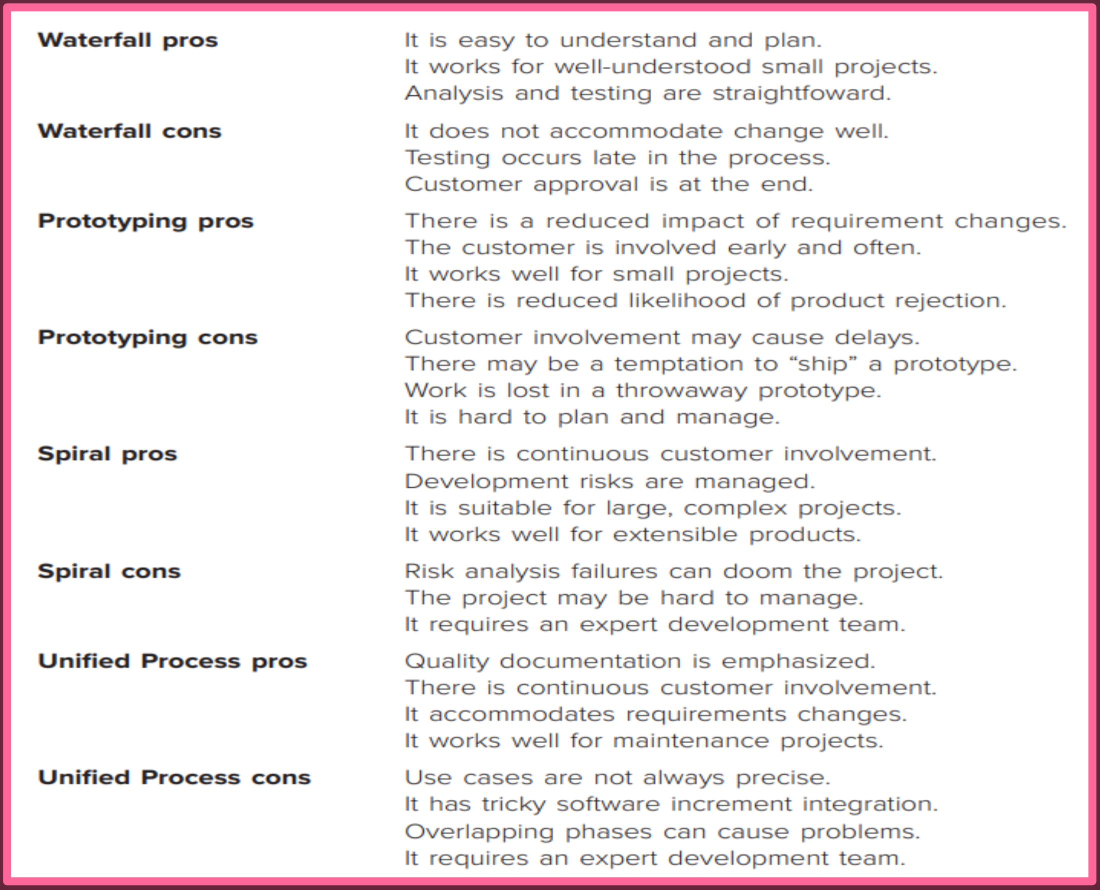
1. The inception phase of the UP is where customer communication and planning takes place. Fundamental business requirements are described through a set of preliminary use cases that describe which features and functions each major class of users desires that will become realized in the software architecture. Planning identifies resources, assesses major risks, and defines a preliminary schedule for the software increments.
2. The elaboration phase encompasses the planning and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases and creates an architectural baseline that includes five different views of the software—the use case model, the analysis model, the design model, the implementation model, and the deployment model. Modifications to the plan are often made at this time.
3. The construction phase of the UP is identical to the construction activity defined for the generic software process. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests are designed and executed for each. In addition, integration activities (component assembly and integration testing) are conducted. Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.
4. The transition phase of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software and supporting documentation is given to end users for beta testing, and user feedback reports both defects and necessary changes. At the conclusion of the transition phase, the software increment becomes a usable software release.
5. The production phase of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated. It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency. It should be noted that not every task identified for a UP workflow is conducted for every software project. The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.



