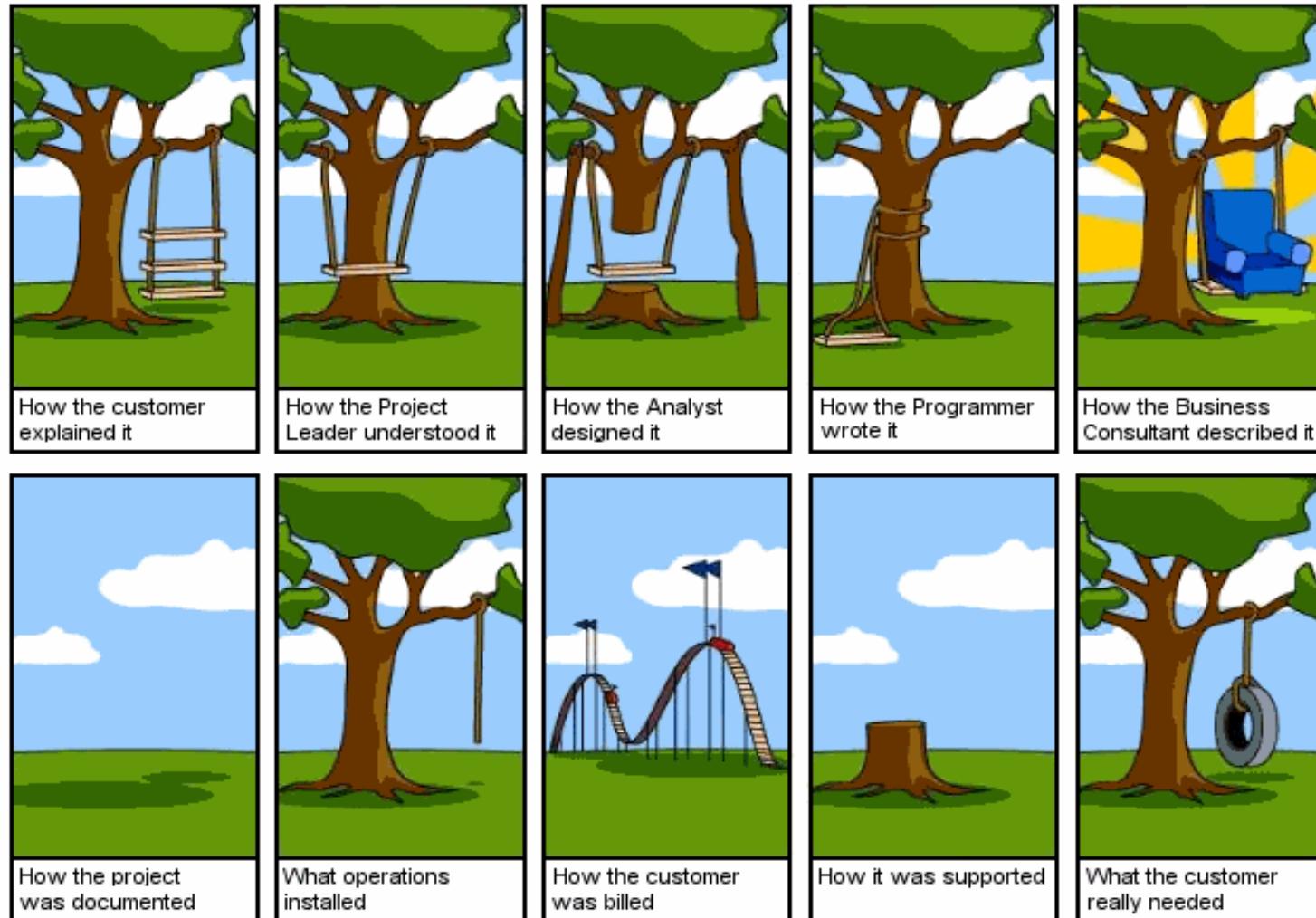

Software Engineering

Introduction to Software Engineering

Course Outline

- To understand the process of designing, building, and maintaining software systems.
- To acquire the skill of software project management.
- To understand software evolution, testing approaches and quality assurance to ensure high standard/professional software

Software Engineering



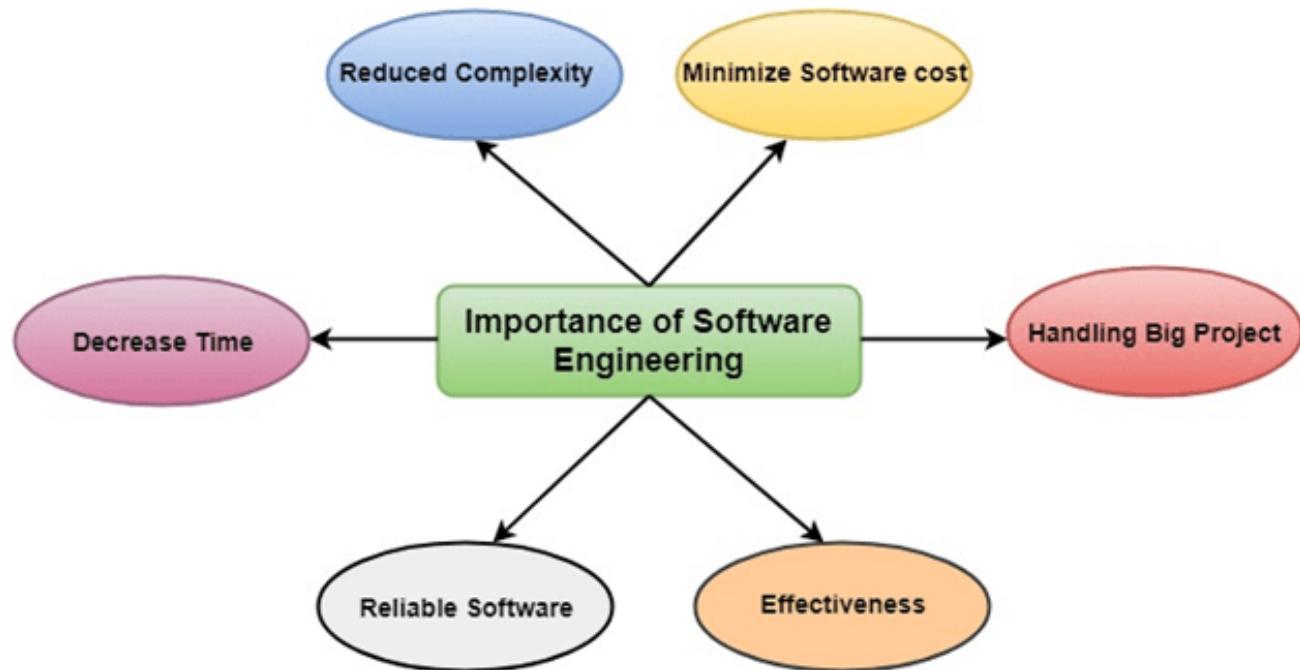
What is Software Engineering

- Engineering discipline that is concerned with all aspects of software production.
 - Design
 - Development
 - Maintenance
- There are various types of software systems
 - embedded systems
 - complex, worldwide information systems
- Different types of software require different approaches
- Every software require software engineering; but not the same technique

Importance of Software Engineering?

The importance of Software engineering is as follows:

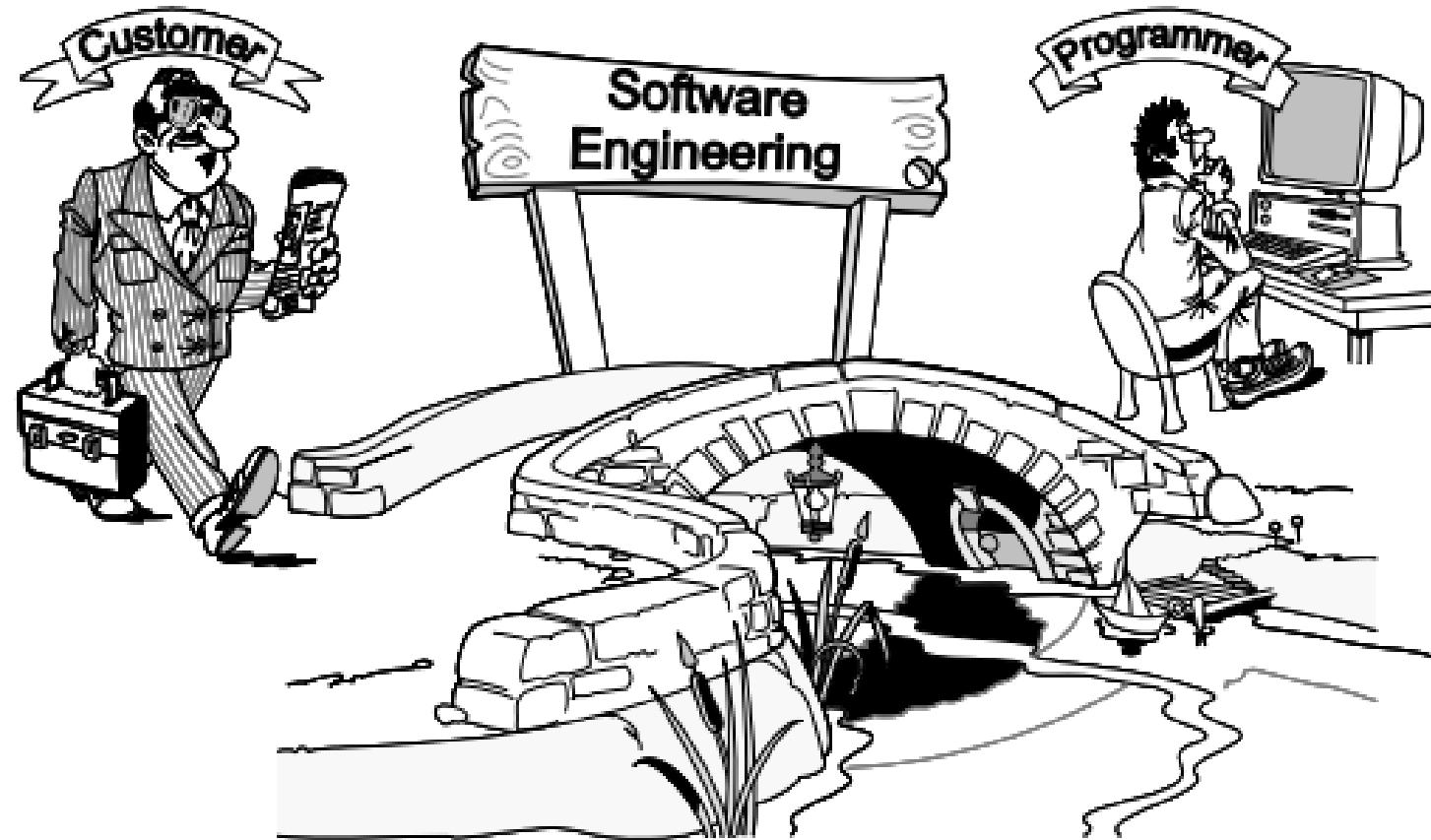
- Reduces complexity
- Minimizes software cost
- Decreases time
- Handles big projects
- Reliable software
- Effectiveness



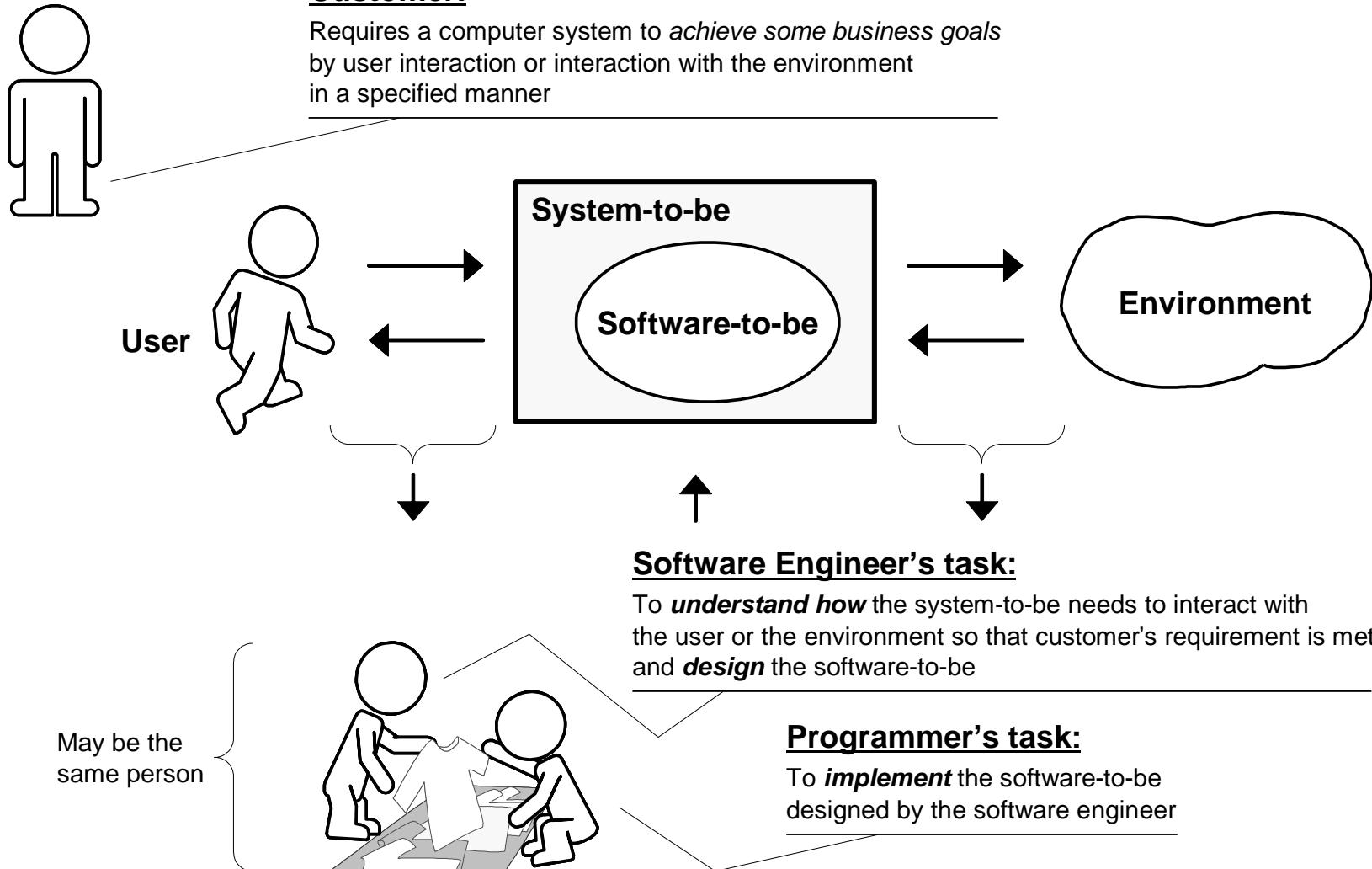
Required Tasks in Software Engineering?

