COMP 433 Software Engineering

Module 2: *Software Processes*

Ahmed Tamrawi

atamrawi atamrawi.github.io ahmedtamrawi@gmail.com

Acknowledgment Notice

Part of the slides are based on content from Ian Somerville's Software Engineering book.

Software Processes

A structured set of activities required to develop a software system

The Software Process

- Many different software processes but all involve:
  - **Specification** – *defining what the system should do;*
  - **Design and implementation** – *defining the organization of the system and implementing the system;*
  - **Validation** – *checking that system does what the customer wants;*
  - **Evolution** – *changing the system in response to changing customer needs.*
- These activities are **complex activities** in themselves, and they include sub-activities such as requirements validation, architectural design, and unit testing.
- **Processes also include other activities**, such as software configuration management and project planning that support production activities.
- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

Software Process Descriptions

- When we describe and discuss processes, we usually talk about the **activities** in these processes such as specifying a data model, designing a user interface, etc. and the **ordering** of these activities.
- Process descriptions may also include:
  - **Products**, which are the outcomes of a process activity;
  - **Roles**, which reflect the responsibilities of the people involved in the process;
  - **Pre- and post-conditions**, which are statements that are true before and after a process activity has been enacted or a product produced.
    - *For example, before architectural design begins, a **precondition** may be that the consumer has approved all requirements; after this activity is finished, a **postcondition** might be that the UML models describing the architecture have been reviewed.*

Software Processes

- Software processes are **complex** and, like all **intellectual** and **creative** processes rely on people making decisions and judgments.
- There is **no universal process** that is right for all kinds of software.
- Most software companies have developed their own development processes.
- Processes have evolved to take advantage of the capabilities of the software developers in an organization and the characteristics of the systems that are being developed.
- For safety-critical systems, a very **structured development process** is required where detailed records are maintained. For business systems, with rapidly changing requirements, a **more flexible, agile process** is likely to be better.

Software Process Models

sometimes called a Software Development Life Cycle or SDLC model

- **The waterfall model**

  Plan-driven model. Separate and distinct phases of specification and development.

- **Incremental development**

  Specification, development and validation are interleaved. May be plan-driven or agile.

- **Integration and configuration**

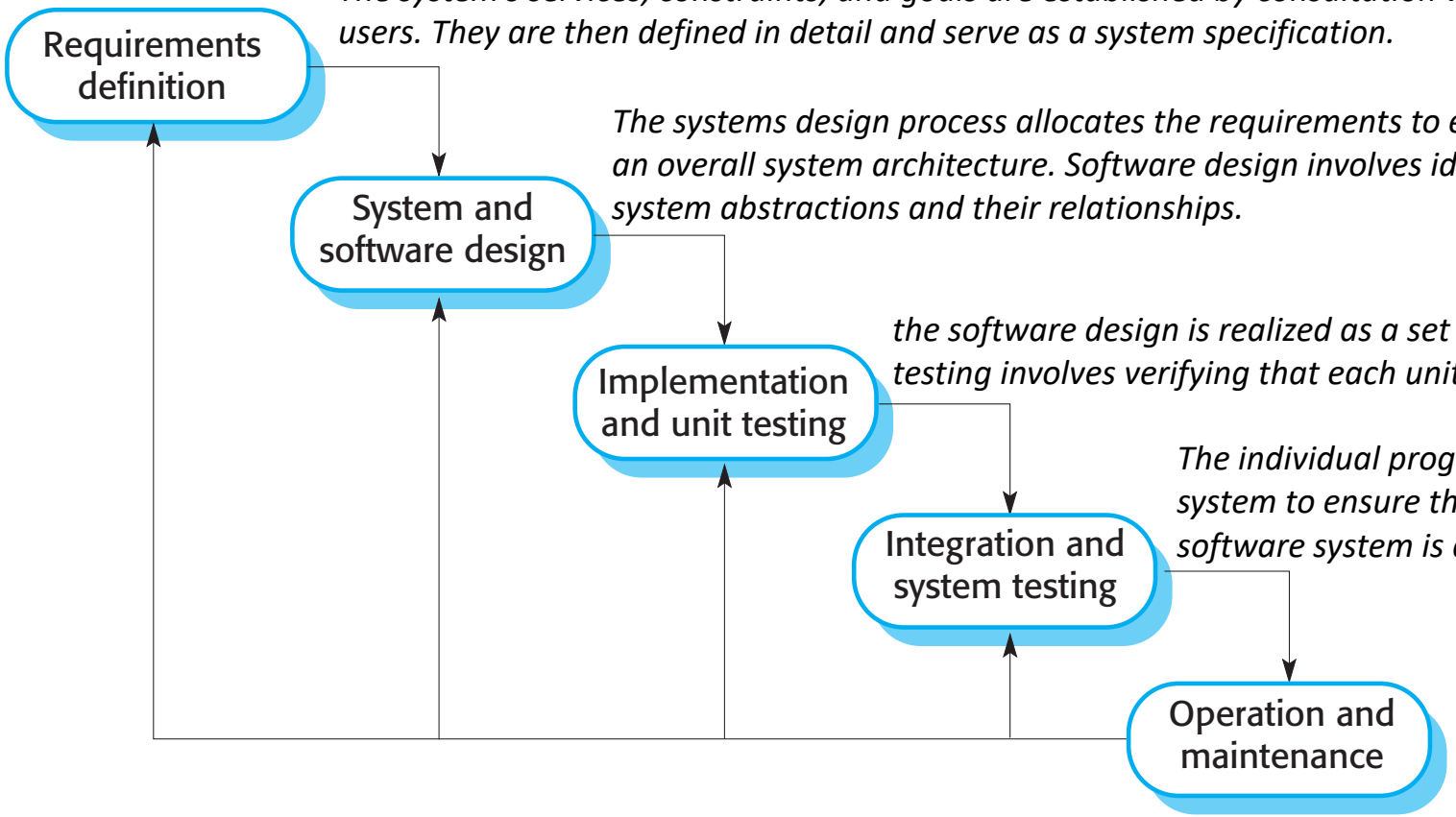
  The system is assembled from existing configurable (reusable) components. May be plan-driven or agile.

- In practice, most large systems are developed using a process that incorporates elements from all of these models.

Software Process Models

- Various attempts have been made to develop "**universal**" process models that draw on all of these general models.
- One of the best known of these universal models is the **Rational Unified Process (RUP)** (Krutchen 2003), which was developed by Rational, a U.S. software engineering company.
- The RUP is a **flexible** model that can be instantiated in different ways to create processes that resemble any of the general process models discussed here.
- The RUP has been adopted by some large software companies (notably IBM), but it has not gained widespread acceptance.

The Waterfall Model



The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.

The systems design process allocates the requirements to either hardware or software systems. It establishes an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

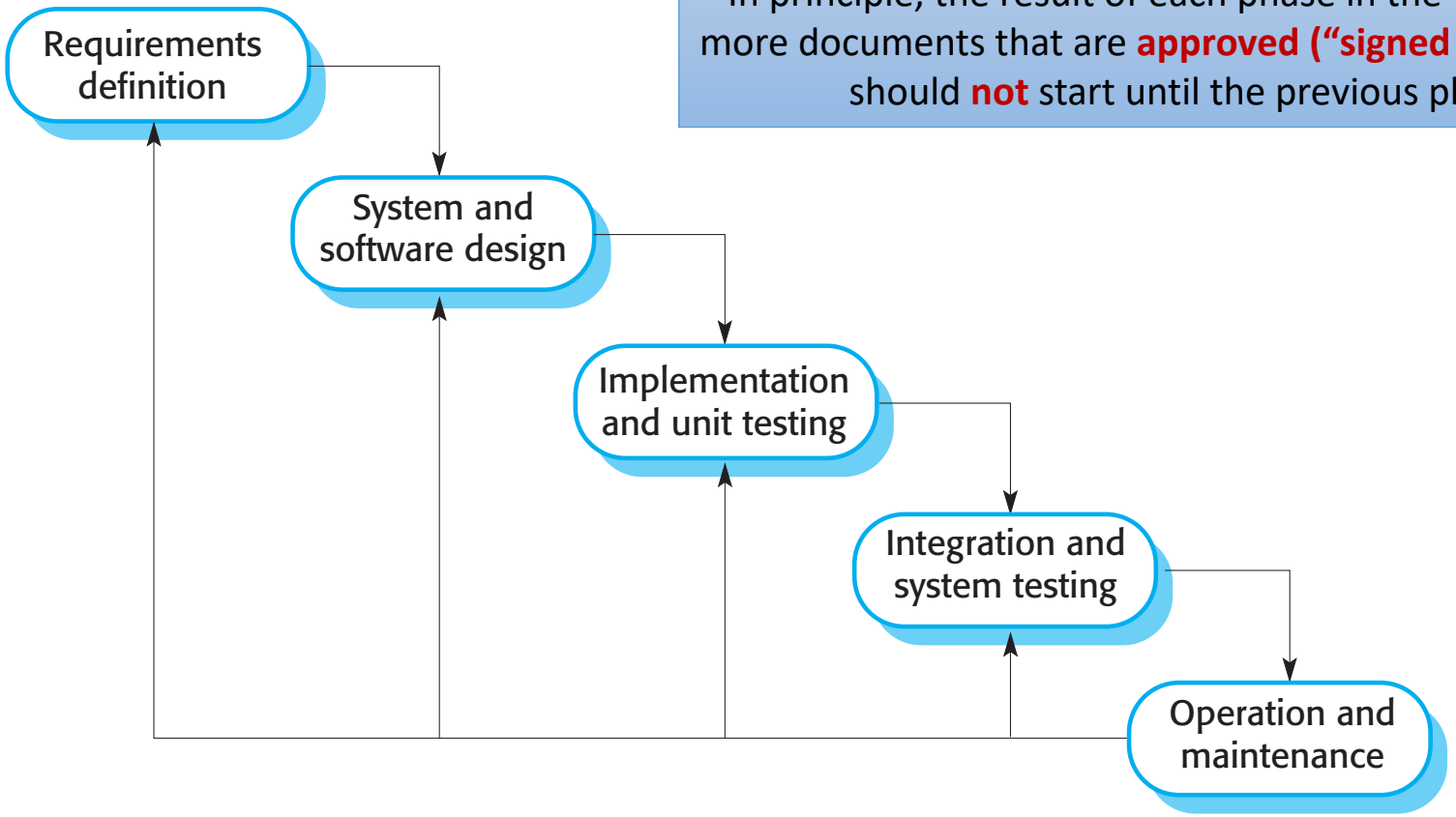
the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

Normally, this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors that were not discovered in earlier stages of the life cycle, improving the implementation of system units, and enhancing the system's services as new requirements are discovered.

The Waterfall Model

In principle, the result of each phase in the waterfall model is one or more documents that are **approved ("signed off").** The following phase should **not** start until the previous phase has finished.



The **main drawback** of the waterfall model is the *difficulty of accommodating change after the process is underway*. In principle, a phase has to be complete before moving onto the next phase.

The Waterfall Model

- In reality, software has to be flexible and accommodate change as it is being developed.
- The need for early commitment and system rework when changes are made means that the waterfall model is only appropriate for some types of system:
  - *Embedded systems* where the software has to interface with hardware systems (hardware inflexibility).
  - *Critical systems* where there is a need for extensive safety and security analysis of the software specification and design.
  - *Large software systems* that are part of broader engineering systems developed by several partner companies.