Phased Development

- Cycle time
  - Time between when requirements document was written and when the system was delivered
- Shorter cycle time
- Decomposed system
  - System delivered in pieces
    - enables customers to have some functionality while the rest is being developed
- Two systems functioning in parallel
  - the production system (release n): currently being used
  - the development system (release n+1): the next version
- Incremental development: starts with small functional subsystem and adds functionality with each new release





Waterfall pros	It is easy to understand and plan. It works for well-understood small projects. Analysis and testing are straightfoward.
Waterfall cons	It does not accommodate change well. Testing occurs late in the process. Customer approval is at the end.
Prototyping pros	There is a reduced impact of requirement changes. The customer is involved early and often. It works well for small projects. There is reduced likelihood of product rejection.
Prototyping cons	Customer involvement may cause delays. There may be a temptation to “ship” a prototype. Work is lost in a throwaway prototype. It is hard to plan and manage.
Spiral pros	There is continuous customer involvement. Development risks are managed. It is suitable for large, complex projects. It works well for extensible products.
Spiral cons	Risk analysis failures can doom the project. The project may be hard to manage. It requires an expert development team.
Unified Process pros	Quality documentation is emphasized. There is continuous customer involvement. It accommodates requirements changes. It works well for maintenance projects.
Unified Process cons	Use cases are not always precise. It has tricky software increment integration. Overlapping phases can cause problems. It requires an expert development team.

Lecture 6:

Agility:

- Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning.
- In many situations, you won’t be able to define requirements fully before the project begins. You must be agile enough to respond to a fluid business environment.
- Fluidity implies change, and change is expensive—particularly if it is uncontrolled or poorly managed. One of the most compelling characteristics of the agile approach is its ability to reduce the costs of change through the software process.
- The prescriptive process models have a major failing: they forget the frailties of the people who build computer software.

- Software engineers are not robots. They exhibit great variation in working styles and significant differences in skill level, creativity, orderliness, consistency, and spontaneity. Some communicate well in written form, others do not.
- If process models are to work, they must provide a realistic mechanism for encouraging the discipline that is necessary, or they must be characterized in a manner that shows "tolerance" for the people who do software engineering work.

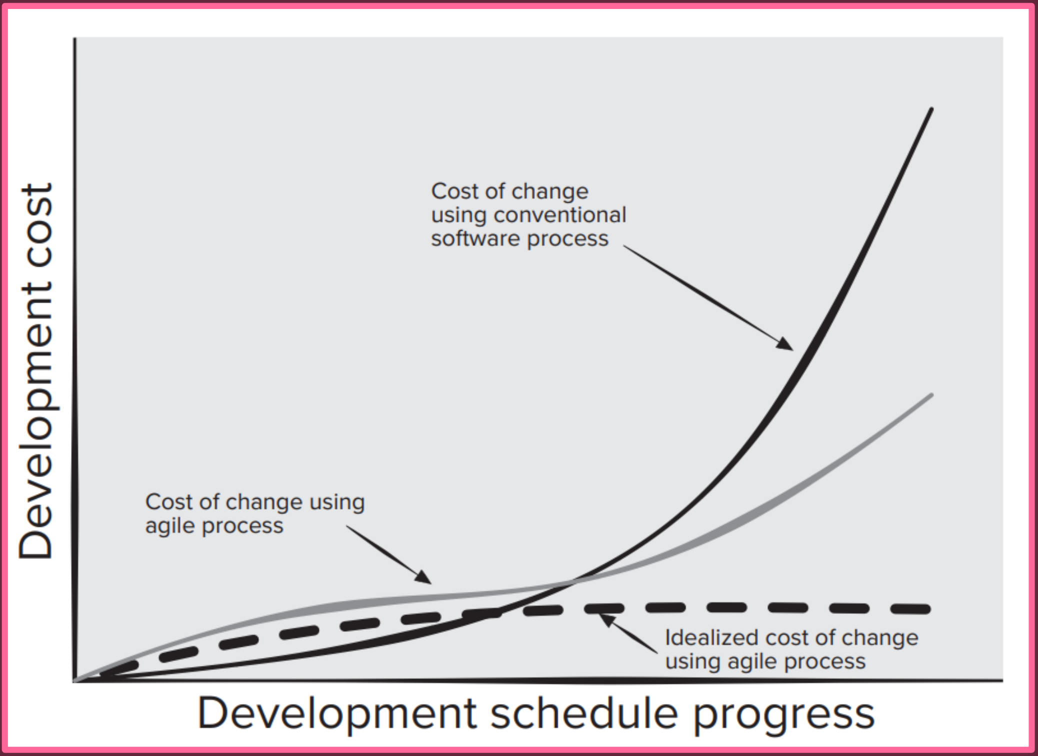
What is Agility?