- Software engineering task requires:
 - Ability to **quickly learn** new and diverse disciplines and business processes
 - Ability to **communicate** with domain experts, extract an abstract model of the problem, and formulate a solution.
 - Ability to **design** the software systems for the proposed solution to meet the business needs for many years.
- Software Engineer's task: understand the **customer's business needs** and **design software** accordingly.

The Role of Software Engineering



The Role of Software Engineering



The Role of Software Engineering (SE)

Some Misconceptions:

- Software engineering is all about Programming
 - Software engineering is about writing loads of documentation
 - Software engineering is about writing a running code
-
- SE is the creative activity of understanding the business problems, coming up with an idea for solution, and designing the ‘blueprints’ of the solution.
 - Programming is the ability of implementing the given blueprints
 - SE is helpful to document the process (not the final solution) to know what alternatives were considered and why particular choice were made.
 - SE is about delivering value for the customer, and both code and documentation are valuable.

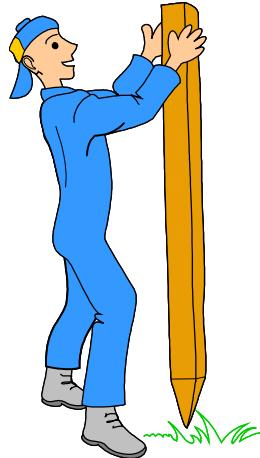
Introduction: Software is Complex

Complex \neq Complicated

- Complex = composed of many simple components related to one another
- Complicated = refers to a high level of difficulty

Complexity Example:

Scheduling Fence Construction Tasks



Setting posts
[3 time units]



Cutting wood
[2 time units]



Nailing
[2 time units for unpainted;
3 time units otherwise]



Painting
[5 time units for uncut wood;
4 time units otherwise]

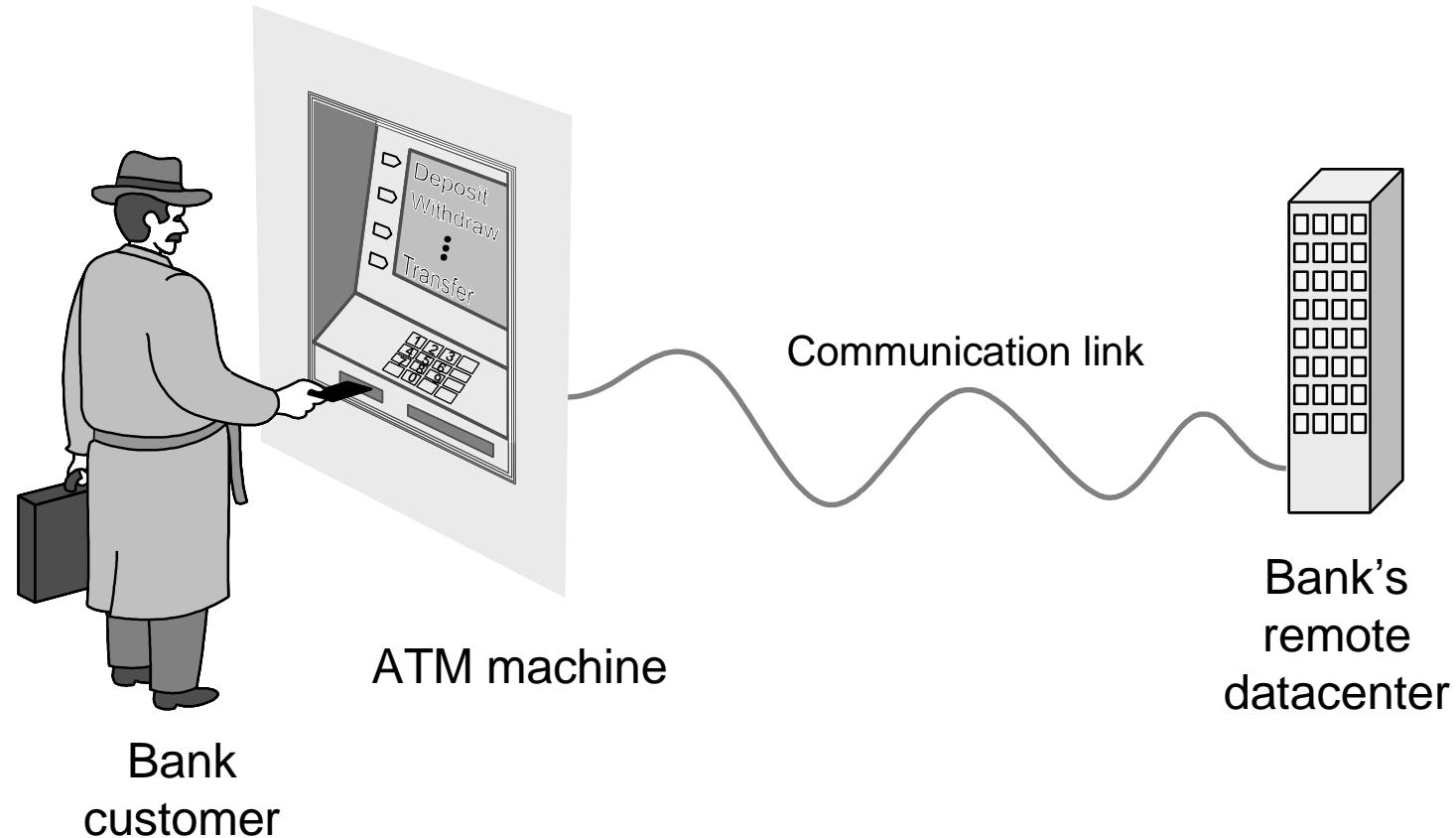
Setting posts < Nailing, Painting

Cutting < Nailing

In what order should these tasks be carried out to complete the project in the shortest possible time?

Example: ATM Machine

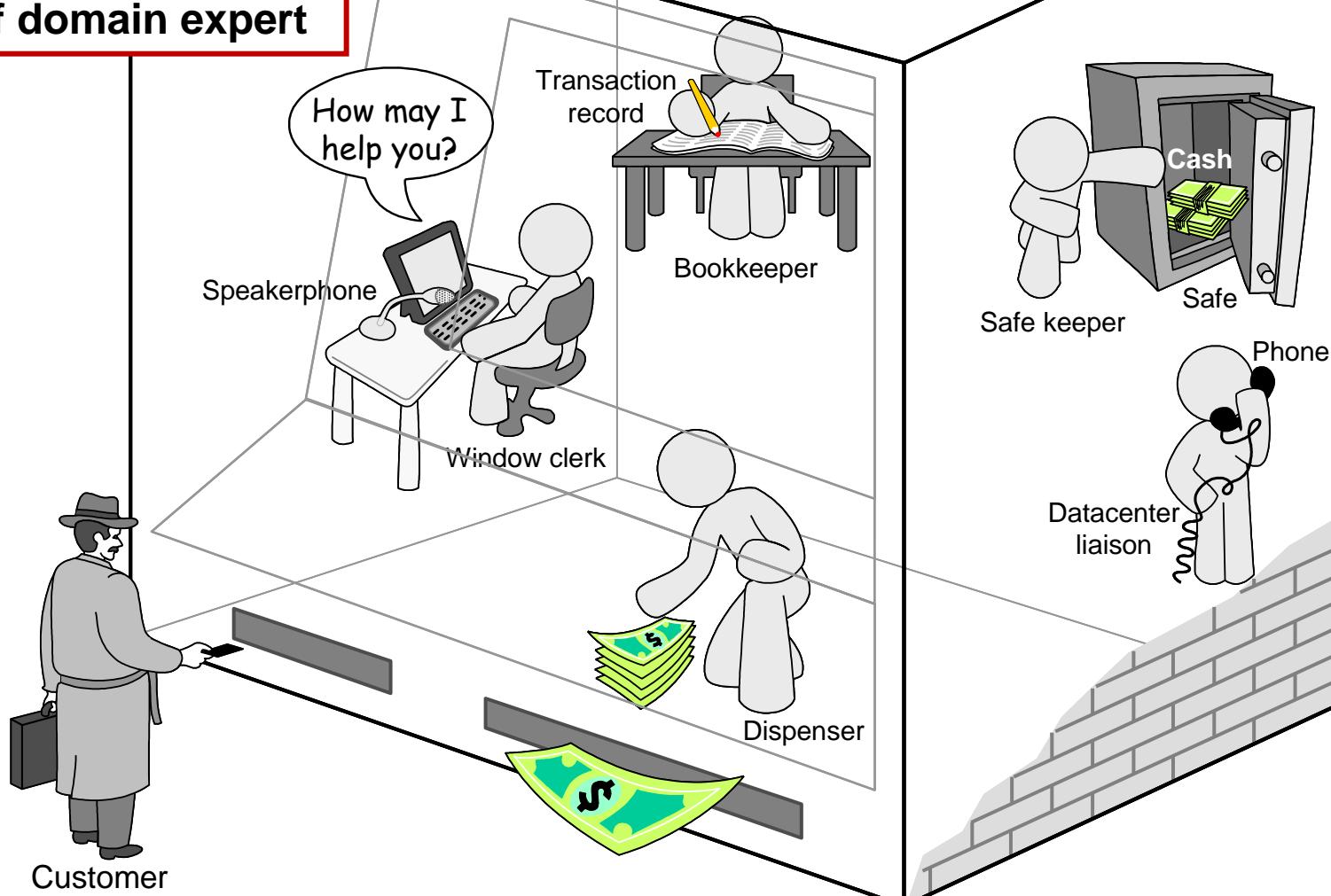
Understanding the money-machine problem:



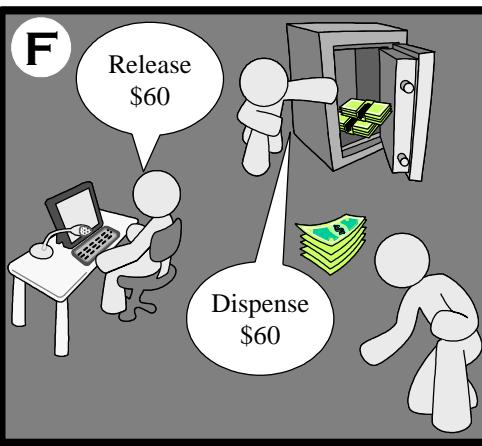
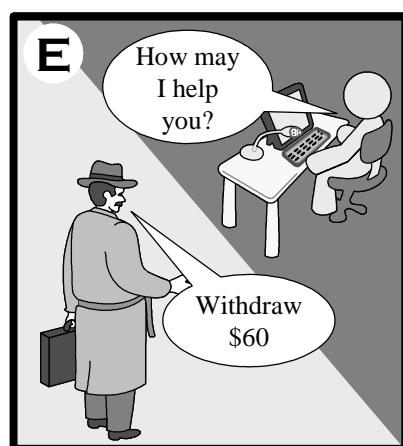
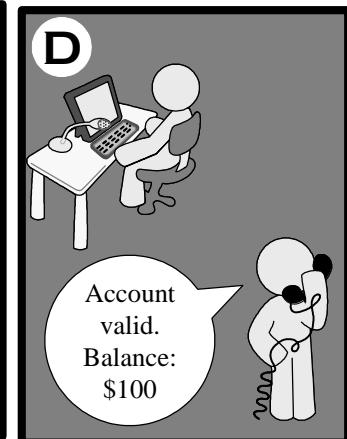
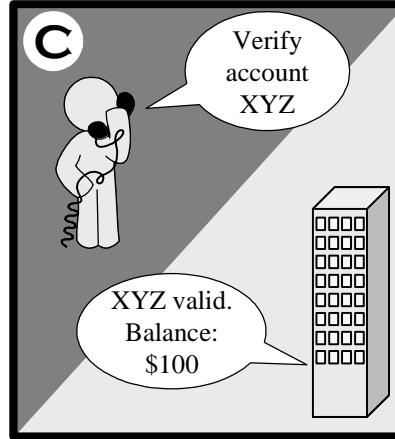
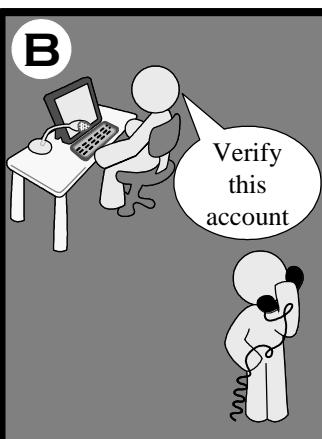
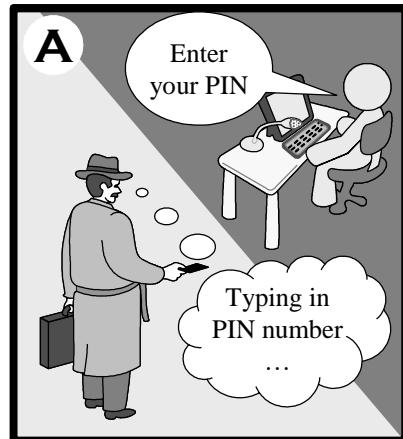
How ATM Machine Might Work