Advantages of Waterfall Model

- Developers and customers agree on what will be delivered **early** in the development lifecycle. This makes planning and designing more **straightforward**.
- Progress is more **easily measured**, as the full scope of the work is known in advance.
- Throughout the development effort, it’s possible for various members of the team to be involved or to continue with other work, depending on the active phase of the project
- Customer presence **is not strictly required** after the requirements phase.
- The software can be designed **completely** and **more carefully**, based upon a more complete understanding of all software deliverables.

Waterfall Model Problems

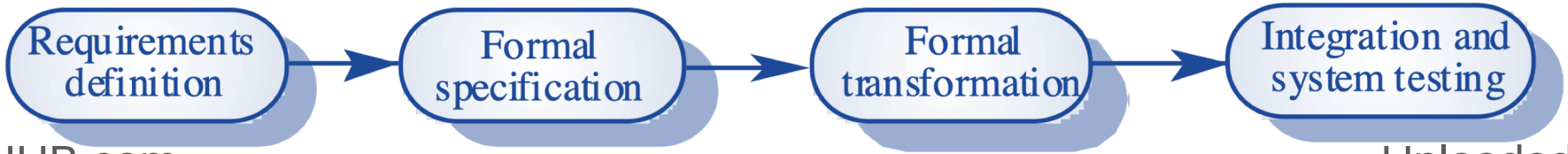
- **Inflexible partitioning of the project** into distinct stages makes it difficult to respond to changing customer requirements.
  - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
  - Few business systems have stable requirements.
- The waterfall model is mostly used **for large systems engineering projects where a system is developed at several sites.**
  - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

Waterfall Model Problems

- One area which almost always falls short is the **effectiveness of requirements**.
- Gathering and documenting requirements in a way that is meaningful to a customer is often the **most difficult part of software development**.
- Another potential drawback of pure Waterfall development is the possibility that the customer will be **dissatisfied with their delivered software product**.
- Testing of whole system that **only happens at end of project**.

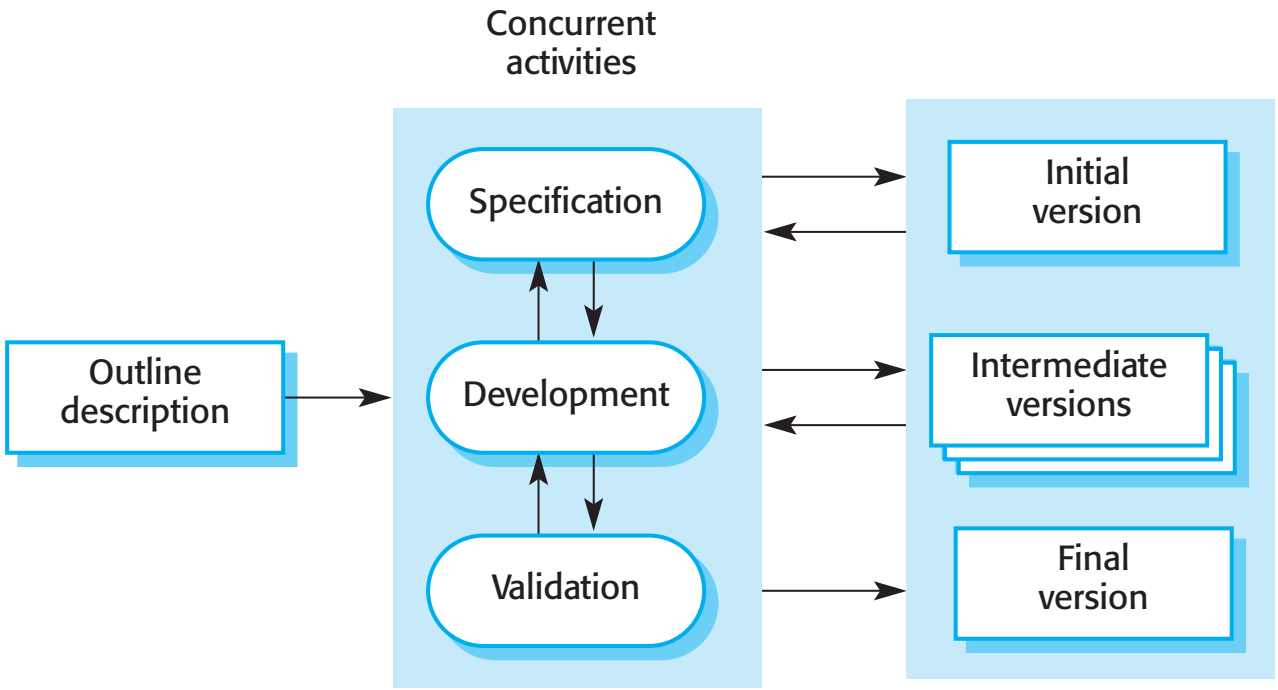
Formal System Development

- An important variant of the waterfall model is **formal system development**, where a *mathematical model of a system specification* is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code.
- This development process is particularly suited to the development of systems that have stringent **safety**, **reliability**, or **security** requirements.
- Formal methods are most likely to be applied to safety-critical or security-critical software and systems, such as **avionics software**.