The pervasiveness of change is the primary driver for agility

Manifesto:

- Concentrate on responding to change rather than on creating a plan and then following it (chaordic)
- Value individuals and interactions over process and tools
- Prefer to invest time in producing working software rather than in producing comprehensive documentation
- Focus on customer collaboration rather than contract negotiation

1. It encourages team structures and attitudes that make communication (among team members, between technologists and business people, and between software engineers and their managers) more simplistic.
2. It emphasizes rapid delivery of operational software and deemphasizes the importance of intermediate work products (not always a good thing)
3. It adopts the customer as a part of the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects
4. It recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.
5. Agility can be applied to any software process.
6. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.



What is an Agile Process?

Any agile software process is characterized in a manner that addresses a number of key assumptions

1. It is difficult to predict in advance which software requirements will persist and which will change.
2. It is equally difficult to predict how customer priorities will change as the project proceeds.
3. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem (a bicycle with seats and pedals for two riders, one behind the other.) so that design models are proven as they are created.
4. It is difficult to predict how much design is necessary before construction is used to prove the design.
5. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

How do we create a process that can manage unpredictability?

The answer lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be adaptable. But continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt incrementally. To accomplish incremental adaptation, an agile team requires customer feedback (so that the appropriate adaptations can be made). An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an incremental development strategy should be instituted. Software increments (executable prototypes or portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

Agility Principles:

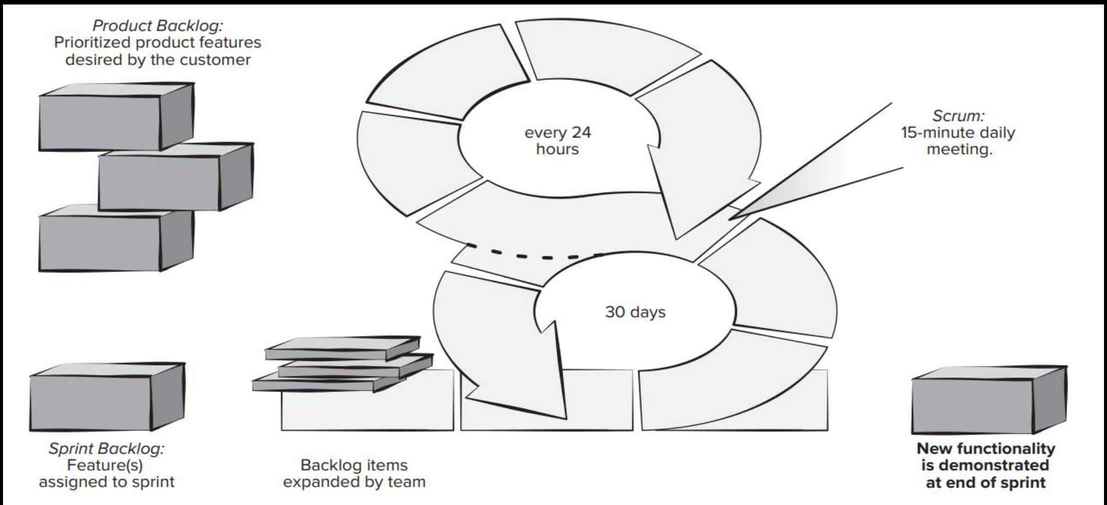
The Agile Alliance defines 12 principles for those software organizations that want to achieve agility.

1. Customer satisfaction is achieved by providing value through software that is delivered to the customer as rapidly as possible. To achieve this, agile developers recognize that requirements will change. They deliver software increments frequently and work together with all stakeholders so that feedback on their deliveries is rapid and meaningful.
2. An agile team is populated by motivated individuals, who communicate face-to face and work in an environment that is conducive to high quality software development. The team follows a process that encourages technical excellence and good design, emphasizing simplicity.
3. Working software that meets customer needs is their primary goal, and the pace and direction of the team's work must be "sustainable," enabling them to work effectively for long periods of time.
4. An agile team is a "self-organizing team"—one that can develop well-structured architectures that lead to solid designs and customer satisfaction. Part of the team culture is to consider its work introspectively, always with the intent of improving the manner in which it addresses its primary goal.
5. Not every agile process model applies characteristics described in this section with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more agile principles

Scrum:

Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery.