Domain model
created with help
of domain expert

Domain Model



Cartoon Strip: How ATM Machine Works



Software Engineering Lifecycle

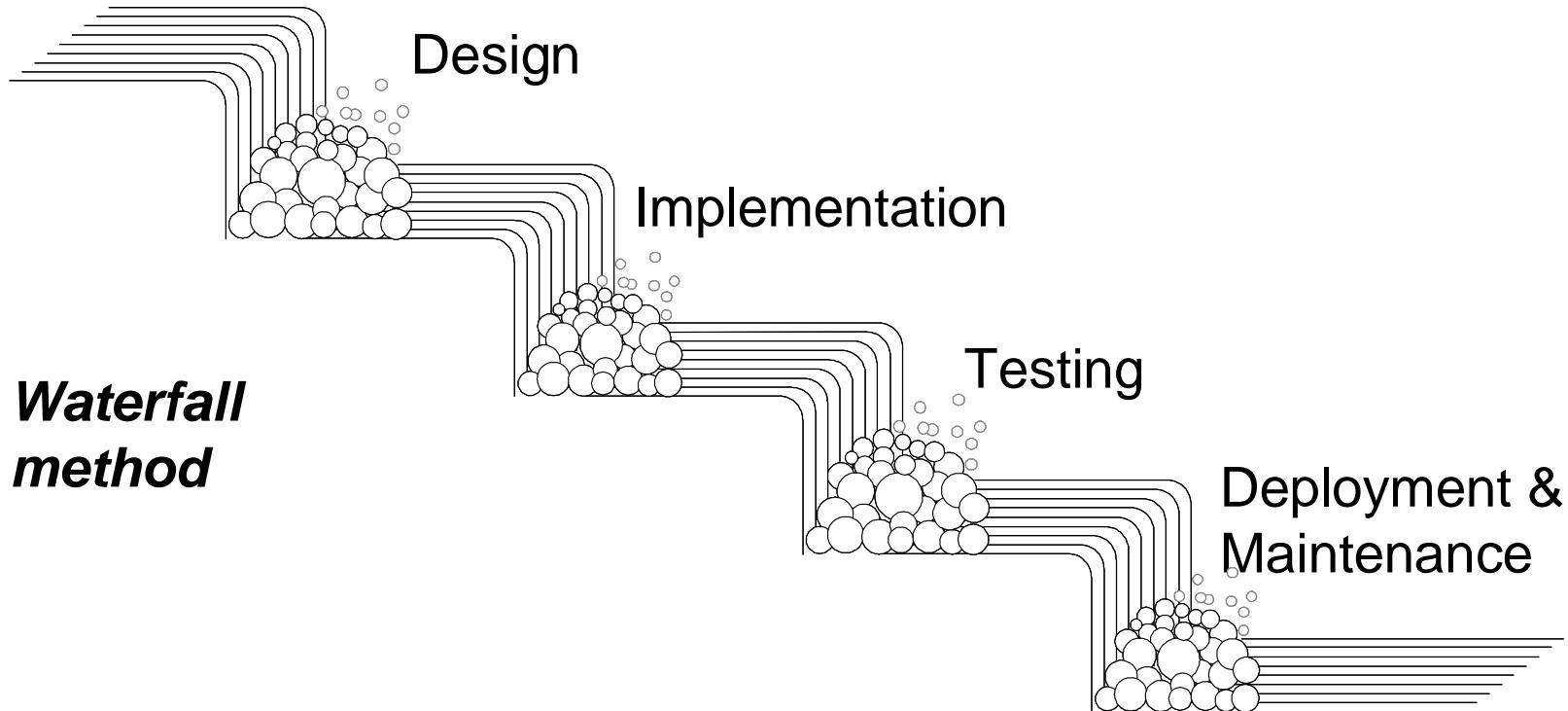
- Any product development process includes the following phases:
 - Planning/specification
 - Design
 - Implementation
 - Evaluation
- The common software development phases are:
 - **Requirement specification** - Understanding the usage scenarios
 - **Design** - Assigning responsibilities to objects
 - **Implementation** - Encoding design in a programming language
 - **Testing** - Individual component (unit) and the entire system (integration)
 - **Operation and Maintenance** - running the systems, fixing bugs, adding new features

Software Development Methods

- **Waterfall** – Activities are proceed in a sequential manner, not practical
- **Iterative Waterfall Model** - allows the mechanism of error correction, not suitable for very large projects
- **Iterative + Incremental (Evolutionary)** - suitable for large projects, decomposed into a set of modules. Develop increment of functionality in each iteration
- **Agile** - an iterative approach, requires customer feedback after each Time-box. The end date of an iteration is fixed

Waterfall Method

Requirements



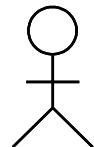
proceeds sequentially
finish this step before moving to the next

Modeling Languages

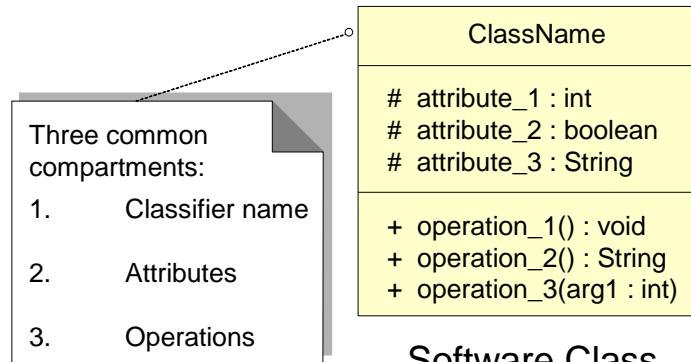
- Modeling is the designing of software applications before coding.
- Modeling is an Essential Part of **large** software projects, and helpful to **medium** and even **small** projects as well.
- There are several modelling languages:
 - UML (Unified Modeling Language)
 - Activity diagram
 - Use case diagram
 - Sequence diagram

UML – Language of Symbols

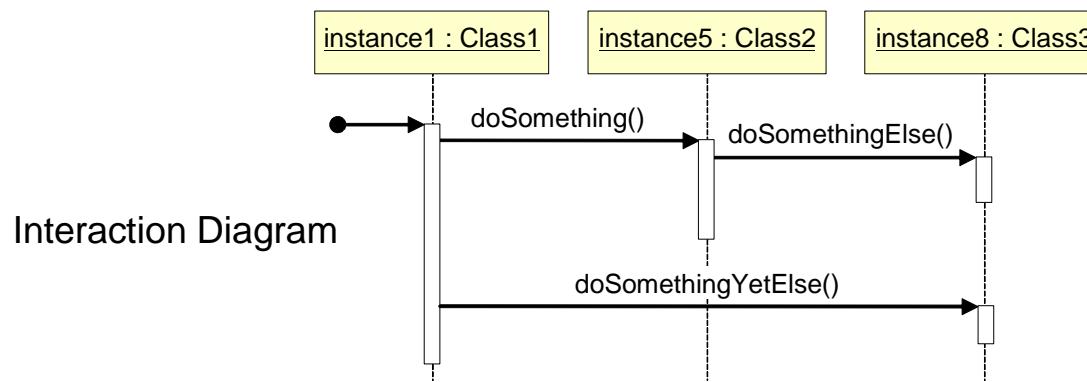
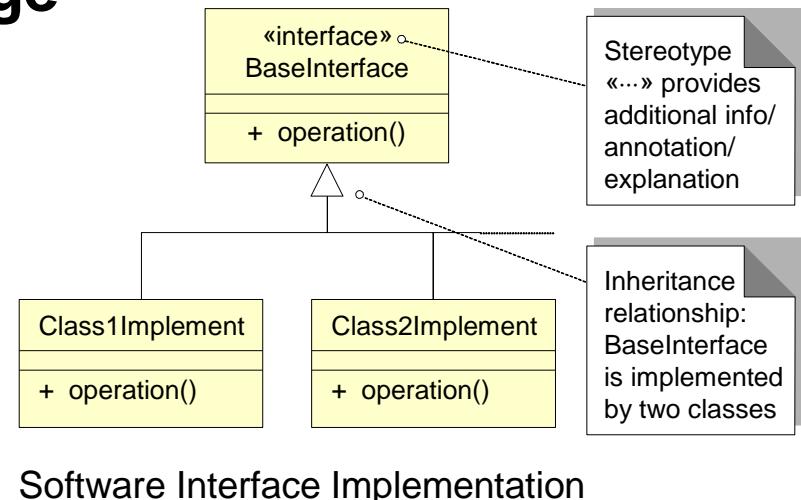
UML = Unified Modeling Language



Actor



Comment

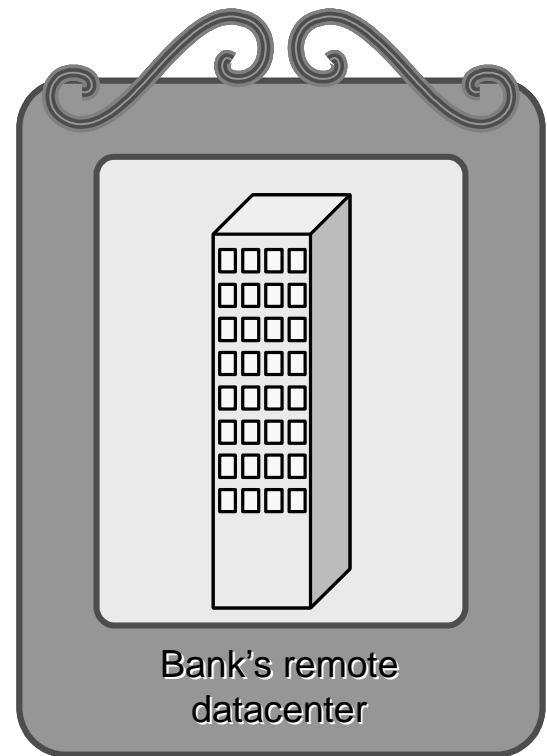
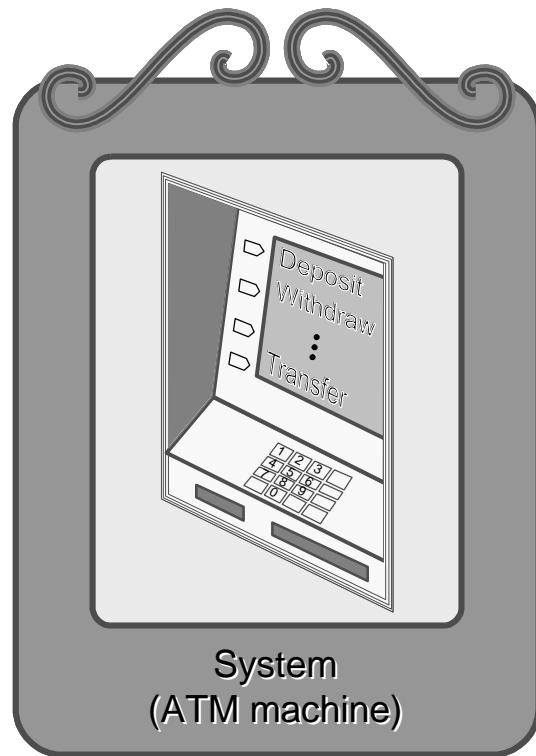
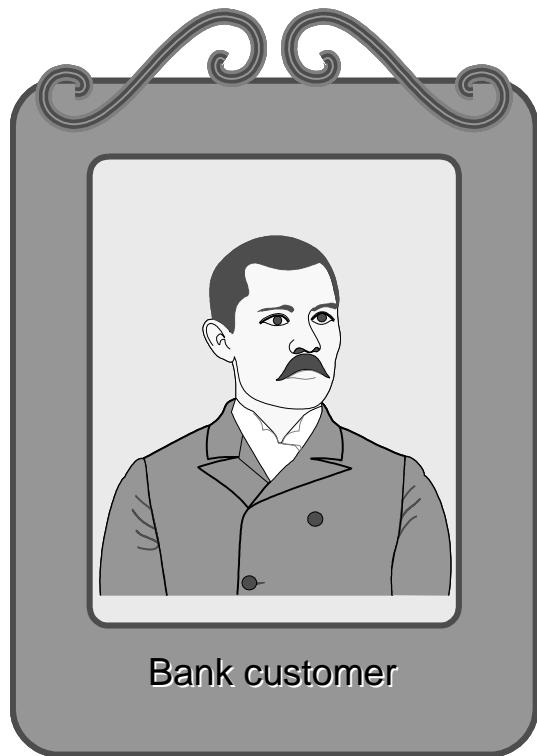


Online information:
<http://www.uml.org>

Requirements Analysis and System Specification

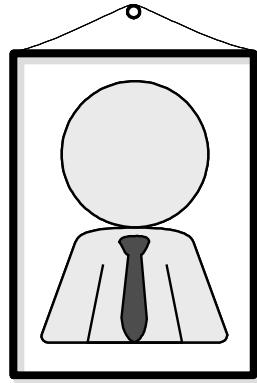
- **Requirement analysis** is the first step of software development process.
- Developer attempts to **understand the problem** and define its **scope**.
- **Goal:** Produce system specification
 - what the system will do
 - how the system will interact with the external players
 - Actors - Agents **external** to the system
 - Concepts/ Objects - Agents working **inside** the system
 - Use Cases – describes the **interaction** between the **System** and **Actors**

ATM: Gallery of Players

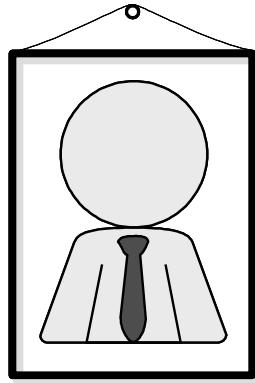


Actors (Easy to identify because they are visible!)