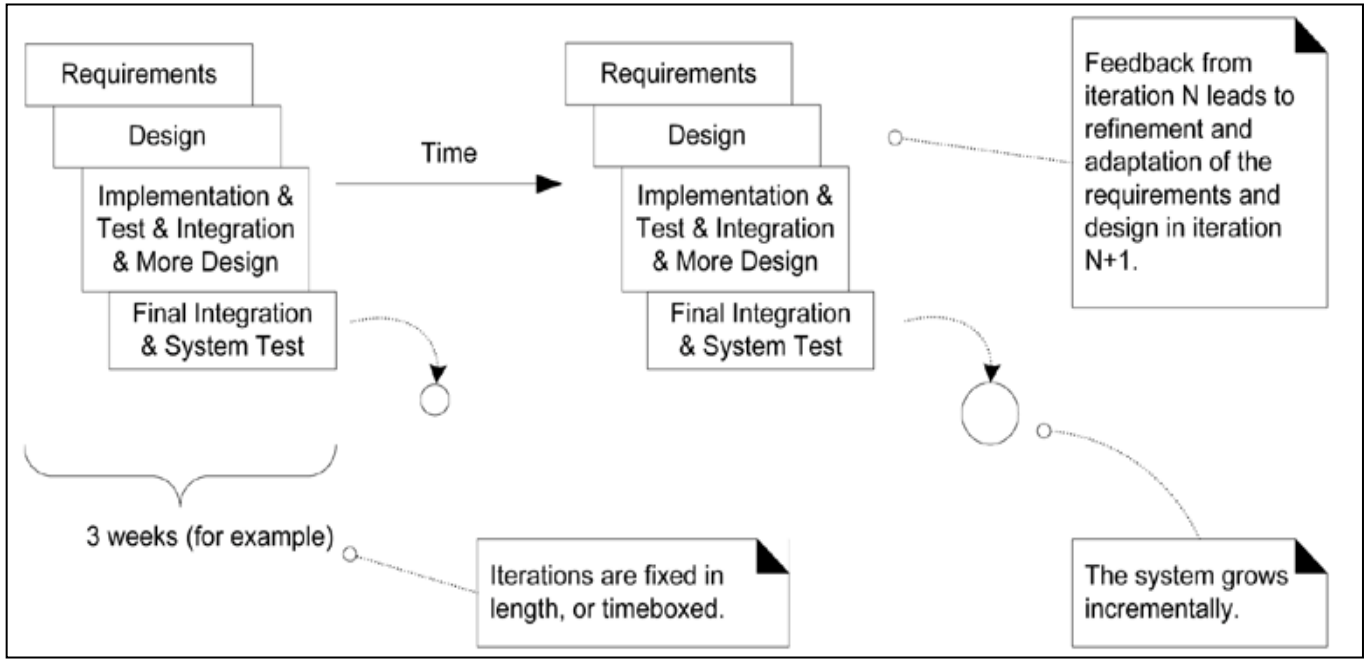


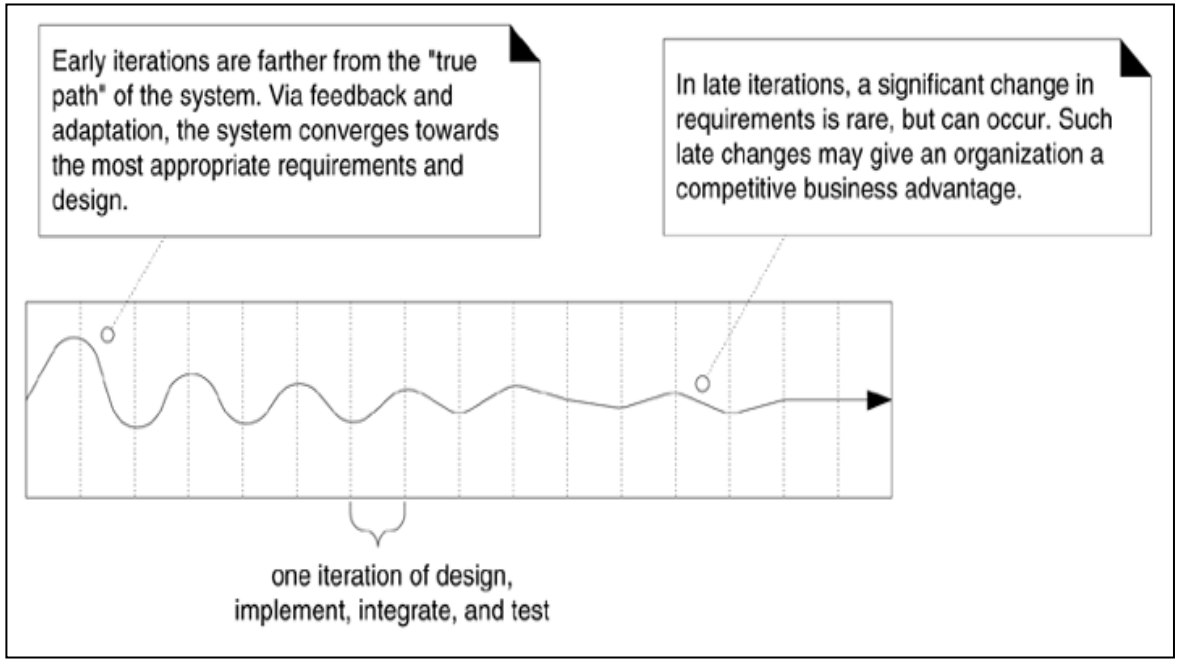
Incremental Development

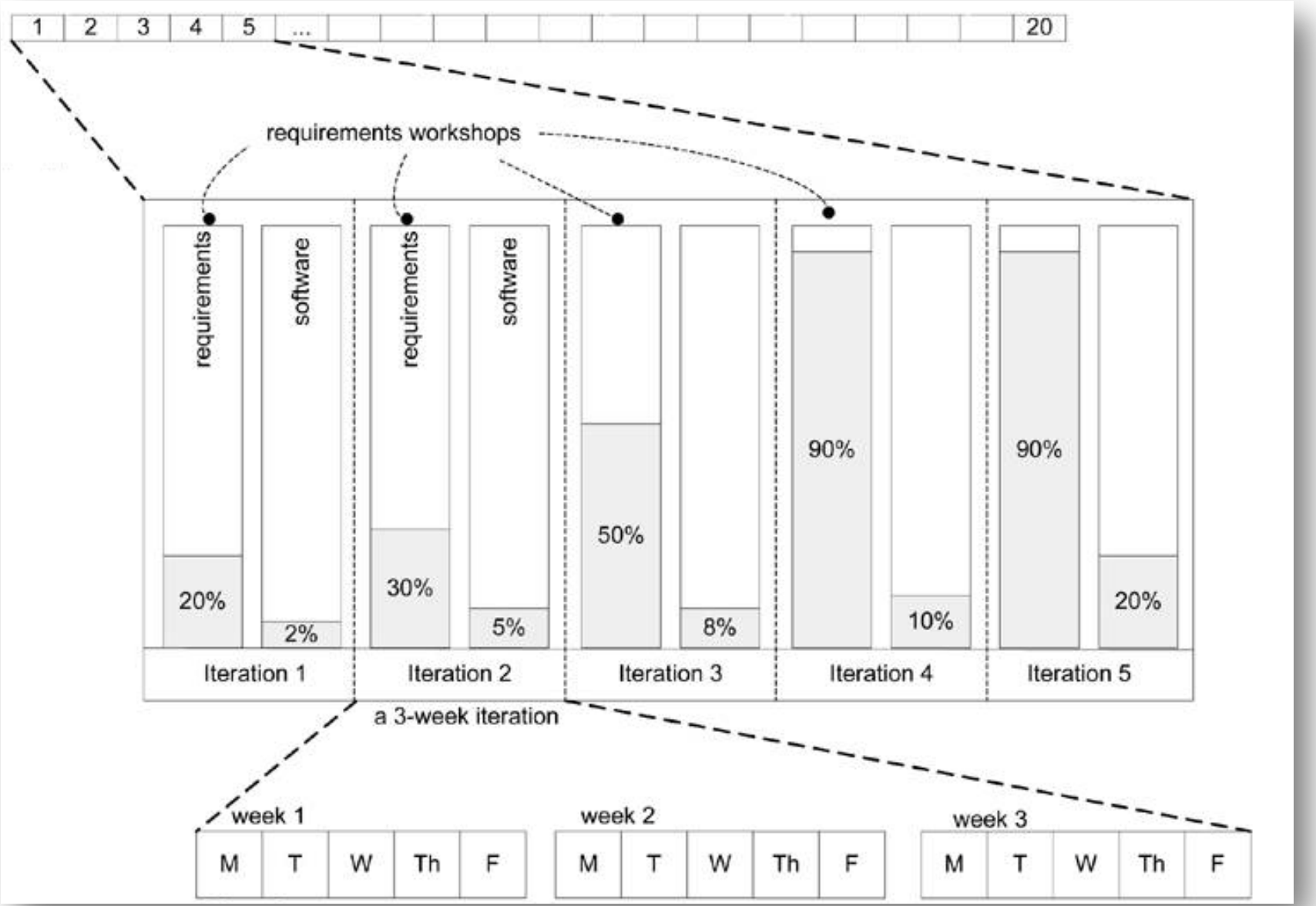
*Incremental development is based on the idea of developing an **initial** implementation, **getting feedback** from users and others, and **evolving the software** through several versions until the required system has been developed*



Each increment or version of the system incorporates some of the functionality that is needed by the customer. The early increments of the system include the most important or most urgently required functionality. This means that the customer or user can evaluate the system at a relatively early stage in the development to see if it delivers what is required.







Incremental Development Benefits

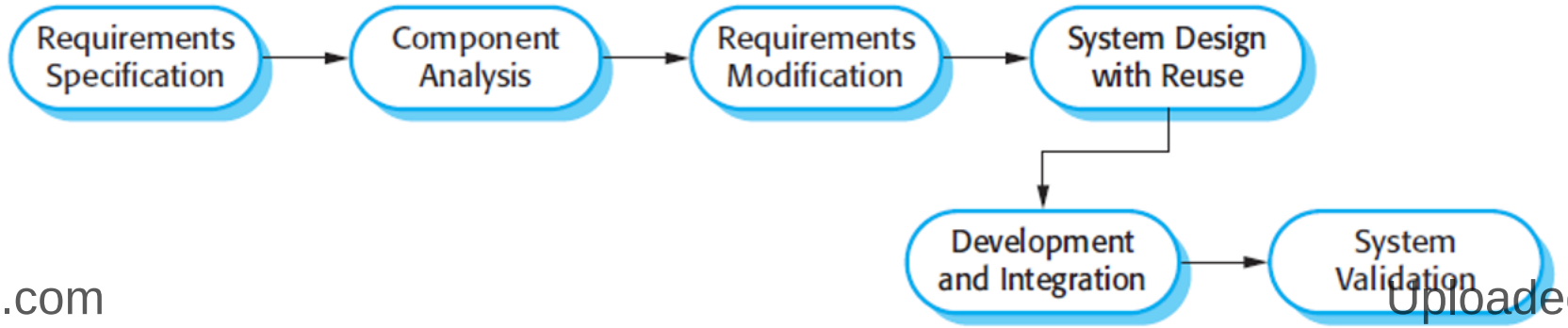
- The **cost** of accommodating changing customer requirements is reduced.
  - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- It is **easier** to get customer feedback on the development work that has been done.
  - Customers can comment on demonstrations of the software and see how much has been implemented.
- More **rapid** delivery and deployment of useful software to the customer is possible.
  - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Incremental Development Problems

- The process is **not visible**.
  - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- System structure tends to **degrade** as new increments are added.
  - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.
- The problems of incremental development become particularly **acute** for large, complex, long-lifetime systems, where different teams develop different parts of the system. Remember that Large systems need a *stable framework or architecture*.

Integration and Configuration

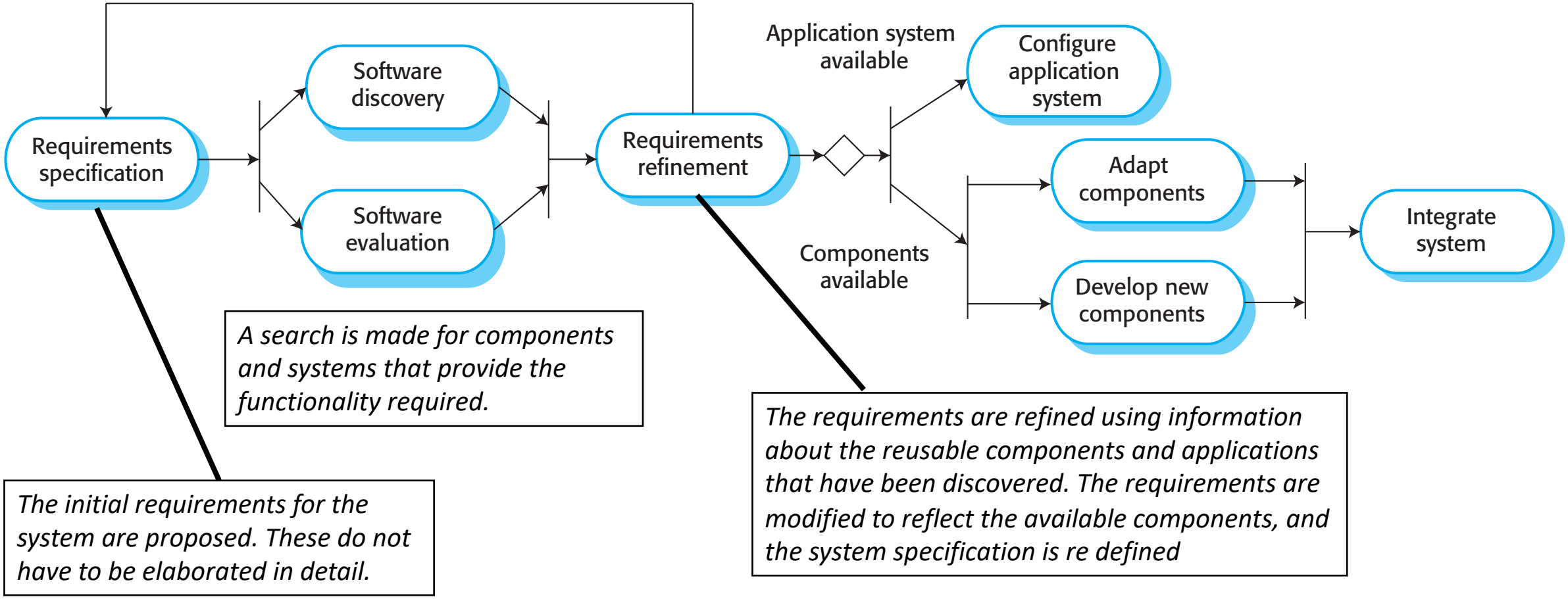
- Based on **software reuse** where systems are integrated from existing components or application systems (sometimes called COTS - Commercial-off-the-shelf) systems).
- Reused elements may be *configured* to adapt their behaviour and functionality to a user's requirements
- Reuse is now the standard approach for building many types of business system.



Types of Reusable Software

- **Stand-alone application systems** (sometimes called COTS) that are configured for use in a particular environment.
- **Collections of objects** that are developed as a package to be integrated with a component framework such as Java Spring Framework, .NET, or J2EE.
- **Web services** that are developed according to service standards and which are available for remote invocation.

Reuse-Oriented Software Engineering



Reuse-Oriented Software Engineering

- Advantages:
  - Reduced costs and risks as less software is developed from scratch.
  - Faster delivery and deployment of system.
- Disadvantages:
  - Requirements compromises are inevitable so system may not meet real needs of users.
  - Loss of control over evolution of reused system elements.

Agile Software Development

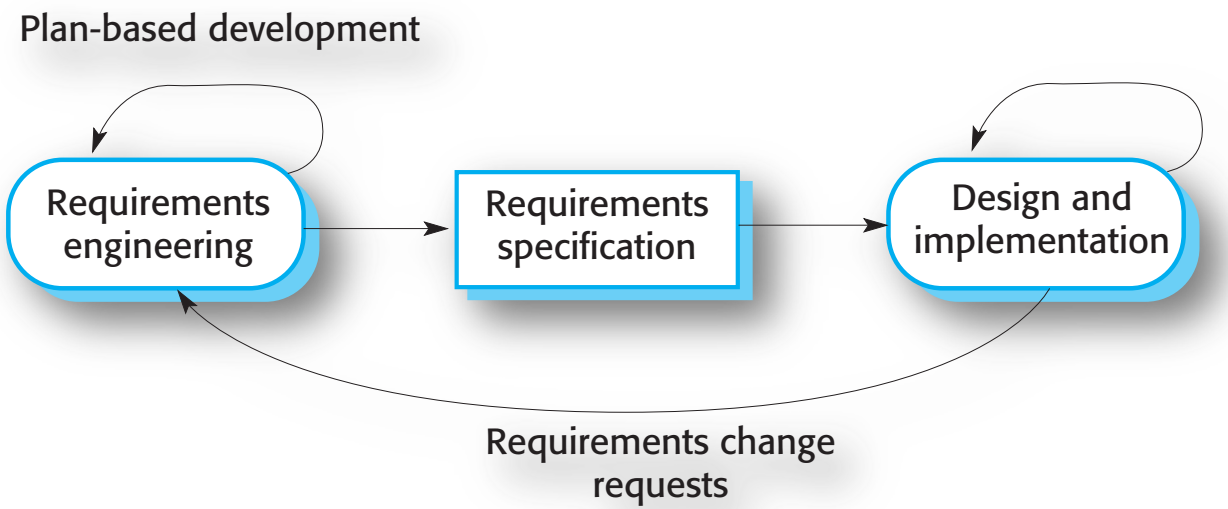
Rapid Software Development

- *Rapid development and delivery* is now often the most important requirement for software systems
  - Businesses operate in a fast – **changing requirement** and it is practically impossible to produce a set of stable software requirements
  - Software must **evolve quickly** to reflect changing business needs.
- Plan-driven development is essential for some types of system but does **not** meet these business needs.
- Agile development methods emerged in the late 1990s whose aim was to **radically reduce the delivery time** for working software systems.