Within each framework activity, work tasks take place in a relatively short time-boxed (A time-box is a project management term that indicates a period of time that has been allocated to accomplish some task) period called a sprint. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on size of the product and its complexity) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.



Product Backlog:
Prioritized product features desired by the customer
every 24 hours
Scrum:
15-minute daily meeting.
30 days
Sprint Backlog:
Feature(s) assigned to sprint
Backlog items expanded by team
New functionality is demonstrated at end of sprint

Scrum Teams and Artifacts:

The Scrum team is a self-organizing interdisciplinary team consisting of a product owner, a Scrum master, and a small (three to six people) development team.

The principle Scrum artifacts are the product backlog, the sprint backlog, and the code increment. Development proceeds by breaking the project into a series of incremental prototype development periods 2 to 4 weeks in length called sprints.

The product backlog is a prioritized list of product requirements or features that provide business value for the customer.

Items can be added to the backlog at any time with the approval of the product owner and the consent of the development team.

The product owner orders the items in the product backlog to meet the most important goals of all stakeholders. The product backlog is never complete while the product is evolving to meet stakeholder needs.

The product owner is the only person who decides whether to end a sprint prematurely or extend the sprint if the increment is not accepted.

The sprint backlog is the subset of product backlog items selected by the product team to be completed as the code increment during the current active sprint.

The increment is the union of all product backlog items completed in previous sprints and all backlog items to be completed in the current sprints.

The development team creates a plan for delivering a software increment containing the selected features intended to meet an important goal as negotiated with the product owner in the current sprint.

Most sprints are time-boxed to be completed in 3 to 4 weeks. How the development team completes the increment is left up to the team to decide. The development team also decides when the increment is done and ready to demonstrate to the product owner.

No new features can be added to the sprint backlog unless the sprint is cancelled and restarted.

The Scrum master serves as facilitator to all members of the Scrum team. She runs the daily Scrum meeting and is responsible for removing obstacles identified by team members during the meeting. She coaches the development team members to help each other complete sprint tasks when they have time available. She helps the product owner find techniques for managing the product backlog items and helps ensure that backlog items are stated in clear and concise terms.

Sprint Planning Meeting

Prior to beginning, any development team works with the product owner and all other stakeholders to develop the items in the product backlog.

The product owner and the development team rank the items in the product backlog by the importance of the owner’s business needs and the complexity of the software engineering tasks (programming and testing) required to complete each of them.

Sometimes this results in the identification of missing features needed to deliver the required functionality to the end users.

Prior to starting each sprint, the product owner states her development goal for the increment to be completed in the upcoming sprint.

The Scrum master and the development team select the items to move from to the sprint backlog. The development team determines what can be delivered in the increment within the constraints of the time-box allocated for the sprint and, with the Scrum master, what work will be needed to deliver the increment.

The development team decides which roles are needed and how they will need to be filled.

Daily Scrum Meeting

The daily Scrum meeting is a 15-minute event scheduled at the start of each workday to allow team members to synchronize their activities and make plans for the next 24 hours.

The Scrum master and the development team always attend the daily Scrum.

Some teams allow the product owner to attend occasionally.

Three key questions are asked and answered by all team members:-

1. What did you do since the last team meeting?
2. What obstacles are you encountering?
3. What do you plan to accomplish by the next team meeting?

The Scrum master leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. It is the Scrum master's task to clear obstacles presented before the next Scrum meeting if possible. These are not problem-solving meetings, those occur off-line and only involve the affected parties.

Also, these daily meetings lead to "knowledge socialization" and thereby promote a self-organizing team structure. Some teams use these meetings to declare sprint backlog items complete or done. When the team considers all sprint backlog items complete, the team may decide to schedule a demo and review of the completed increment with the product owner

The XP Framework

Extreme Programming encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing

Planning:

1. The planning activity (also called the planning game) begins with a requirements activity called listening.
2. Listening leads to the creation of a set of "stories" (also called user stories) that describe required output, features, and functionality for software to be built.

3. Each user story is written by the customer and is placed on an index card. The customer assigns a value (i.e., a priority) to the story based on the overall business value of the feature or function.
4. Members of the XP team then assess each story and assign a cost—measured in development weeks—to it.
5. It is important to note that new stories can be written at any time.
6. Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team.
7. Once a basic commitment (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways:
    i. all stories will be implemented immediately (within a few weeks)
    ii. the stories with highest value will be moved up in the schedule and implemented first
    iii. the riskiest stories will be moved up in the schedule and implemented first.