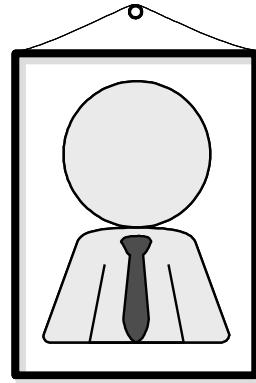
Gallery of Workers + Things



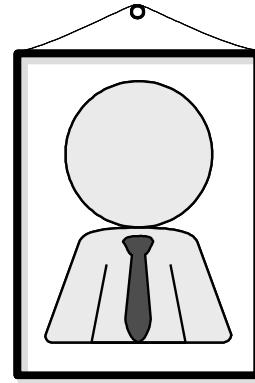
Window clerk



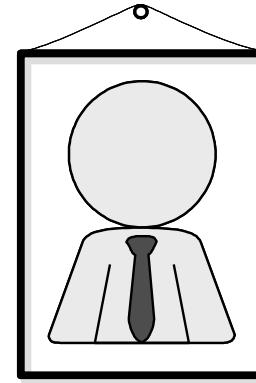
Datacenter
liaison



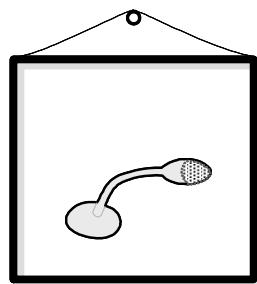
Bookkeeper



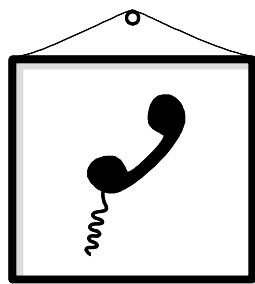
Safe keeper



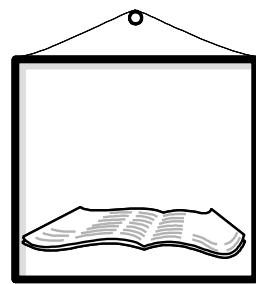
Dispenser



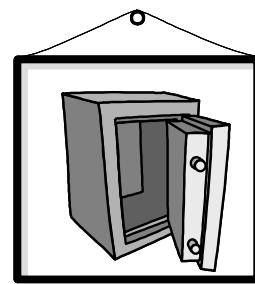
Speakerphone



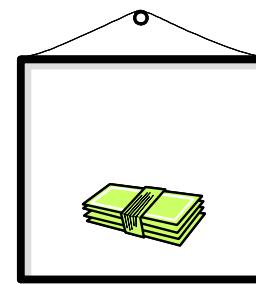
Telephone



Transaction
record



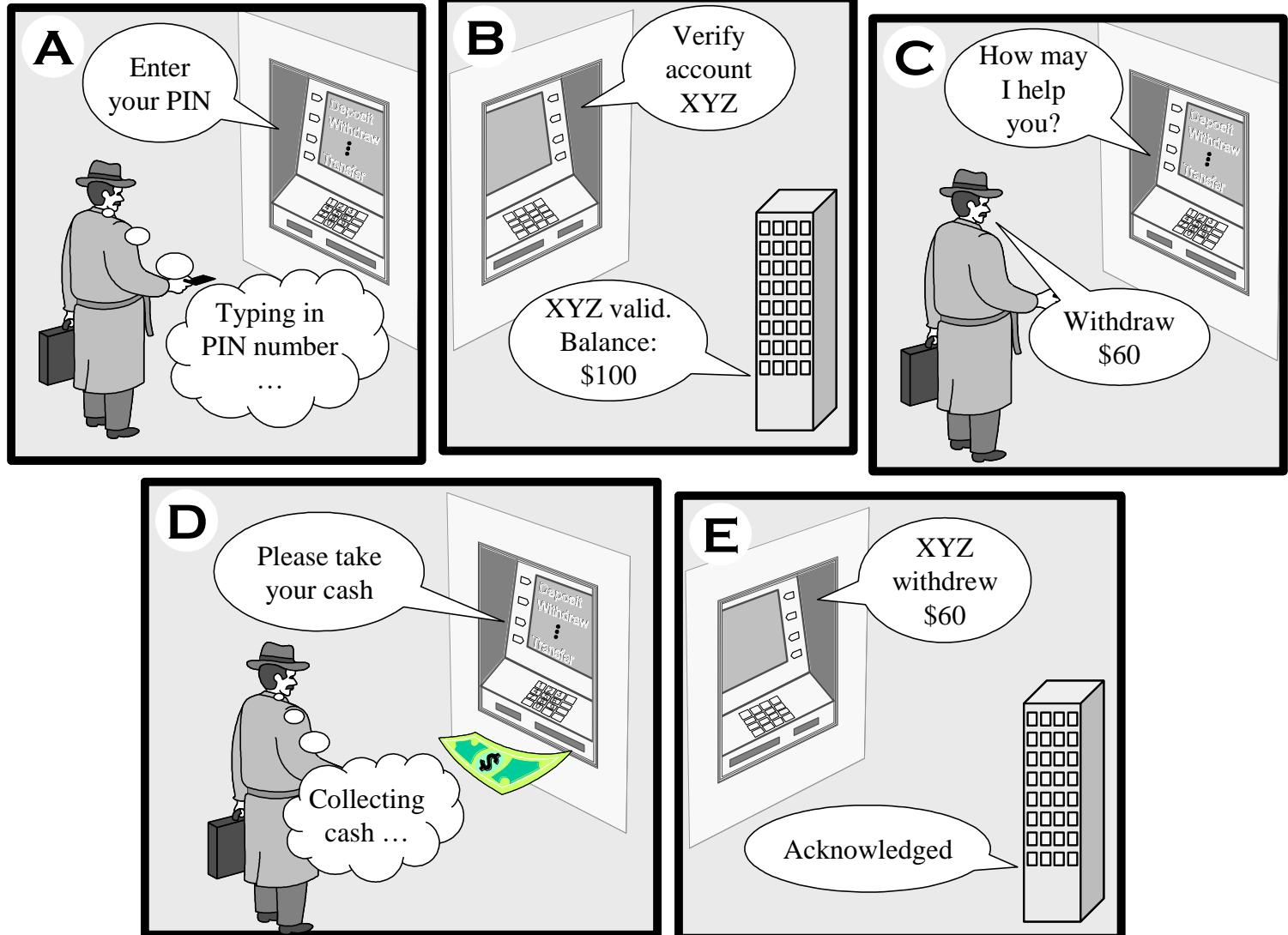
Safe



Cash

Concepts (Hard to identify because they are invisible/imaginary!)

Use Case: Withdraw Cash



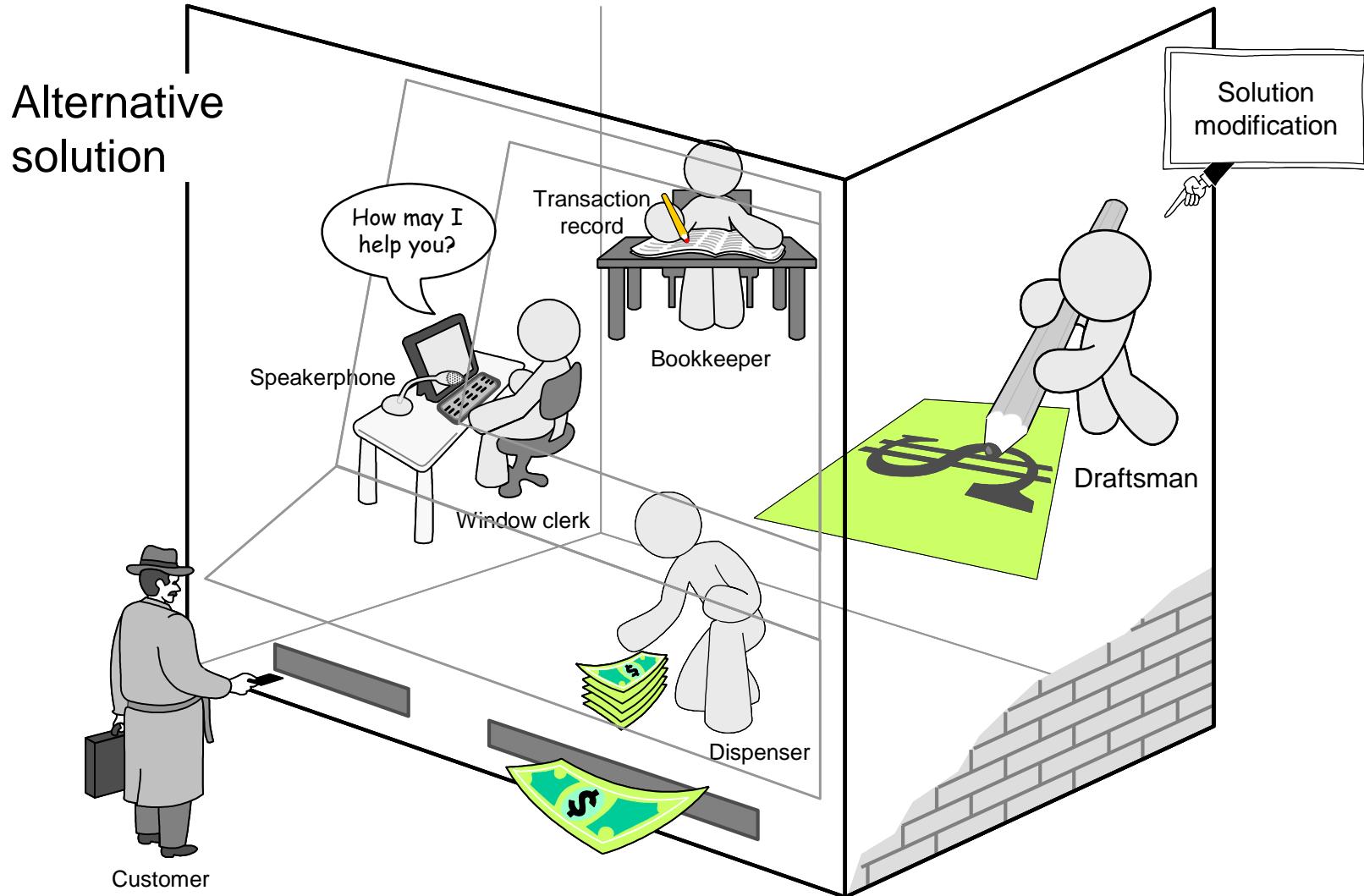
Domain Model

- Models the **inside** of the system.
- Design problems have unlimited number of **alternative solutions**
 - ❖ having a draftsman to draw the banknotes requested by the customer
 - ❖ we imagine having a courier run to a nearest bank depository to retrieve the requested money

Which solution is the best or even feasible?

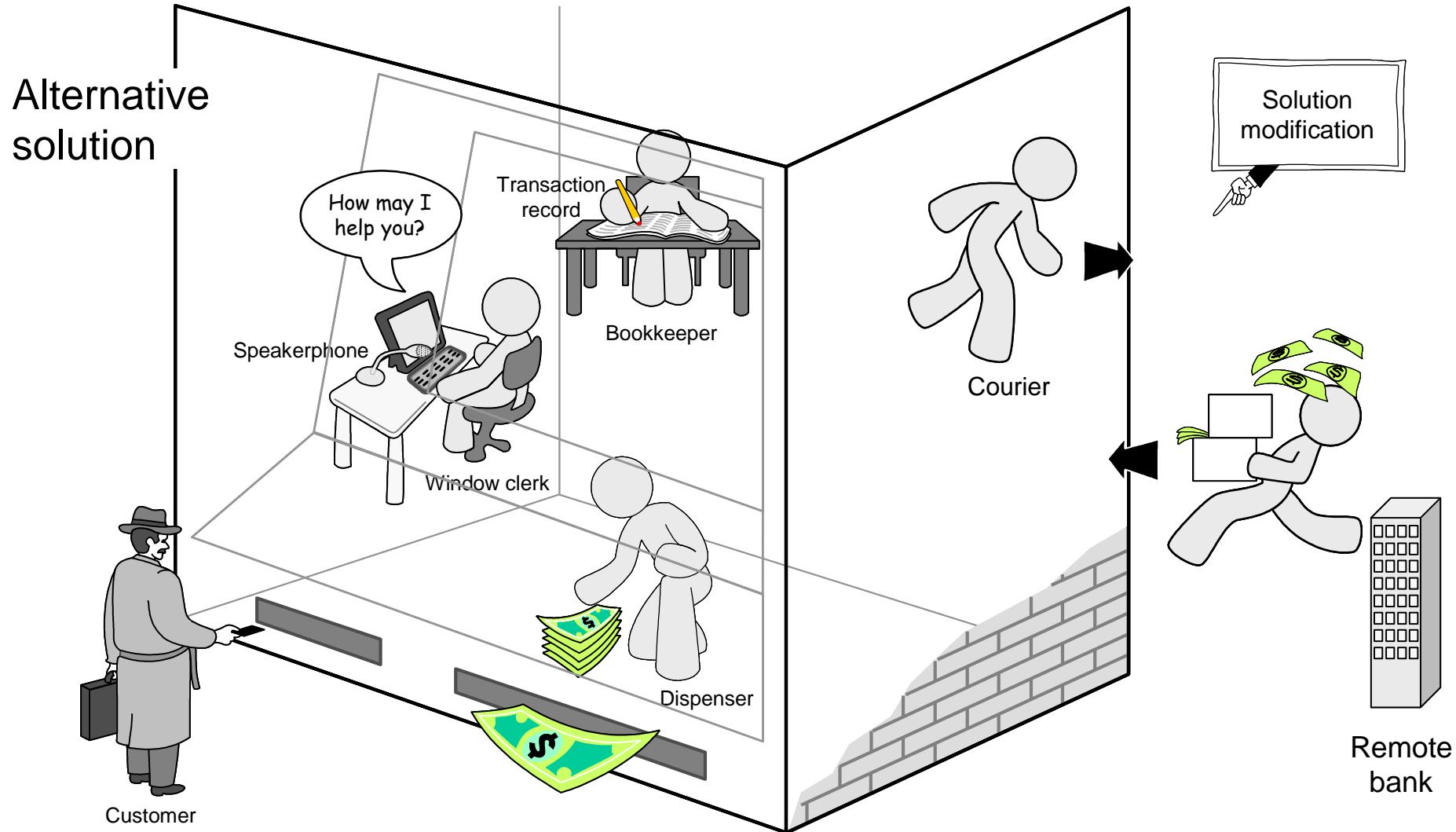
How ATM Machine Works (2)

Domain Model (2)



How ATM Machine Works (3)

Domain Model (3)