Agile Development

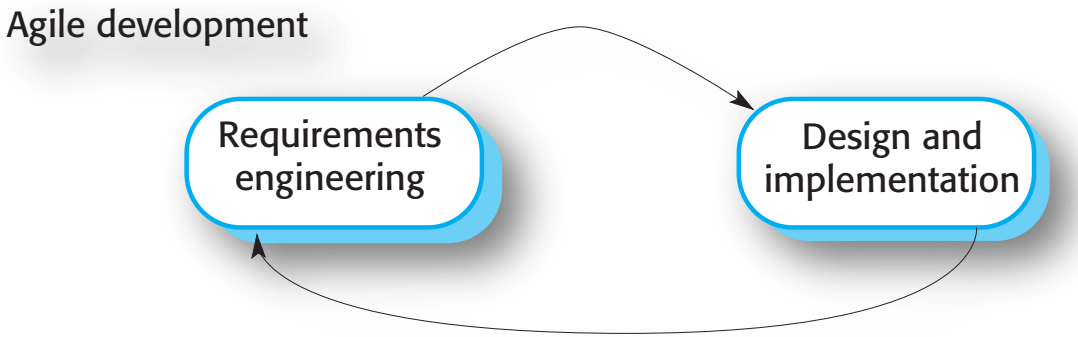
- Program specification, design and implementation are **inter-leaved**.
- The system is developed as a *series of versions or increments* with stakeholders involved in version specification and evaluation.
- **Frequent** delivery of new versions for evaluation.
- Extensive **tool support** (e.g., automated testing tools) used to support development.
- **Minimal documentation** – focus on working code.

Plan-Driven and Agile Development



Plan-Driven Development

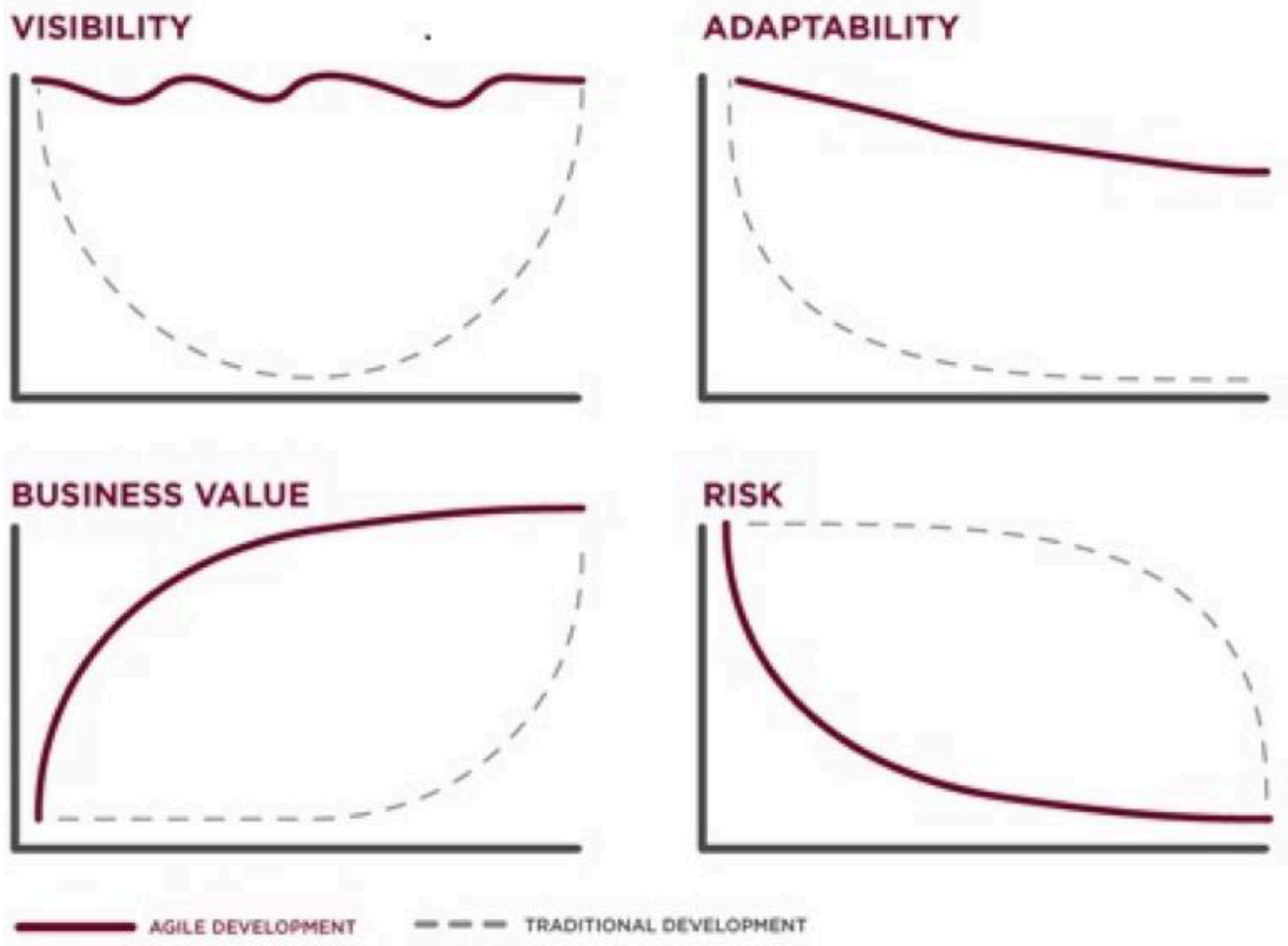
- A plan-driven approach to software engineering is based around **separate** development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible.
- Iteration occurs within activities.



Agile Development

Specification, design, implementation and testing are inter-leaved and the outputs from the development process are decided through a **process of negotiation** during the software development process.

Plan-Driven vs Agile Development



Agile Methods

Agile Methods

- Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
  - **Focus on the code** rather than the design
  - Are based on an **iterative approach** to software development
  - Are intended to **deliver working software quickly** and **evolve this quickly** to meet changing requirements.
- The aim of agile methods is to **reduce overheads** in the software process (e.g. by limiting documentation) and to be able to **respond quickly to changing requirements** without excessive rework.

Manifesto for Agile Software Development

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.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

The Principles of Agile Methods

Principle	Description
Customer involvement	Customers should be **closely involved** throughout the development process. Their role is provide and **prioritize** new system requirements and to **evaluate** the iterations of the system.
Incremental delivery	The software is developed in **increments** with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	**Expect** the system requirements to **change** and so design the system to **accommodate** these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, **actively work to eliminate complexity from the system**.

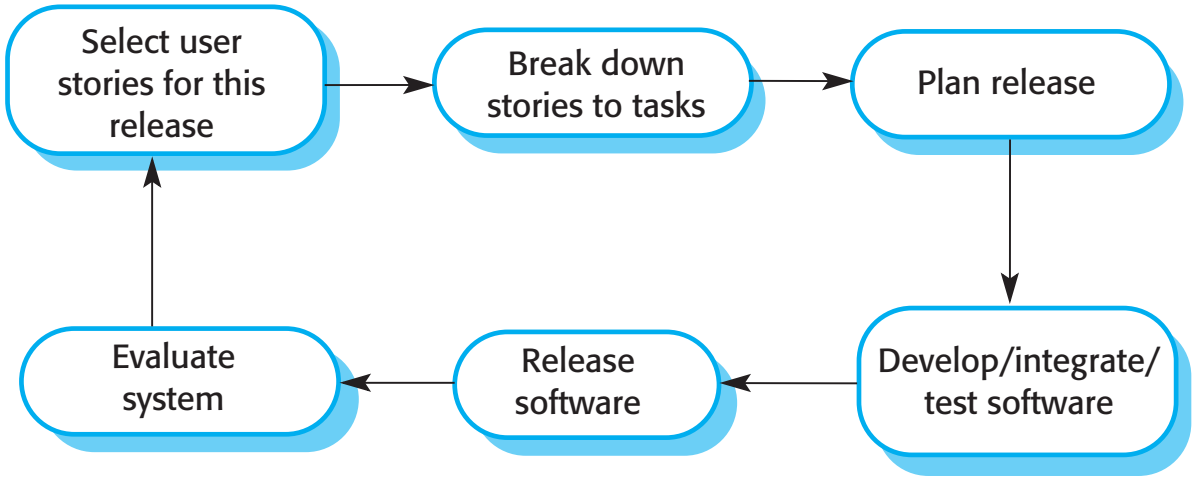
Agile Method Applicability

- **Product development** where a software company is developing a small or medium-sized product for sale.
  - Virtually all software products and apps are now developed using an agile approach
- **Custom system development** within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external rules and regulations that affect the software.

Agile Development Techniques

Extreme Programming

- A very influential agile method, developed in the late 1990s, that introduced a range of agile development techniques.
- Extreme Programming (XP) takes an 'extreme' approach to iterative development.
  - New versions may be built several times per day;
  - Increments are delivered to customers every 2 weeks;
  - All tests must be run for every build and the build is only accepted if tests run successfully.



Extreme Programming Practices

Principle or practice	Description
Incremental planning	Requirements are recorded on **story cards** and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into **development 'Tasks'**.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to re-factor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Extreme Programming Practices

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

XP and Agile Principles

- Incremental development is supported through **small, frequent system releases**.
- Customer involvement means **full-time customer engagement with the team**.
- People not process through pair programming, collective ownership and a process that avoids long working hours.
- Change supported through **regular system releases**.
- Maintaining simplicity through **constant refactoring of code**.

Influential XP Practices

- Extreme programming has a **technical focus** and is not easy to integrate with **management practice** in most organizations.
- Consequently, while agile development uses practices from XP, the method as originally defined is **not widely used**.

User Stories for Requirements

- In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- User requirements are expressed as **user stories or scenarios.**
- These are written on cards and the development team break them down into **implementation tasks**. These tasks are the basis of **schedule and cost estimates**.
- The customer chooses the **stories for inclusion in the next release** based on their priorities and the schedule estimates.

A ‘prescribing medication’ Story and Tasks

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.
Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.
Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low.
If within the range, enable the ‘Confirm’ button.

Refactoring

- Conventional wisdom in software engineering is to **design for change**. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- Rather, it proposes **constant code improvement (refactoring)** to make changes easier when they have to be implemented.

Refactoring

- Programming team look for possible software improvements and make these improvements even *where there is no immediate need for them*.
- This improves the **understandability** of the software and so **reduces the need for documentation**.
- Changes are easier to make because the code is well-structured and clear.
- However, some changes requires architecture refactoring, and this is much more expensive.