After the first project release (also called a software increment) has been delivered, the XP team computes project velocity.

Stated simply, project velocity is the number of customer stories implemented during the first release.

Project velocity can then be used to help estimate delivery dates and schedule for subsequent releases. The XP team modifies its plans accordingly.

Design:

1. XP design rigorously follows the KIS (keep it simple) principle. The design of extra functionality (because the developer assumes it will be required later) is discouraged.
2. XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context.
3. CRC (class-responsibility collaborator) cards identify and organize the object-oriented classes that are relevant to the current software increment.
4. CRC cards are the only design work product produced as part of the XP process. If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design.
5. A central notion in XP is that design occurs both before and after coding commences.
6. Refactoring—modifying/optimizing the code in a way that does not change the external behavior of the software means that design occurs continuously as the system is constructed.
7. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

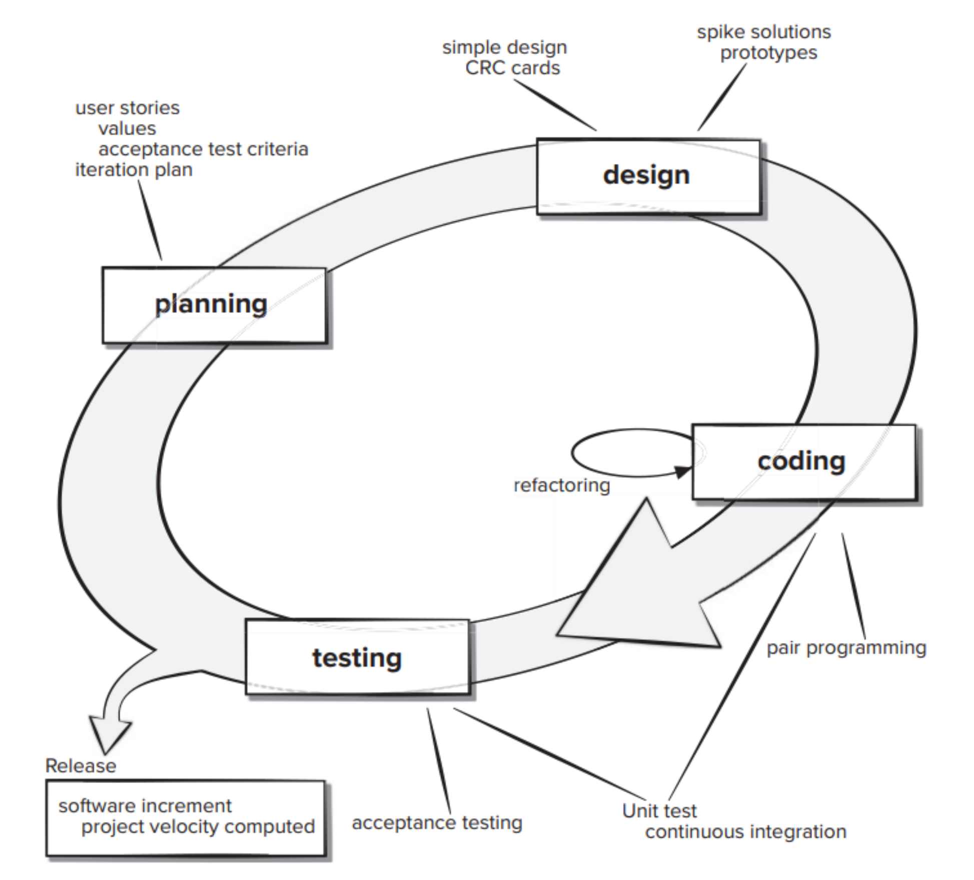
Coding:

1. After user stories are developed and preliminary design work is done, the team does not move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).
2. Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the test.

3. Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.
4. A key concept during the coding activity (and one of the most talked-about aspects of XP) is pair programming.
5. XP recommends that two people work together at one computer to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created).
6. As pair programmers complete their work, the code they develop is integrated with the work of others.
7. This "continuous integration" strategy helps uncover compatibility and interfacing errors early.

Testing:

The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages implementing a regression testing strategy whenever code is modified (which is often, given the XP refactoring philosophy). XP acceptance tests, also called customer tests, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. They are derived from user stories that have been implemented as part of a software release.