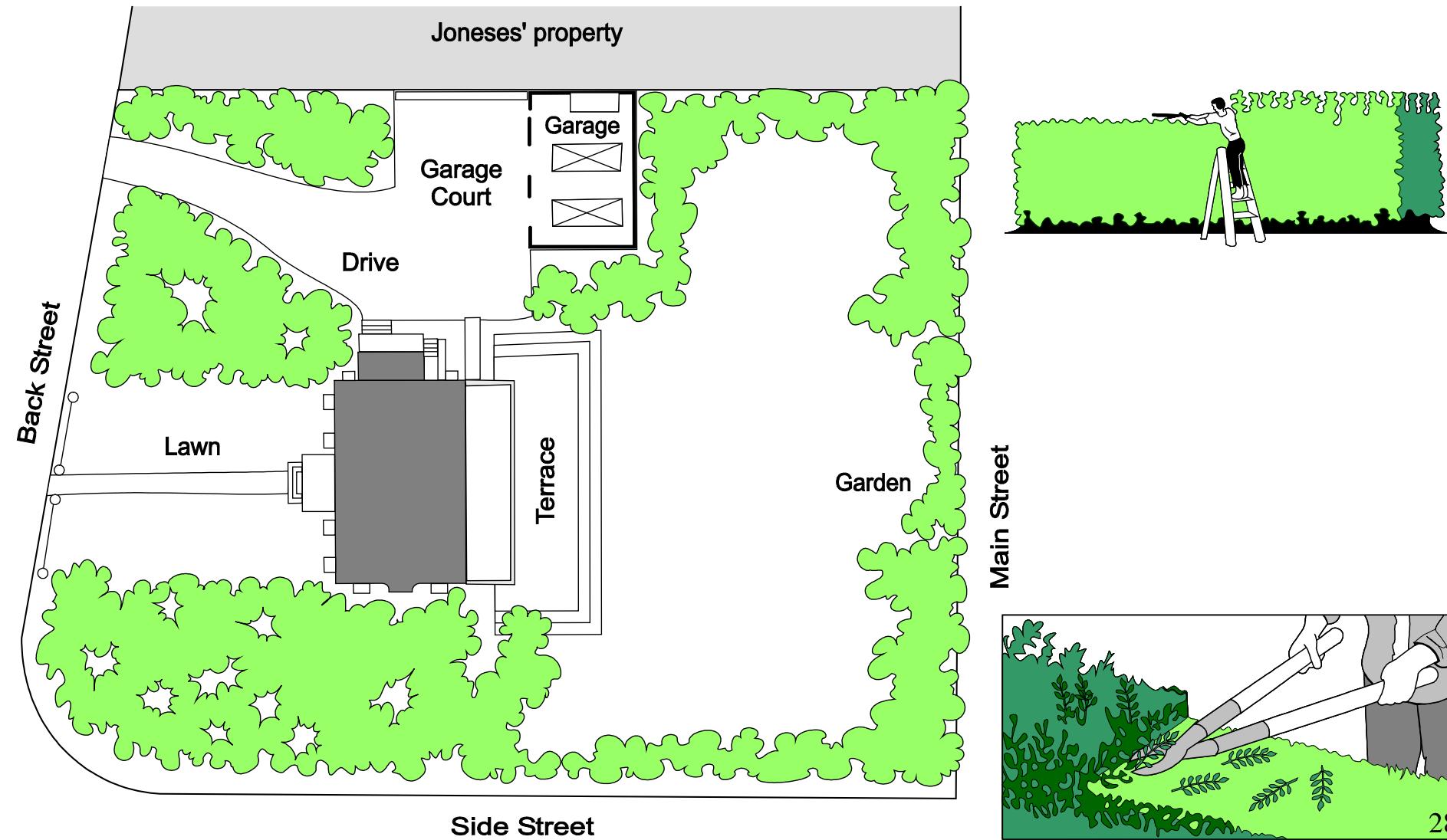
Software Design

Recurring issues of software design include:

- Design quality evaluation
- Design for change
- Design for reuse

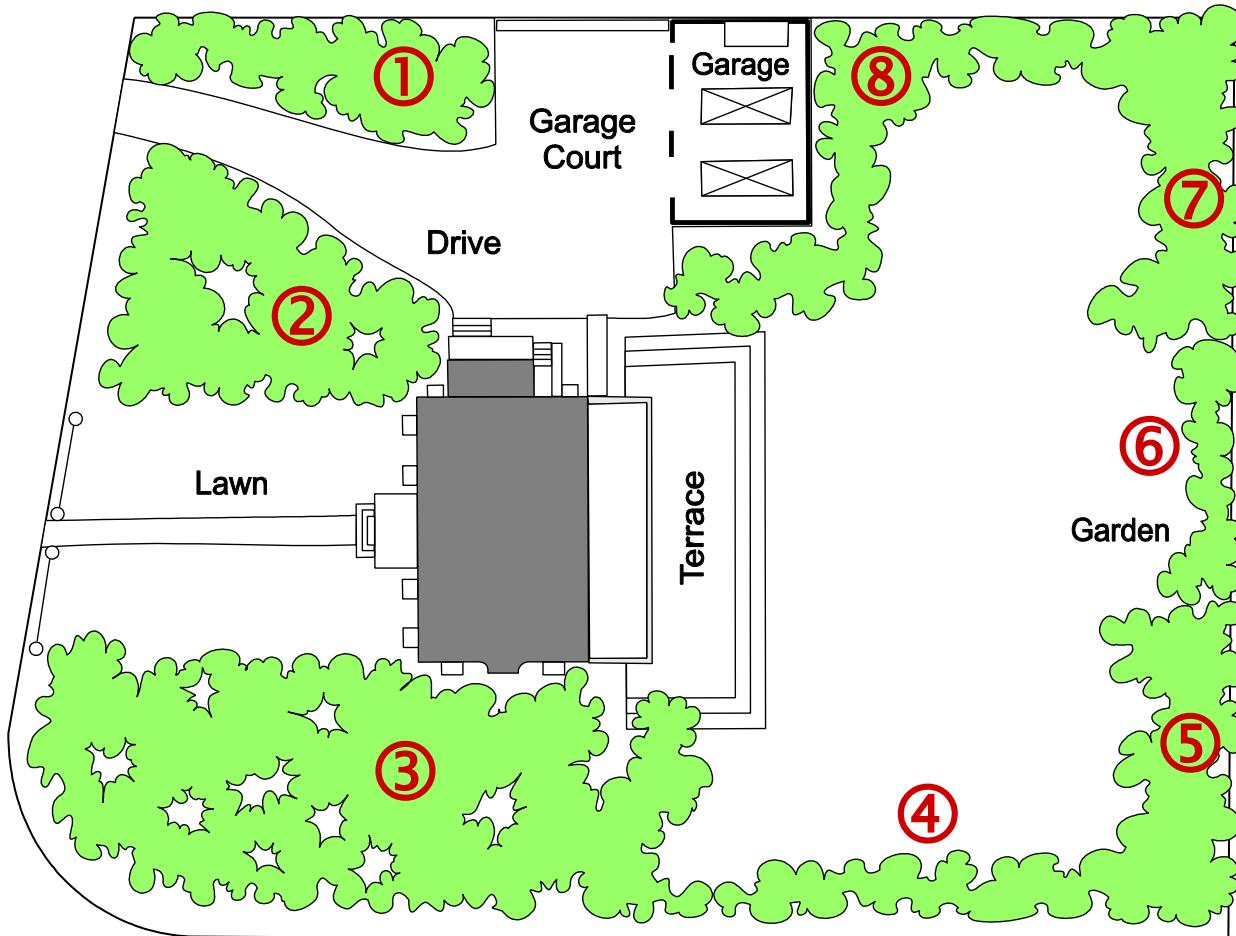
Effort Estimation and Quality measurement

Formal hedge pruning



Sizing the Problem (1)

Step 1: Divide the problem into *small & similar* parts



**Step 2:
Estimate *relative
sizes* of all parts**

$$\text{Size}(\textcircled{1}) = 4$$

$$\text{Size}(\textcircled{2}) = 7$$

$$\text{Size}(\textcircled{3}) = 10$$

$$\text{Size}(\textcircled{4}) = 3$$

$$\text{Size}(\textcircled{5}) = 4$$

$$\text{Size}(\textcircled{6}) = 2$$

$$\text{Size}(\textcircled{7}) = 4$$

$$\text{Size}(\textcircled{8}) = 7$$

Sizing the Problem (2)

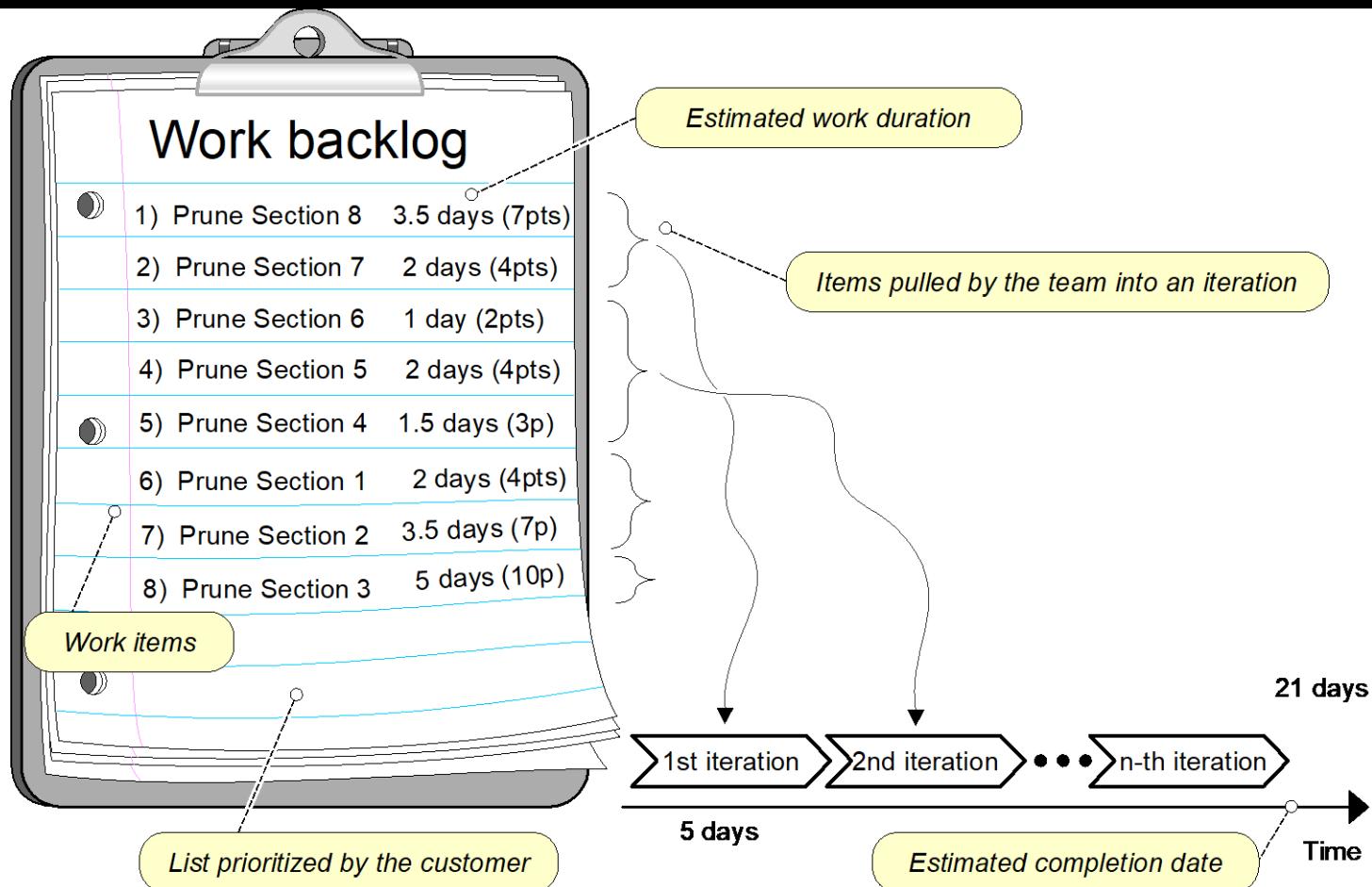
- Step 3: Estimate the size of the total work

$$\text{Total size} = \sum \text{points-for-section } i \quad (i = 1..N)$$

- Step 4: Estimate speed of work (velocity)
- Step 5: Estimate the work duration

$$\text{Travel duration} = \frac{\text{Path size}}{\text{Travel velocity}}$$

Effort Estimation of Incremental and Iterative Process

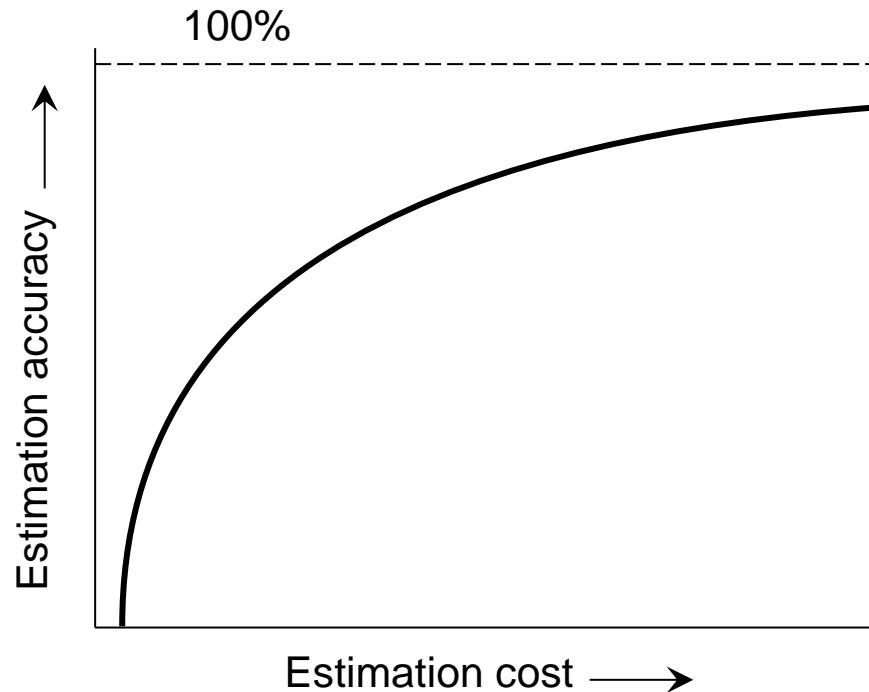


- ❖ After the first iteration you can use the **measured velocity** to obtain a more accurate **estimate** of the project duration