Examples of Refactoring

- Re-organization of a class hierarchy to remove duplicate code.
- Tidying up and renaming attributes and methods to make them easier to understand.
- The replacement of inline code with calls to methods that have been included in a program library.

Test-First Development

- Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
- XP testing features:
  - Test-first development.
  - Incremental test development from scenarios.
  - User involvement in test development and validation.
  - Automated test harnesses are used to run all component tests each time that a new release is built.

Test-Driven Development

- Writing tests before code clarifies the requirements to be implemented.
- Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
  - Usually relies on a testing framework such as Junit.
- All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.

Customer Involvement

- The role of the customer in the testing process is to *help develop acceptance tests* for the stories that are to be implemented in the next release of the system.
- The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- However, people adopting the customer role have **limited time available** and so **cannot work full-time with the development team**.
  - They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test Case Description for Dose Checking

Test 4: Dose checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

Test Automation

- Test automation *means that tests are written as executable components before the task is implemented*
  - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification.
  - An automated test framework (e.g., Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.
- As testing is **automated**, there is always a set of tests that can be quickly and easily executed
  - Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Problems with Test-First Development

- Programmers prefer **programming to testing** and sometimes they take short cuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- **Some tests can be very difficult to write incrementally**. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
- It difficult to judge the **completeness of a set of tests**. Although you may have a lot of system tests, your test set may not provide complete coverage.

Pair Programming

- Pair programming involves programmers working in pairs, developing code together.
- This helps develop common **ownership** of code and spreads **knowledge** across the team.
- It serves as an **informal review** process as each line of code is looked at by more than 1 person.
- It encourages refactoring as the whole team can benefit from improving the system code.
- In pair programming, programmers sit together at the same computer to develop the software.

Pair Programming

- Pairs are created **dynamically** so that all team members work with each other during the development process.
- The sharing of knowledge that happens during pair programming is very important as it **reduces the overall risks** to a project when team members leave.
- Pair programming is **not necessarily inefficient** and there is some evidence that suggests that a pair working together is more efficient than 2 programmers working separately.

Agile Project Management

Agile Project Management

- The principal responsibility of software project managers is to *manage the project so that the software is delivered on time and within the planned budget for the project.*
- The standard approach to project management is plan-driven. Managers draw up a plan for the project showing **what** should be delivered, **when** it should be delivered and **who** will work on the development of the project deliverables.
- Agile project management requires a different approach, which is adapted to incremental development and the practices used in agile methods.

Scrum

- Scrum is an *agile method that focuses on managing iterative development rather than specific agile practices*.
- There are three phases in Scrum.
  - The **initial phase** is an outline planning phase where you establish the general objectives for the project and design the software architecture.
  - This is followed by a series of **sprint cycles**, where each cycle develops an increment of the system.
  - The **project closure phase** wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.

SCRUM

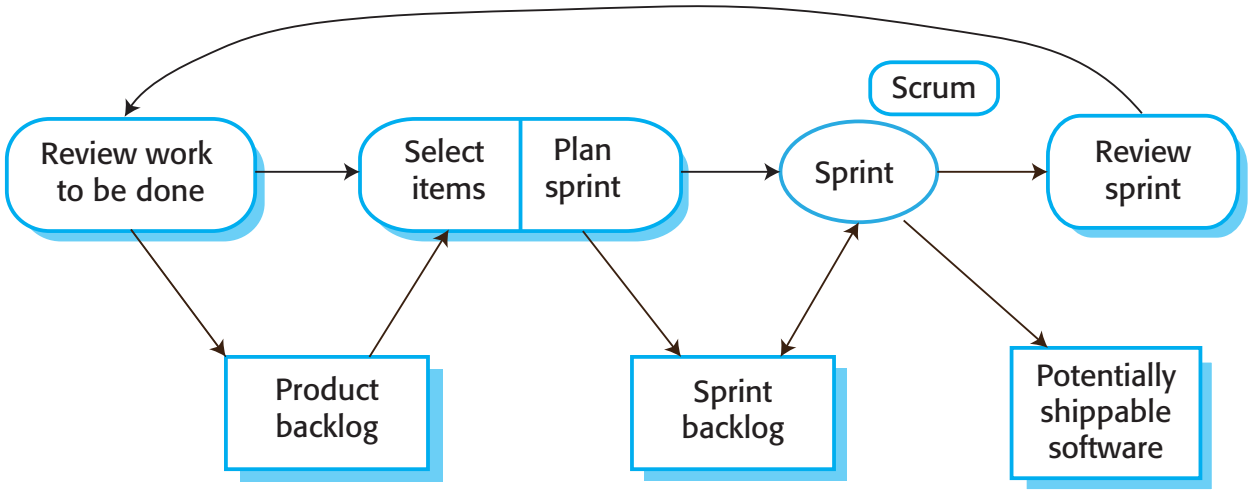
A framework for managing work with an emphasis on software development. It is designed for teams of developers (3 to 9) who break their work into actions that can be completed within timeboxed iterations, called sprints (30 days or less, most commonly two weeks) and track progress and re-plan in 15-minute stand-up meetings, called daily scrums.

Scrum Terminology

Scrum term	Definition
Development team	A self-organizing group of software developers responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be '**potentially shippable**' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

The Scrum Sprint Cycle

- Sprints are fixed length, normally 2–4 weeks.
- The starting point for planning is the **product backlog**, which is the list of work to be done on the project.
- The selection phase involves all of the project team who work with the customer to select the features and functionality from the product backlog to be developed during the sprint.



The Sprint Cycle

- Once these are agreed, the team organize themselves to develop the software.
- During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called ‘Scrum master’.
- The role of the Scrum master is to protect the development team from external distractions.
- At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

Teamwork in Scrum

- The ‘Scrum master’ is a **facilitator** who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- The whole team attends short daily meetings (Scrums) where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
  - This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

Scrum Benefits

- The product is **broken down into a set of manageable and understandable chunks**.
- Unstable requirements do not hold up progress.
- The whole team have **visibility** of everything and consequently team **communication** is improved.
- Customers see **on-time delivery of increments** and gain **feedback** on how the product works.
- **Trust** between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Practical Problems with Agile Methods

- The informality of agile development is incompatible with the **legal approach to contract definition** that is commonly used in large companies.
- Agile methods are **most appropriate for new software development rather than software maintenance**. Yet the majority of software costs in large companies come from maintaining their existing software systems.
- Agile methods are designed for **small co-located teams** yet much software development now involves worldwide distributed teams.

Contractual Issues

- Most software contracts for custom systems are based around a specification, which sets out what has to be implemented by the system developer for the system customer.
- However, this precludes interleaving specification and development as is the norm in agile development.
- A contract that pays for developer time rather than functionality is required.
  - However, this is seen as a high risk in many legal departments because what has to be delivered cannot be guaranteed.

Agile Methods and Software Maintenance

- Key problems are:
  - Lack of product documentation
  - Keeping customers involved in the development process
  - Maintaining the continuity of the development team
- Agile development relies on the development team knowing and understanding what has to be done.
- For long-lifetime systems, this is a real problem as the original developers will not always work on the system.

Agile and Plan-Driven Methods

- Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:
  - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
  - Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
  - How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

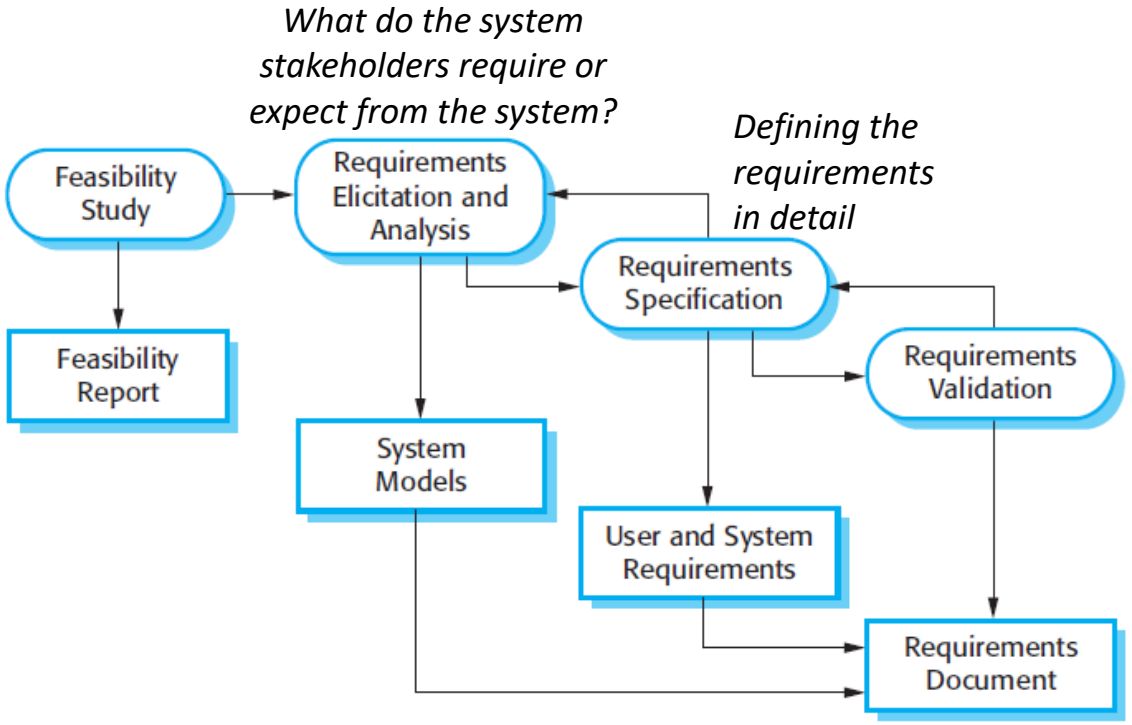
Process Activities

Process Activities

- Real software processes are inter-leaved sequences of *technical*, collaborative and *managerial* activities with the overall goal of specifying, designing, implementing and testing a software system.
- The four basic process activities of **specification**, **development**, **validation** and **evolution** are organized differently in different development processes.
- For example, in the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved.
- Generally, **processes are now tool-supported**. This means that software developers may use a range of software tools to help them, such as requirements management systems, design model editors, program editors, automated testing tools, and debuggers.