Kanban:

1. The Kanban method is a lean methodology that describes methods for improving any process or workflow. Kanban is focused on change management and service delivery. Change management defines the process through which a requested change is integrated into a software-based system. Service delivery is encouraged by focusing on understanding customer

needs and expectations. The team members manage the work and are given the freedom to organize themselves to complete it. Policies evolve as needed to improve outcomes.

2. Kanban itself depends on six core practices:
3. Visualizing workflow using a Kanban board. The Kanban board is organized into columns representing the development stage for each element of software functionality. The cards on the board might contain single user stories or recently discovered defects on sticky notes and the team would advance them from “to do,” to “doing,” and “done” as the project progresses.
4. Limiting the amount of work in progress (WIP) at any given time. Developers are encouraged to complete their current task before starting another. This reduces lead time, improves work quality, and increases the team’s ability to deliver software functionality frequently to their stakeholders.
5. Managing workflow to reduce waste by understanding the current value flow, analyzing places where it is stalled, defining changes, and then implementing the changes.
6. Making process policies explicit (e.g., write down your reasons for selecting items to work on and the criteria used to define “done”).
7. Focusing on continuous improvement by creating feedback loops where changes are introduced based on process data and the effects of the change on the process are measured after the changes are made.
8. Make process changes collaboratively and involve all team members and other stakeholders as needed.

The team meetings for Kanban are like those in the Scrum framework.

If Kanban is being introduced to an existing project, not all items will start in the backlog column.

Developers need to place their cards in the team process column by asking themselves:

1. Where are they now?
2. Where did they come from?
3. Where are they going?

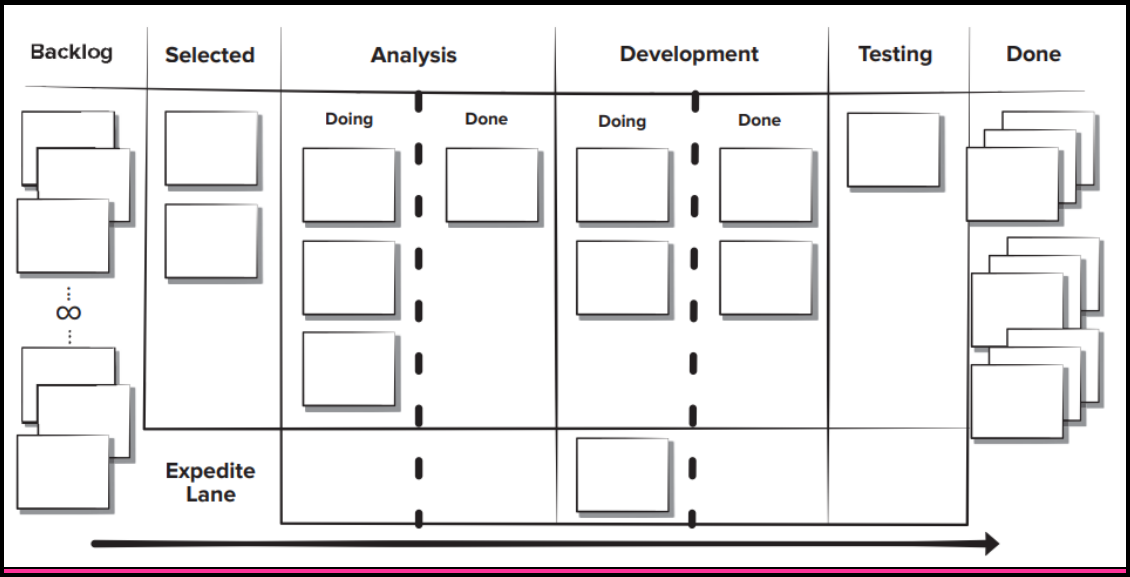
The basis of the daily Kanban standup meeting is a task called “walking the board.”

Leadership of this meeting rotates daily.

The team members identify any items missing from the board that are being worked on and add them to the board. The team tries to advance any items they can to “done.” The goal is to advance the high business value items first.

The team looks at the flow and tries to identify any impediments to completion by looking at workload and risks. During the weekly retrospective meeting, process measurements are examined.

The team considers where process improvements may be needed and proposes changes to be implemented. Kanban can easily be combined with other agile development practices to add a little more process discipline.


Backlog
Selected
Analysis
Development
Testing
Done
Doing
Done
Doing
Done
∞
Expedite Lane