Sizing the Problem (3)

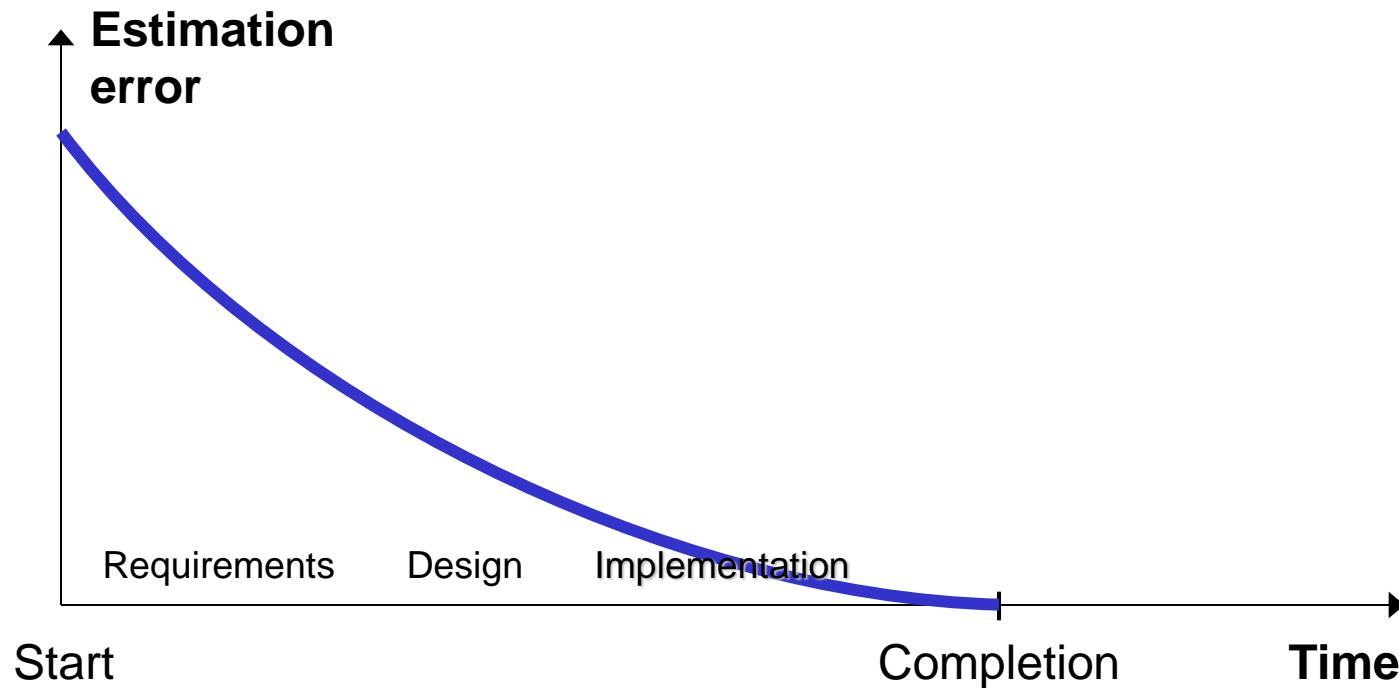
- Advantages:
 - Velocity estimate may need to be **adjusted** (based on observed **progress**)
 - However, the total duration can be computed **quickly** (provided that the *relative* size estimates of parts are accurate – easier to achieve if the parts are **small** and **similar-size**)

Exponential Cost of Estimation



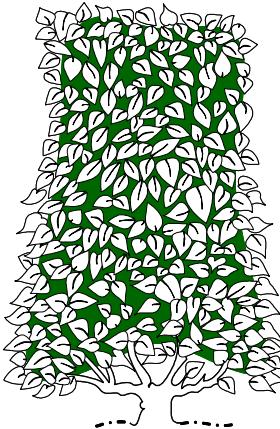
- **Improving accuracy** of estimation beyond a certain point requires **huge cost and effort** (known as the law of diminishing returns)
- In the beginning of the curve, a modest effort investment yields huge gains in accuracy

Estimation Error Over Time

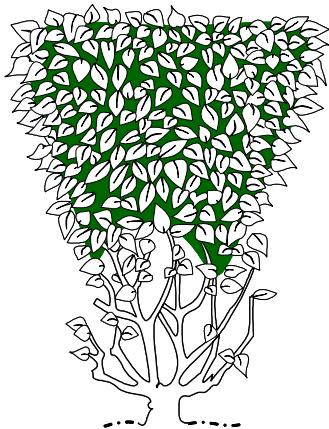


The *cone of uncertainty* starts high and narrows down to zero as the project approaches completion.

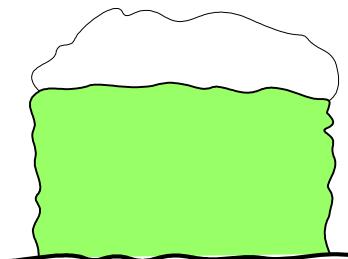
Measuring Quality of Work



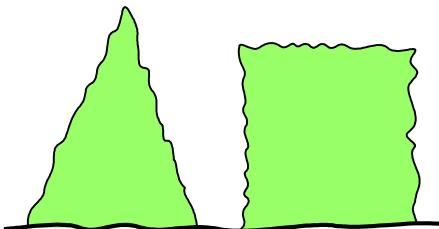
Good Shape
(Low branches get sun)



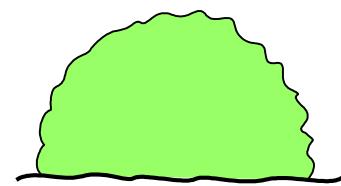
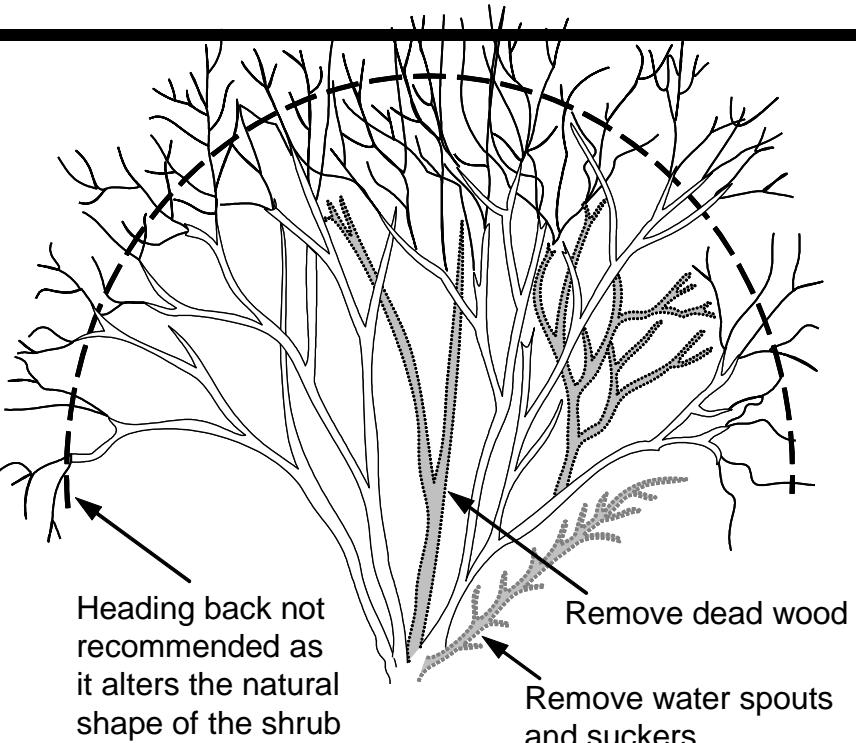
Poor Shape
(Low branches
shaded from sun)



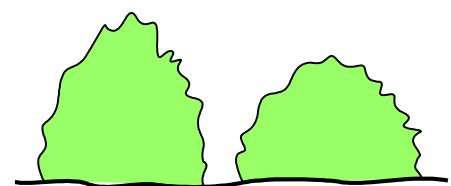
Snow accumulates
on broad flat tops



Straight lines require
more frequent trimming



Peaked and rounded tops
hinder snow accumulation



Rounded forms, which
follow nature's tendency,
require less trimming

Summary

1. *Incremental and iterative* strategy helps in **quick and more accurate effort estimation**
2. Improves the development process **quality**

Conceptual Maps

Concept maps are graphical tools for organizing and representing knowledge and concepts. Represented by:

- **circles** or **boxes** of some type, and
- relationships between the concepts are connected by **lines** linking two concepts.

How to construct a concept map:

- Establish a main concept or idea, topic, or issue to focus on
- Identify related concepts
- Connect the concepts - creating linking phrases and words
- Fine-tune the map

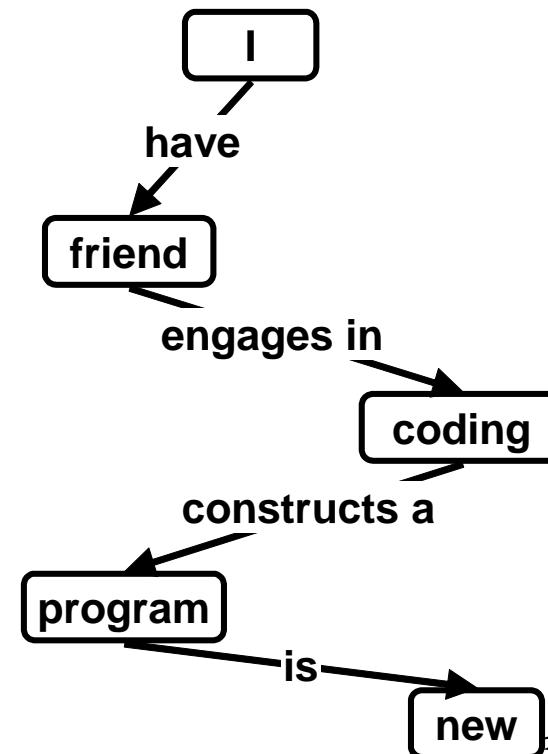
Conceptual Maps

Useful tool for problem domain description

SENTENCE: “My friend is coding a new program”

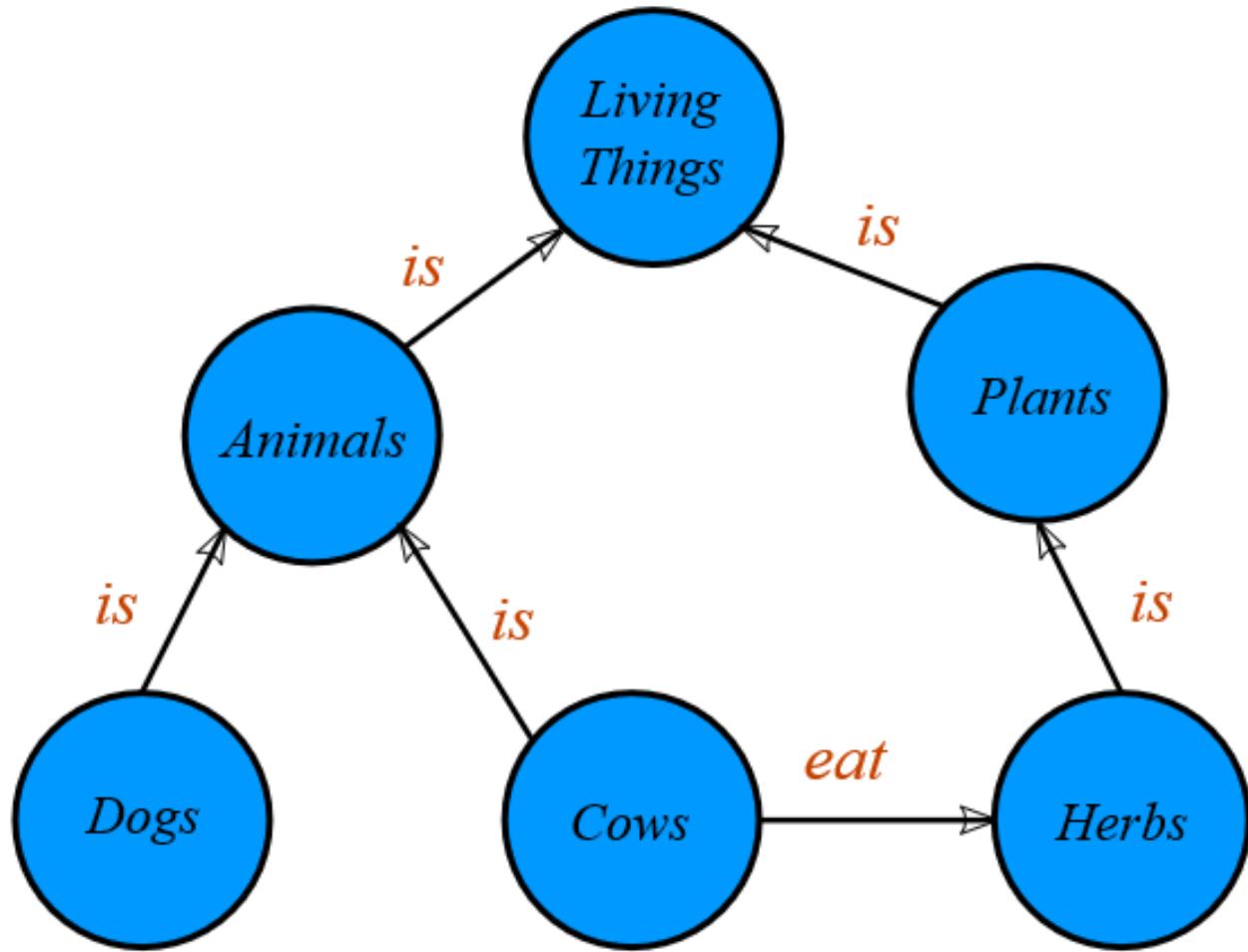
translated into propositions

Proposition	Concept	Relation	Concept
1.	I	have	friend
2.	friend	engages in	coding
3.	coding	constructs a	program
4.	program	is	new