Software Specification

- *The process of establishing what services are required and the constraints on the system's operation and development.*
- Requirements engineering process
  - *Requirements elicitation and analysis*
    - What do the system stakeholders require or expect from the system?
  - *Requirements specification*
    - Defining the requirements in detail
  - *Requirements validation*
    - Checking the validity of the requirements

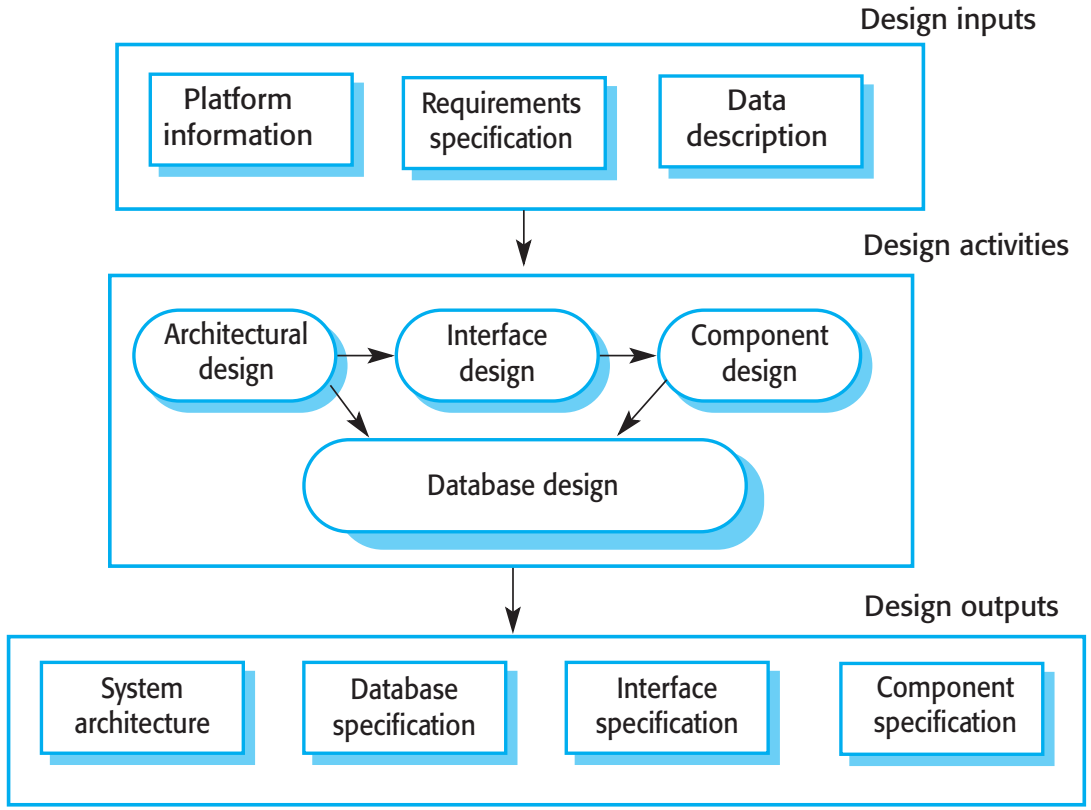


Software Design and Implementation

- *The process of converting the system specification into an executable system.*
- Software design
  - Design a software structure that realises the specification;
- Implementation
  - Translate this structure into an executable program;
- The activities of design and implementation are **closely related** and may be **inter-leaved**.

Design Activities

- *Architectural design,* where you identify the overall structure of the system, the principal components (subsystems or modules), their relationships and how they are distributed.
- *Database design,* where you design the system data structures and how these are to be represented in a database.
- *Interface design,* where you define the interfaces between system components.
- *Component selection and design,* where you search for reusable components. If unavailable, you design how it will operate.



*The design process activities are both **interleaved** and **interdependent**. New information about the design is constantly being generated, and this affects previous design decisions. **Design rework is therefore inevitable**.*

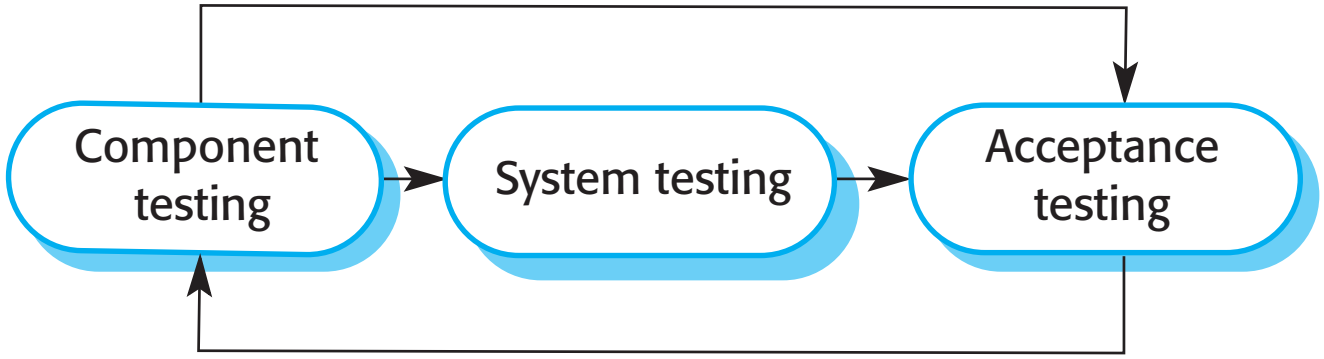
System Implementation

- The software is implemented either by *developing a program* or programs or by *configuring an application system*.
- Software development tools may be used to generate a skeleton program from a design.
- Design and implementation are **interleaved** activities for most types of software system.
- Programming is an individual activity with no standard process.
- Normally, programmers carry out some testing of the code they have developed. This often reveals program defects (bugs) that must be removed from the program. Finding and fixing program defects is called **debugging**.

Software Validation

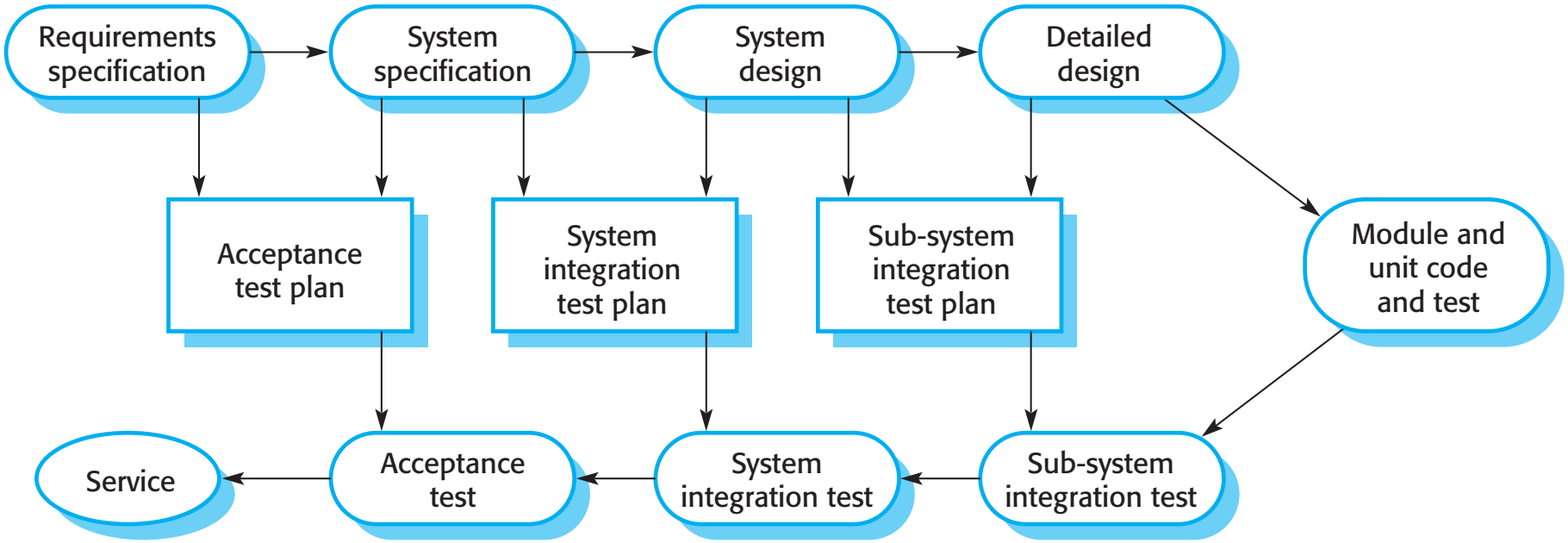
- *Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.*
- Involves **checking** and **review processes** and **system testing**.
- System testing involves *executing the system with test cases* that are derived from the specification of the real data to be processed by the system.
- Testing is the most commonly used V & V activity.

Testing Stages



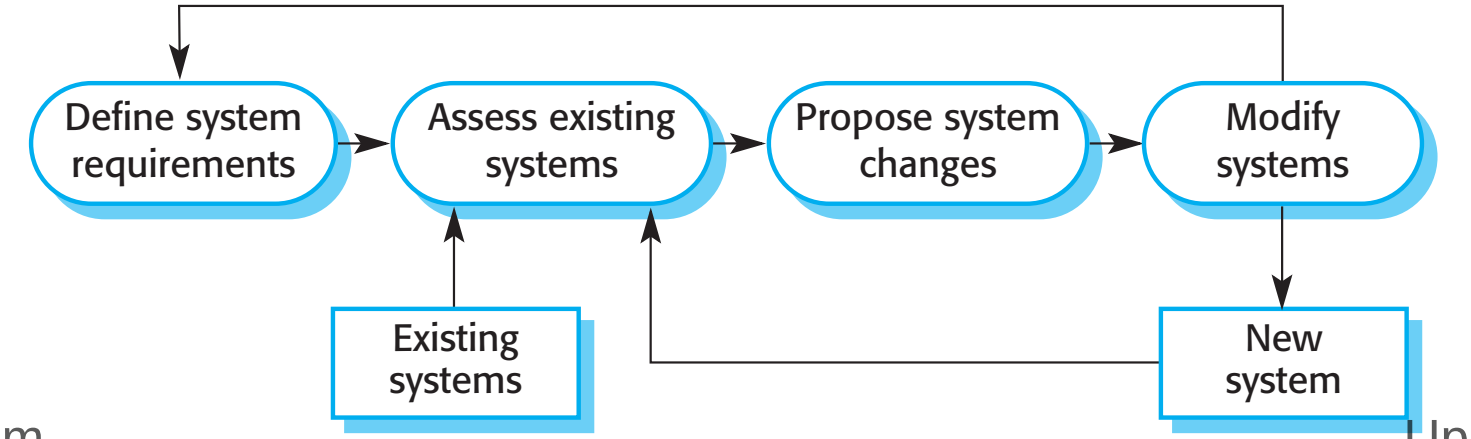
- **Component testing**
  - Individual components are tested independently;
  - Components may be functions or objects or coherent groupings of these entities.
- **System testing**
  - Testing of the system as a whole. Testing of **emergent properties** is particularly important.
- **Customer testing**
  - Testing with customer data to check that the system meets the customer's needs.

Testing Phases in a Plan-Driven Software Process (V-model)



Software Evolution

- Software is inherently flexible and can change.
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.



Coping with Change

Coping with Change

- Change is **inevitable** in all large software projects.
  - Business changes lead to new and changed system requirements
  - New technologies open up new possibilities for improving implementations
  - Changing platforms require application changes
- Change leads to rework so the costs of change include both **rework** (e.g., re-analyzing requirements) as well as the costs **of implementing new functionality.**

Reducing the Costs of Rework

- **Change anticipation**, where the software process includes activities that can anticipate possible changes before significant rework is required.
  - For example, a prototype system may be developed to show some key features of the system to customers.
- **Change tolerance**, where the process is designed so that changes can be accommodated at relatively low cost.
  - This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have be altered to incorporate the change.

Coping with Changing Requirements

- **System prototyping**, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions. This approach supports change anticipation.
- **Incremental delivery**, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance.