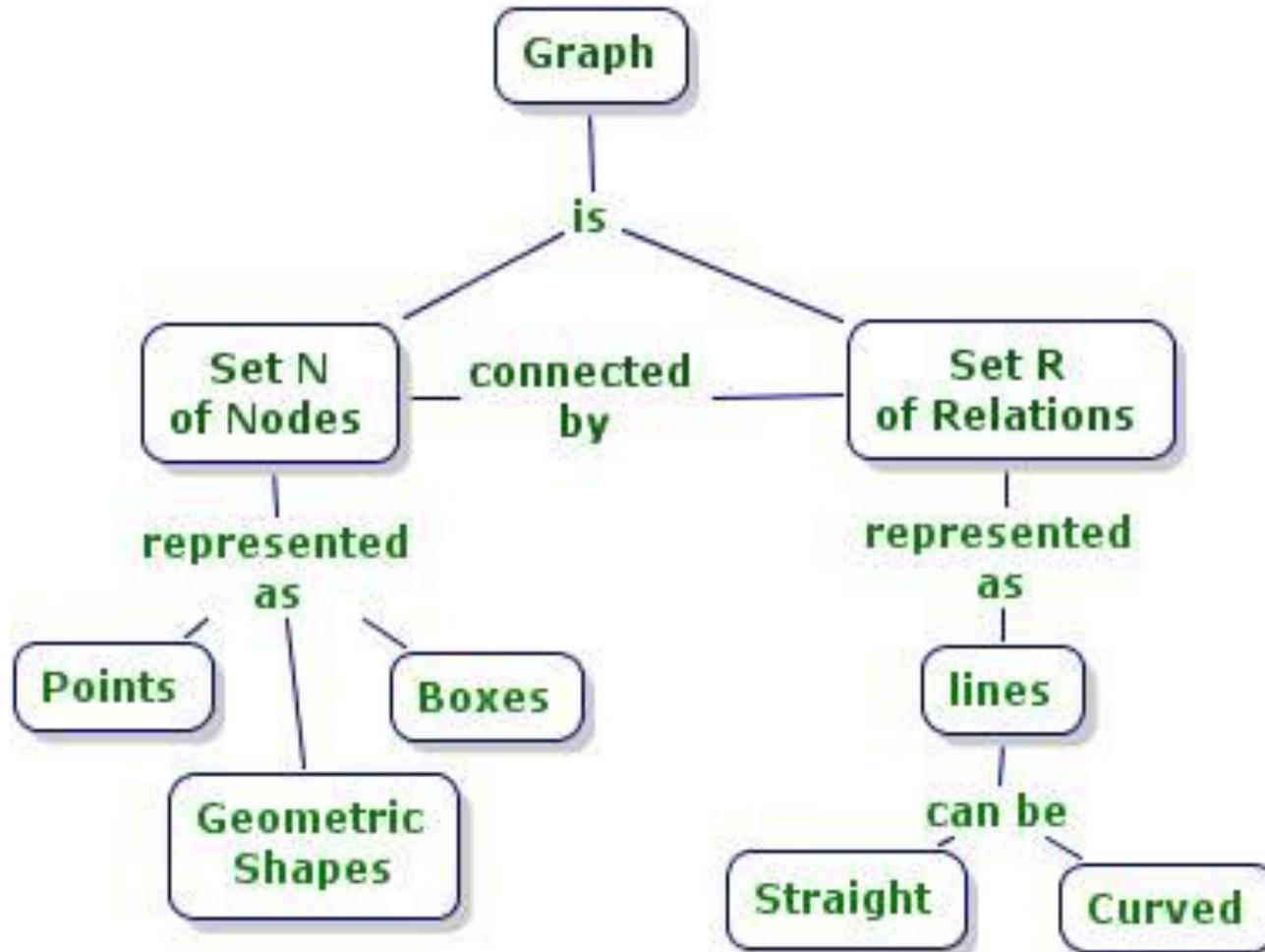


Search the Web for Concept Maps

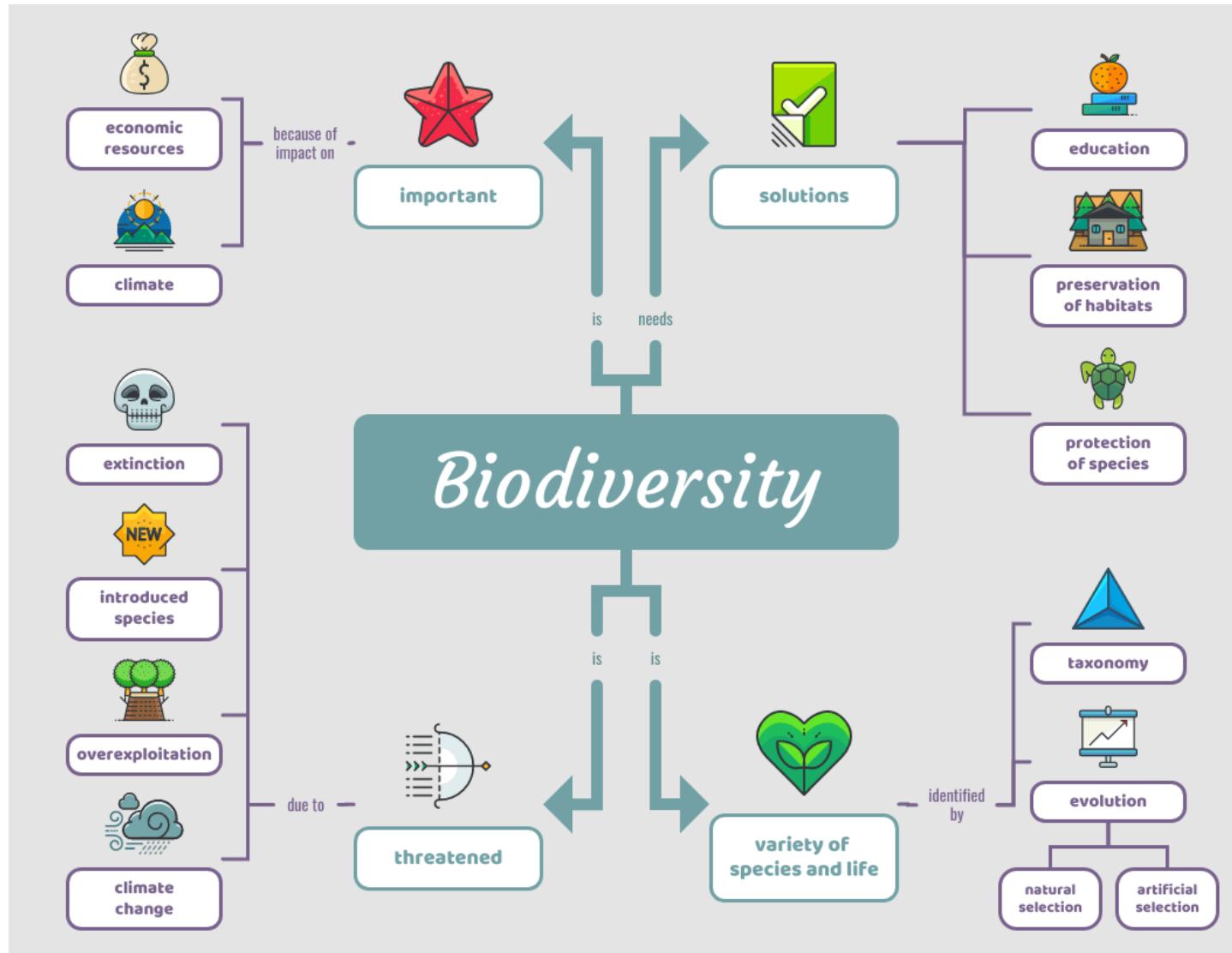
Conceptual Maps



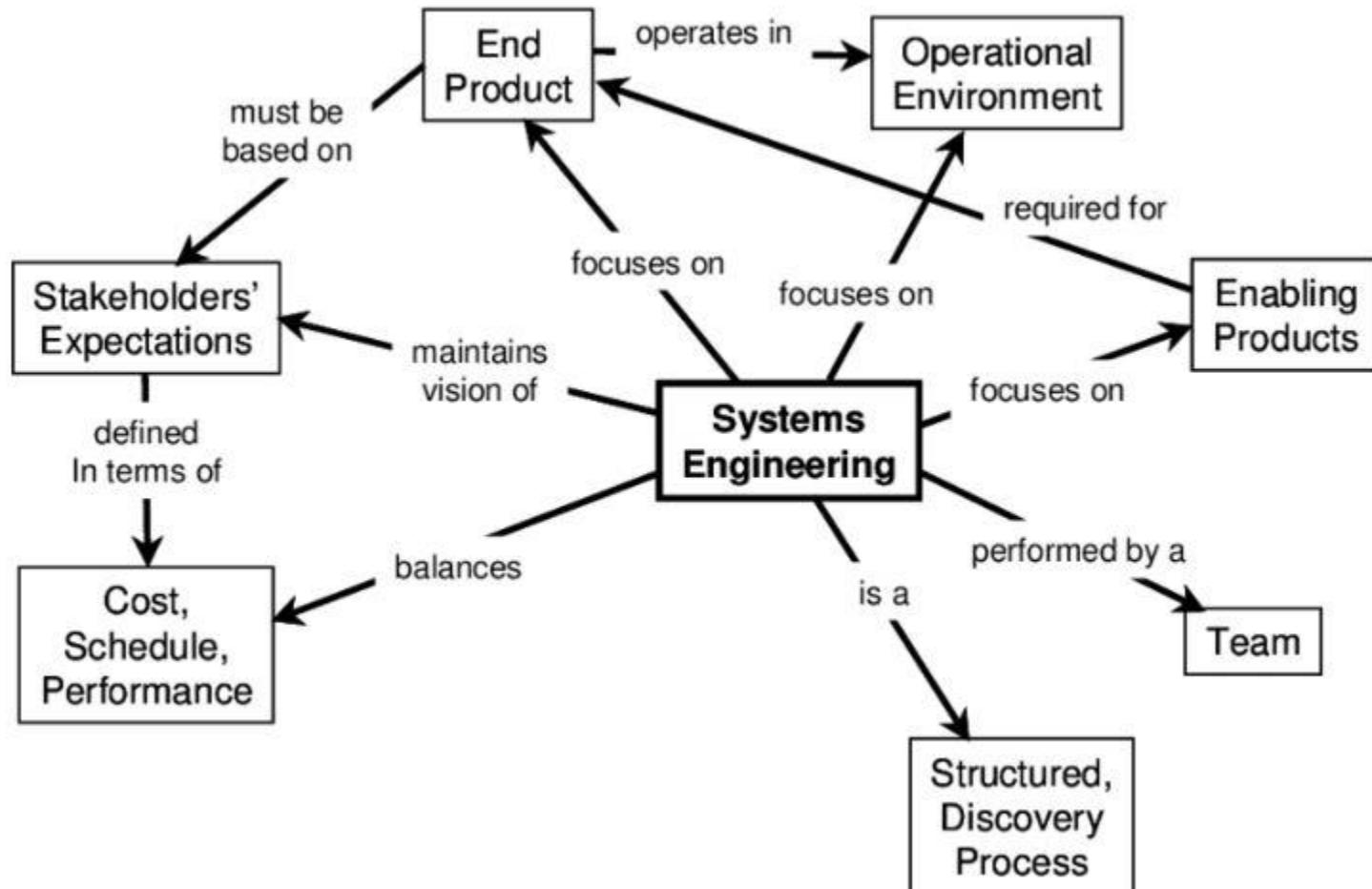
Conceptual Maps



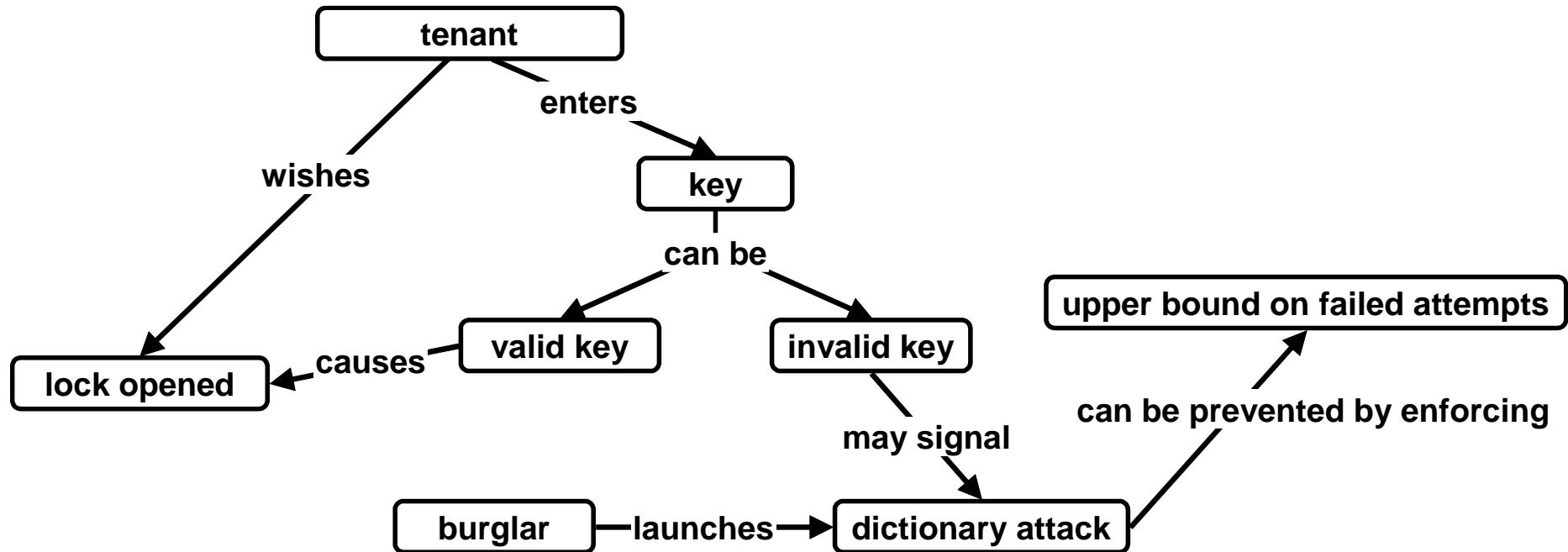
Conceptual Maps



Conceptual Maps



Concept Map for Home Access Control



Conceptual Maps

Benefits of Concept Mapping

- Help for brainstorming and generating new ideas
- Encourage to discover new concepts and the propositions that connect them
- Allow to more clearly communicate ideas, thoughts and information
- Help to integrate new concepts with older concepts
- Help to gain enhanced knowledge of any topic and evaluate the information

Practicing!!!