Software Prototyping

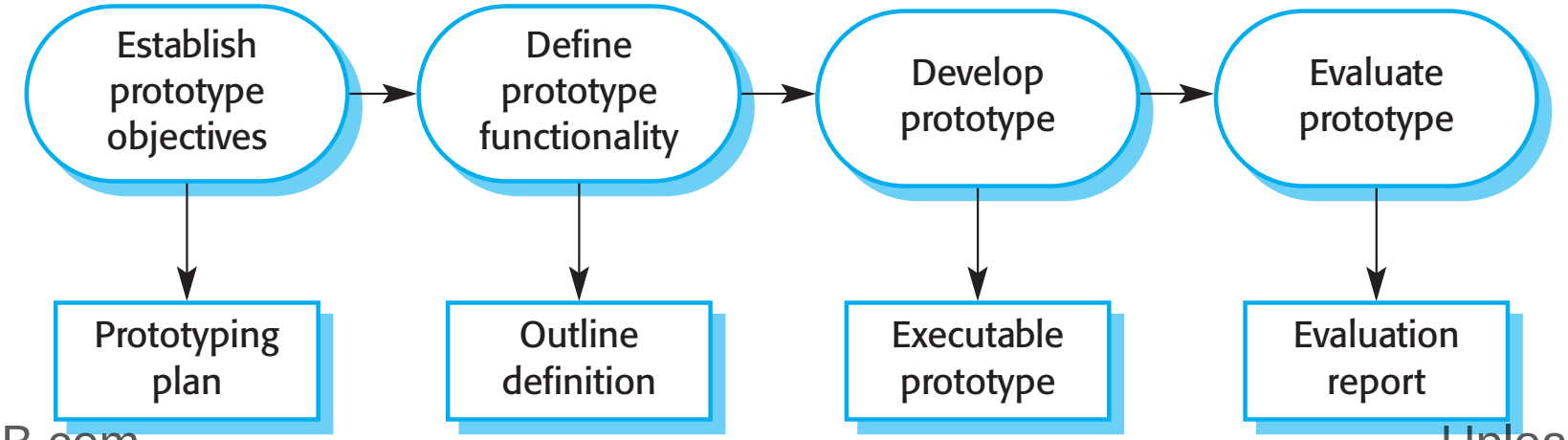
- A prototype is an ***initial*** *version of a system used to demonstrate concepts and try out design options.*
- A prototype can be used in:
  - The requirements engineering process to help with requirements elicitation and validation;
  - In design processes to explore options and develop a UI design;
  - In the testing process to run back-to-back tests.

Benefits of Prototyping

- Improved system usability.
- A closer match to users' real needs.
- Improved design quality.
- Improved maintainability.
- Reduced development effort.

Prototype Development

- May be based on rapid prototyping languages or tools
- May involve leaving out functionality
  - Prototype should focus on areas of the product that are not well-understood;
  - Error checking and recovery may not be included in the prototype;
  - Focus on functional rather than non-functional requirements such as reliability and security.



Throw-Away Prototypes

- Prototypes should be **discarded** after development as they are not a good basis for a production system:
  - It may be impossible to tune the system to meet non-functional requirements;
  - Prototypes are normally undocumented;
  - The prototype structure is usually degraded through rapid change;
  - The prototype probably will not meet normal organizational quality standards.

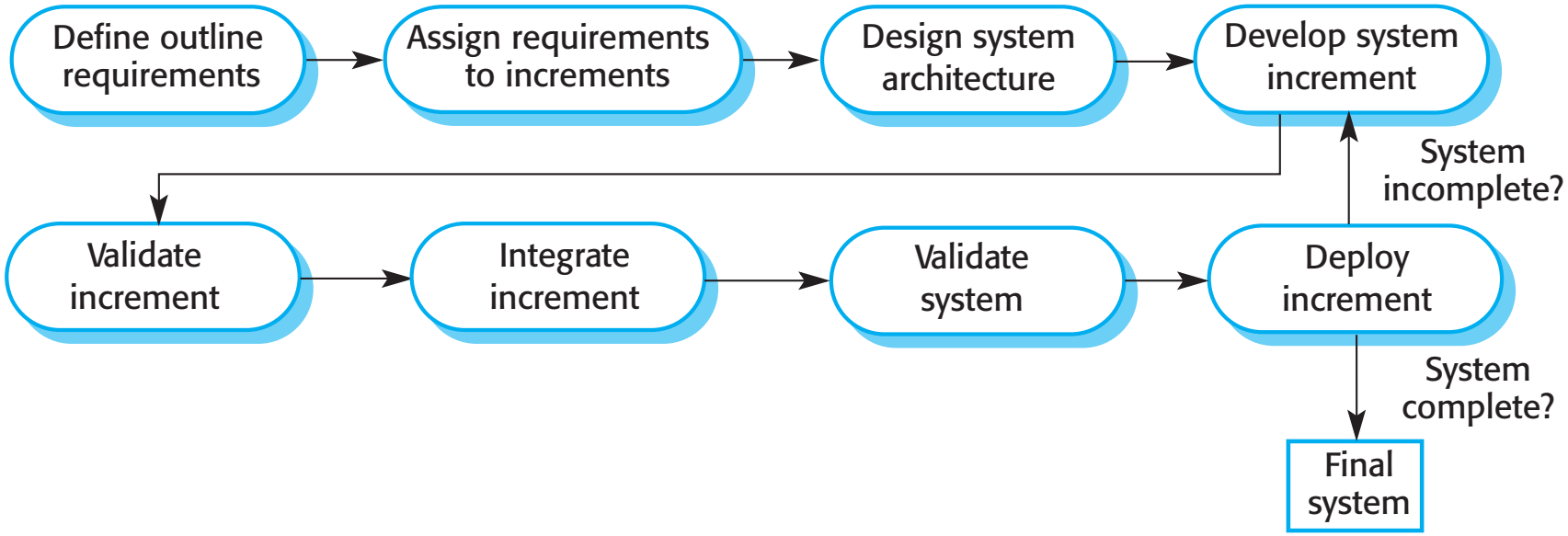
Incremental Delivery

- Rather than deliver the system as a single delivery, the development and delivery is broken down into **increments** with each increment delivering part of the required functionality.
- User requirements are **prioritised** and the highest priority requirements are included in early increments.
- Once the development of an increment is started, the requirements are **frozen** though requirements for later increments can continue to evolve.

Incremental Development and Delivery

- **Incremental development**
  - Develop the system in increments and evaluate each increment before proceeding to the development of the next increment;
  - Normal approach used in agile methods;
  - Evaluation done by user/customer proxy.
- **Incremental delivery**
  - Deploy an increment for use by end-users;
  - More realistic evaluation about practical use of software;
  - Difficult to implement for replacement systems as increments have less functionality than the system being replaced.

Incremental Delivery



Incremental Delivery Advantages

- Customer value can be delivered with each increment so system functionality is available earlier.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- The highest priority system services tend to receive the most testing.

Incremental Delivery Problems

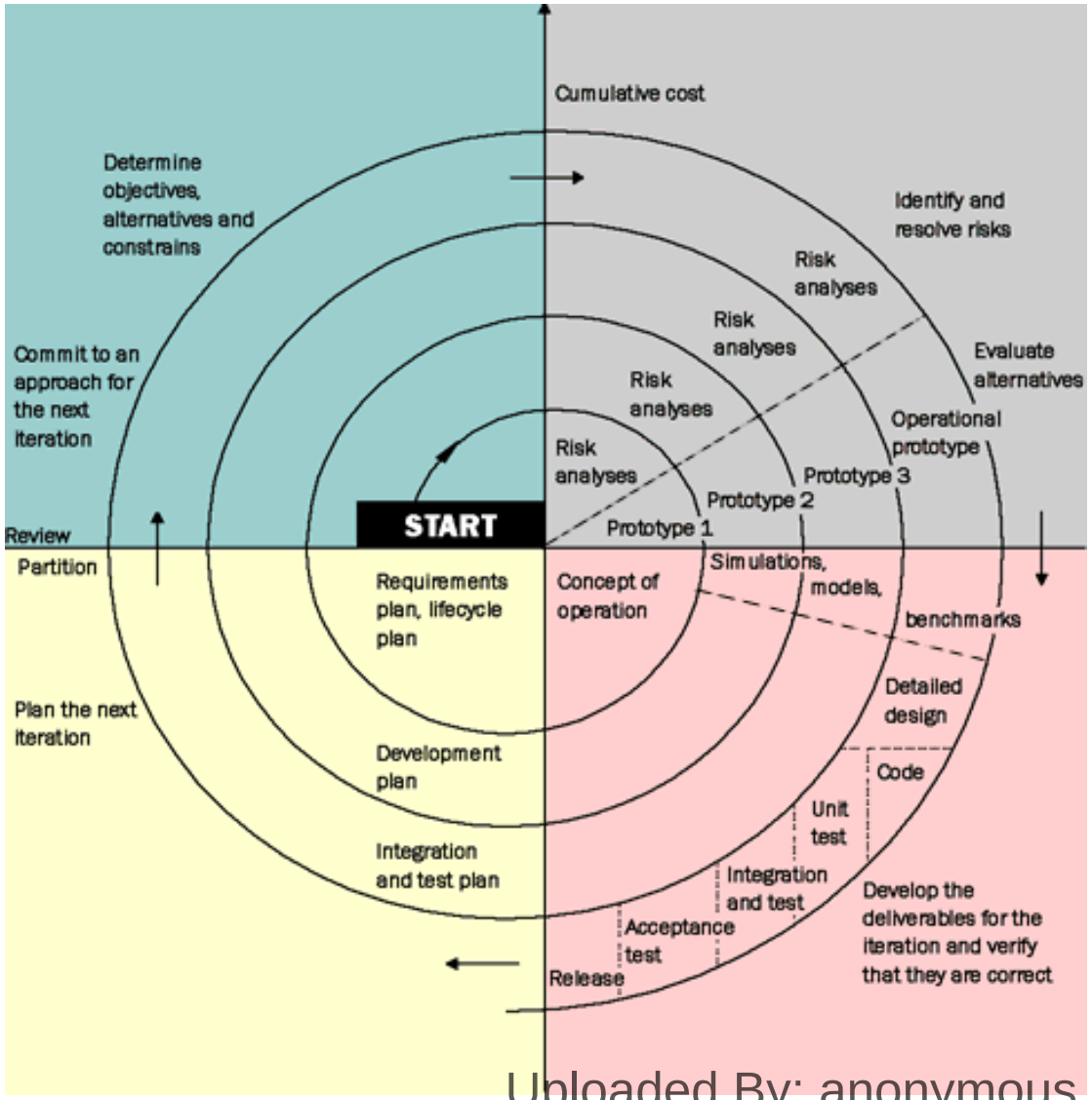
- Most systems require a set of basic facilities that are used by different parts of the system.
  - As requirements are not defined in detail until an increment is to be implemented, **it can be hard to identify common facilities that are needed by all increments**.
- The essence of iterative processes is that the specification is developed in conjunction with the software.
  - However, this **conflicts with the procurement model of many organizations**, where the complete system specification is part of the system development contract.

Spiral Model of Software Development

- The spiral model was defined by Barry Boehm in his article A Spiral Model of Software Development and Enhancement from 1986.
- This model was not the first model to discuss iteration, but it was the first model to explain why the iteration matters. As originally envisioned, the iterations were typically 6 months to 2 years long.
- The spiral model (Boehm, 1988) aims at **risk reduction** by any means in any phase. The spiral model is often referred to as a **risk-driven model**.
- Introducing **prototyping** in a Software Process aims at risk reduction at the *requirements level*. There is always an element of risk involved in the other phases of development.

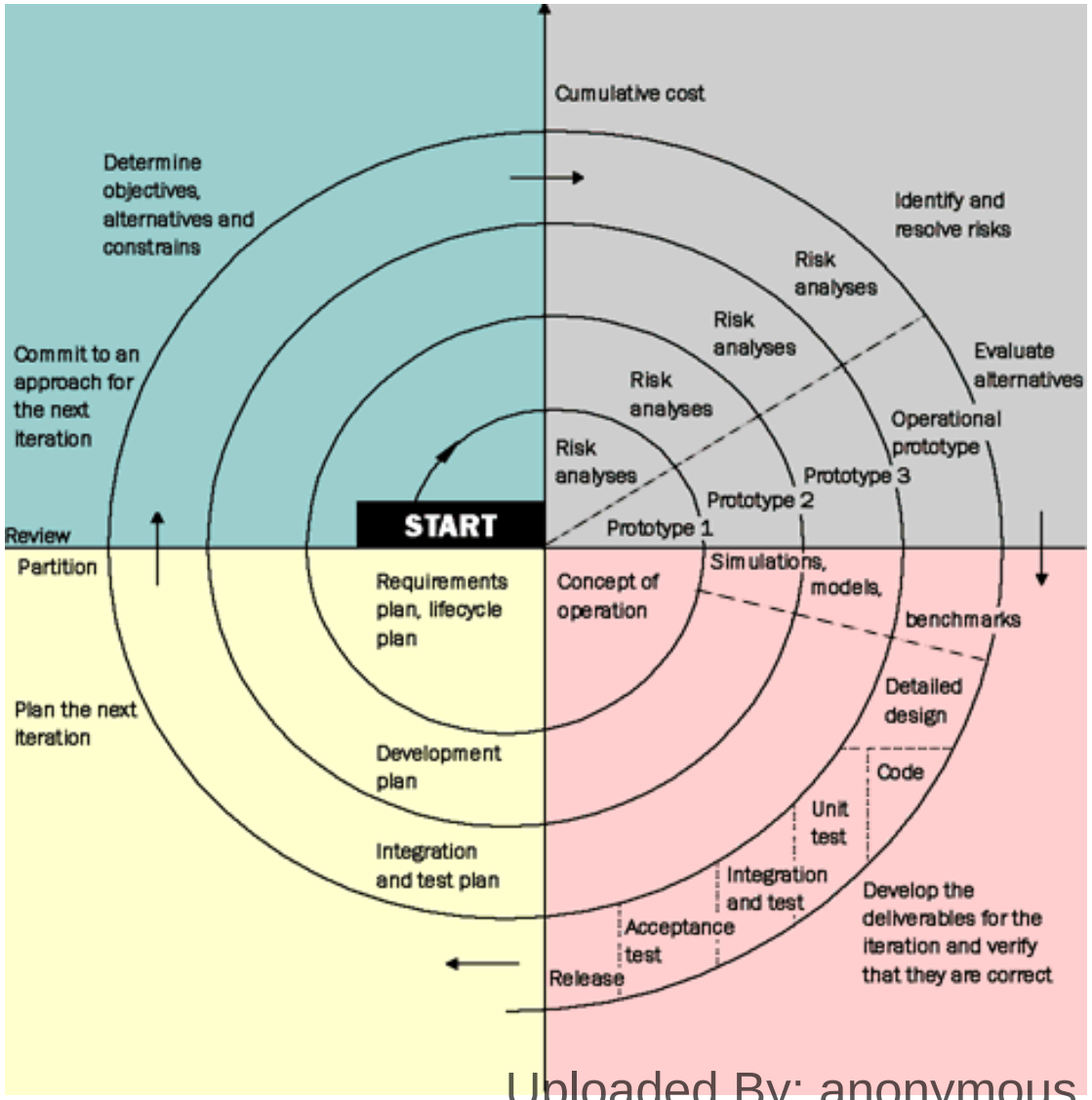
Spiral Model of Software Development

- A spiral phase begins in the top left quadrant (**quadrant 1**), by determining objectives of that phase, alternatives and constraints. This is a way to define a strategy for achieving the goals of this iteration.
- Next (**quadrant 2**), the strategy is analyzed form the viewpoint of risk, and solutions to minimize these risks are investigated, often using **prototyping**.



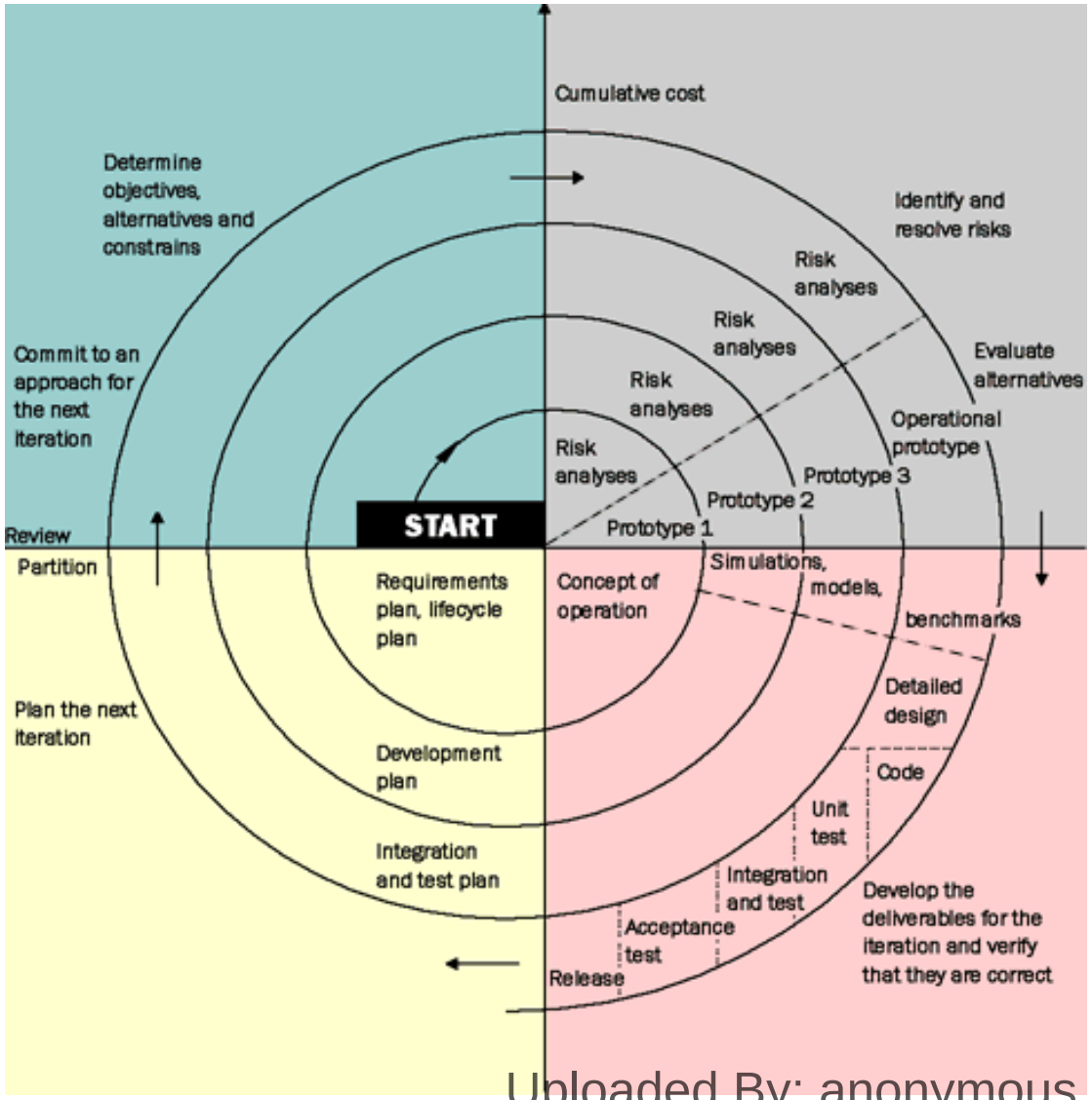
Spiral Model of Software Development

- Then (**quadrant 3**), considering the investigations made in **quadrant 2**, a solution is put into practice to produce the artifacts necessary to reach the goals identified in **quadrant 1**. This **quadrant 3** corresponds to where the traditional waterfall model phases are put into practice.



Spiral Model of Software Development

- Finally (**quadrant 4**), the results of the risk-reduction strategies are assessed, and if all risks are resolved, the next phase is planned and started.
- If some risks are left unsolved, another iteration can be made to continue to work on the uneliminated risks. If certain risks can not be resolved, the project might be **terminated immediately**.



Spiral Model of Software Development

Advantages

- Emphasis on alternatives and constraints supports the reuse of existing solutions.
- Targets testing by treating it as a risk, which has to be addressed.
- Maintenance is just another phase of the spiral model. It is treated in the same way as development.
- Estimates (budget and schedule) get more realistic as work progresses, because important issues are discovered earlier.
- It is more able to cope with the (nearly inevitable) changes that software development generally entails.
- Software engineers, who can get restless with protracted design processes, can get their hands in and start working on a project earlier.

Spiral Model of Software Development

Disadvantages

- Only intended for internal projects (inside a company), because risk is assessed as the project is developed. Hardly suitable for contractual software development.
- In case of contractual software development, all risk analysis must be performed by both client and developers before the contract is signed and not as in the spiral model.
- Spiral model is risk driven. Therefore, it requires knowledgeable staff.
- Suitable for only large-scale software development. Does not make sense if the cost of risk analysis is a major part of the overall project cost.

In-class Activity

Software Development Processes

Case 1: *Secure ATM System*

To develop a **secure ATM sub-system** and integrate it with an existing *banking system*. The developed ATM sub-system will be deployed across a 1000 ATM machines. It should have an **availability rate of 99%**. It should also have a **99.9% accuracy** money notes counting dispenser, and **three-level security** that requires a card, a pin code and a biometric code.

Case 2: *Student Management/Registration*

To develop **university student management/registration system** that can support **75000 students**, and **up-to 15000 concurrent students' access**, would not need more than 1 hour (student/user) **training** and need to be **delivered** in 4 years for operational use.

Case 3: *Health Monitoring Mobile App*

To develop a mobile app, that **monitors health indicators** (e.g., blood pressure, sugar level, and pulse) of patients, by collecting readings through special medical sensors, then provides medical advice based on the collected readings by an external medical decision system, which your system must be connected to it.

Case 4: *Word Processing Application*

To develop a **word processing application**, that uses existing print, graphic, font styles, spelling check, and grammar check components. The application must be designed to be used by people with **dyslexic/learning** difficulty.

Case 5: *Product-Line Ordering System*

- You are a project manger responsible for developing a system for a **product-line ordering system** for a *manufacturer of car parts*.
  - The system, should allow telephone and online ordering of car parts, and has a dedicated team to process the orders.
  - The system should keep inventory of existing stock and be accessed by the manufacturer's product-line to manufacture parts according to sales.
  - The users of the system are online users, who should register an account online and store users' information including their credit card details to enable them ordering online within a secure login sub-system.
  - To enable users order online, the system will be required to connect to the respective credit card bank to authorize payment. Other system users also include salesmen who can place orders through telephone calls, and process payment through the system, and inventory users who manage availability of car parts and system administrators who manage the system database.
  - The system should allow 10 concurrent salesmen and 10 inventory men to use the system.