**Draw a Concept Map for MIST
admission process**

Software Engineering

**Professional Software Development &
Software Engineering Ethics**

Chapter Outcomes

- Understand what software engineering is and why it is important
- Understand that the development of different types of software systems may require different software engineering techniques
- Understand some ethical and professional issues that are important for software engineers

Software Engineering

- Modern world is solely computer oriented
- More and more systems are software controlled
- Software systems can quickly become extremely complex, difficult to understand, and expensive to change.
- Software engineering is concerned with theories, methods and tools for professional software development
- Universal notations, methods, or techniques for software engineering
- Software engineering is concerned with cost-effective software development.

Software Project Failure

Software Failures are a consequence of two factors:

❖ *Increasing system complexity*

- As new software engineering techniques help us to build larger, more complex systems, the demands change. Systems have to be built and delivered more quickly; larger, even more complex systems are required; systems have to have new capabilities that were previously thought to be impossible.

❖ *Failure to use software engineering methods*

- It is fairly easy to write computer programs without using software engineering methods and techniques. Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be.

Professional Software Development

Professional Software Development

- A professionally developed software system is often **more** than a single program
- Consists of a number of separate programs and configuration files
 - *system documentation*
 - *User documentation*

Frequently asked questions about software engineering

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user, should be cost-effective , maintainable , dependable and usable and ease of use should be high.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

FAQs about software engineering

Question	Answer
What are the key challenges facing software engineering?	Coping with increasing diversity , demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.
What differences has the web made to software engineering?	The web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

Types of Software products

There are two kinds of software products:

❖ Generic products

- Stand-alone systems that are sold on the open market.
- Examples – software for PCs (databases, graphics programs, project management tools); software for specific purpose (library information systems, dentists appointment)

❖ Customized products

- Software that are developed for a specific customer
- Examples – control systems for electronic devices, air traffic control software, traffic monitoring systems.

Difference between Generic and Customized products

- **Generic:** specifications are owned, modified and controlled by the **development company**
- **Customized:** Software specifications are owned, modified and controlled by the **customer**
- Many systems are now being built with a generic product as a base, which is then adapted to suit the requirements of a customer. E.g., Enterprise Resource Planning (ERP).

Qualities of Professional software

- Quality is not just concerned with what the software does
- Rather, it depends on
 1. Software's behavior while it is executing - **software's response time to a user query**
 2. The structure and organization of the system programs and associated documentation - **understandability of the program code**
- Specific set of attributes depends on its application

Essential attributes of good software

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability , security and safety . Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness , processing time , memory utilisation , etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable , usable , and compatible with other systems that they use.

Software Engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

- Engineering discipline

- Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints.

- All aspects of software production

- Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.

Fundamental Activities of Software Processes

- ❖ A software process is a sequence of activities that leads to the production of a software product
- ❖ There are four fundamental activities that are common to all software processes
 1. **Software specification** - where customers and engineers **define** the **software** that is to be produced and the **constraints** on its operation
 2. **Software development** - where the software is **designed** and **programmed**
 3. **Software validation** - where the software is checked to **ensure** that it is what the customer **requires**
 4. **Software evolution** - where the software is modified to reflect **changing** customer and market **requirements**

General issues that affect software

□ Heterogeneity

- Increasingly, systems are required to **operate** as distributed systems across networks that include different types of **computer and mobile devices**. We need to **integrate new software** with **older legacy systems** written in different programming languages. The **challenge** here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity

□ Business and social change

- Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to **change their existing software** and to rapidly **develop new software**. Many traditional software engineering techniques are **time consuming** and delivery of new systems often takes longer than planned. They **need to evolve** so that the time required for software to deliver value to its customers is reduced

General issues that affect software

□ Security and trust

- As software is intertwined with all aspects of our lives, it is essential that we can **trust** that software. We have to make sure that **malicious users** cannot attack our software and that information **security** is maintained

□ Scale

- Software has to be developed across a **very wide range of scales**, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community.

Software engineering diversity

- ✧ The software engineering approach is implemented depending on the **organization** developing the software, the **type** of software, and the **people** involved in the development process.
- ✧ There are many different types of software system and there is **no universal set of techniques** that is applicable to all of these.
- ✧ The software engineering **methods and tools** used **depend** on the type of application being developed, the requirements of the customer and the background of the development team.

Application Types

- ❖ Stand-alone applications
 - Application systems that **run on a local computer**, such as a PC. They include all necessary functionality and do not need to be connected to a network. Examples: CAD programs, photo manipulation software, etc.
- ❖ Interactive transaction-based applications
 - Applications that execute on a **remote computer** and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.
- ❖ Embedded control systems
 - These are software control systems that **control and manage hardware devices**. Numerically, there are probably more embedded systems than any other type of system. E.g., SW in a microwave oven to control the cooking process.

Application Types

❖ Batch processing systems

- These are business systems that are designed **to process data in large batches**. They process large numbers of individual inputs to create corresponding outputs. **Examples:** periodic billing systems, such as phone billing systems, and salary payment systems.

❖ Entertainment systems

- These are systems that are primarily built for personal use and are intended to entertain the user. **Examples:** computer games.

❖ Systems for modelling and simulation

- These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.

Application Types

❖ Data collection systems

- These are systems that **collect data** from their environment using a set of sensors and **send that data to other systems** for processing. Often installed in a hostile environment such as inside an engine or in a remote location.

❖ Systems of systems

- These are systems that are **composed of a number of other software systems**. Some of these may be generic software products, such as a spreadsheet program. Other systems in the assembly may be specially written for that environment

Software Engineering and The Web

- The Web was primarily a universally accessible information store
- The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- Software are **deployed** on a web server instead of users' PC, making it much **cheaper** to **change** and **upgrade** the software. **Reduces cost**, correspondingly.
- **Web services** are software components that deliver **specific**, useful **functionality** and which are **accessed** over the Web. They can be **integrated** and their linking can be **dynamic**.
- **Cloud computing** is an approach to the provision of computer services where applications run remotely on the 'cloud'.
 - Users do not buy software but pay according to use.

Web-based software engineering

- Web-based systems are complex **distributed systems** but the **fundamental** principles of software engineering discussed previously are as **applicable** to them as they are to any other types of system.
- The **fundamental ideas** of software engineering apply to web-based software in the same way that they apply to other types of software system.

Software Engineering Fundamentals

Some fundamental principles apply to all types of software system, irrespective of the development techniques used:

- Systems should be developed using a managed and understood development process. The organization developing the software should **plan** the development process and have clear ideas of **what** will be produced and **when** it will be completed.
- **Dependability** and **performance** are important for all types of system. Software should behave as expected, without failures and should be available for use when it is required .
- Understanding and managing the software **specification** and **requirements** (what the software should do) are important.
- Making as effective **use** as possible of existing **resources**.

Web Software Engineering

Features that mostly apply in web software engineering:

❖ Software reuse

- Software reuse is the dominant approach for constructing web-based systems. When building these systems, you think about how you can **assemble** them from **pre-existing software** components and systems.

❖ Incremental and agile development

- Web-based systems should be developed and delivered incrementally. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.

Web Software Engineering

- ❖ Service-oriented systems

- Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services.

- ❖ Rich interfaces

- Interface development technologies such as AJAX and HTML5 have emerged that support the creation of rich interfaces within a web browser.

Software Engineering Ethics

Software Engineering Ethics

- Software engineering involves wider responsibilities than simply the application of technical skills.
- Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals.
- Ethical behaviour is more than simply upholding the law but involves following a set of principles that are morally correct.

Software Engineering Ethics

□ Confidentiality

- Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.

□ Competence

- Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outside their competence.

Software Engineering Ethics

□ Intellectual property rights

- Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.

□ Computer misuse

- Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

ACM/IEEE Code of Ethics

- The professional societies in the US have cooperated to produce a code of ethical practice.
- Members of these organisations sign up to the code of practice when they join.
- The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

Rationale for the code of ethics

- *Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems.*
- *Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession.*

The ACM/IEEE Code of Ethics

Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

Ethical principles

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Software Processes

Chapter Objectives

Software process models

Process activities

Coping with change

Software Process

- ❖ A software process is a set of related activities that leads to the production of a software product
- ❖ There are four fundamental software process activities:
 - Specification – defining what the system should do;
 - Design and implementation – defining the organization of the system and implementing the system;
 - Validation – checking that it does what the customer wants;
 - Evolution – changing the system in response to changing customer needs.
- ❖ These activities further include sub-activities (requirements validation, architectural design) and supporting activities (documentation)

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.

Categorization of Software Processes

- Plan-driven processes
 - all the process activities are planned in advance and progress is measured against this plan.
- Agile processes
 - planning is incremental and it is easier to change the process to reflect changing customer requirements.
- In practice, most practical processes include elements of both plan-driven and agile approaches.
- There is no ‘ideal’ software process, thus, there is scope for improving the software process.

Software Process Models

Software Process Models

A software process model is an abstract representation of a process.

1. The waterfall model

- Takes the fundamental process activities and represents them as separate and distinct phases. Plan-driven model.

2. Incremental development

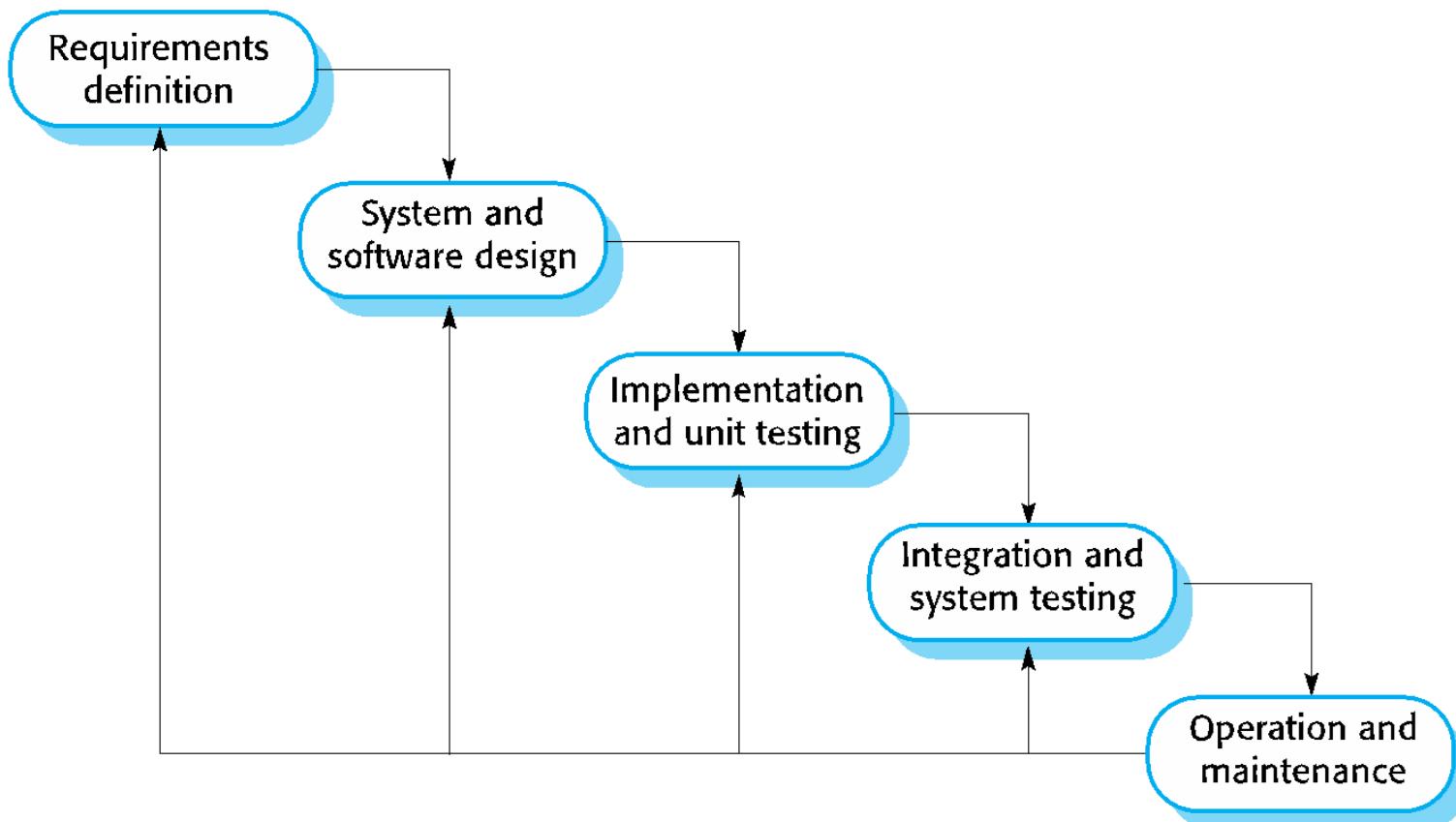
- The system is developed as a series of versions (increments), with each version adding functionality to the previous version. Could be plan-driven or agile.

3. Reuse-oriented software engineering

- The system is assembled from existing configurable 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.

The Waterfall Model



Phases of Waterfall Model

- ❖ There are separate identified phases in the waterfall model:
 - Requirements analysis and definition
 - System and software design
 - Implementation and unit testing
 - Integration and system testing
 - Operation and maintenance

- ❑ Drawback - difficulty of accommodating change after the process is underway. In principle, **a phase has to be complete before moving onto the next phase.**

Problems of Waterfall Model

There are some reasons which are given below due to this waterfall model fails.

- **One way street:**

This model is just like the one-way street. Once a phase is completed and next phase has started, there is no way to going back on the previous phase.

- **Overlapping:**

The waterfall model has lacked an overlapping among phases. It recommends that new phase can start only after the completion of the previous phase. But in real projects, this can't be maintained. To increase the **efficiency** and **reduce the cost**, phases may overlap.

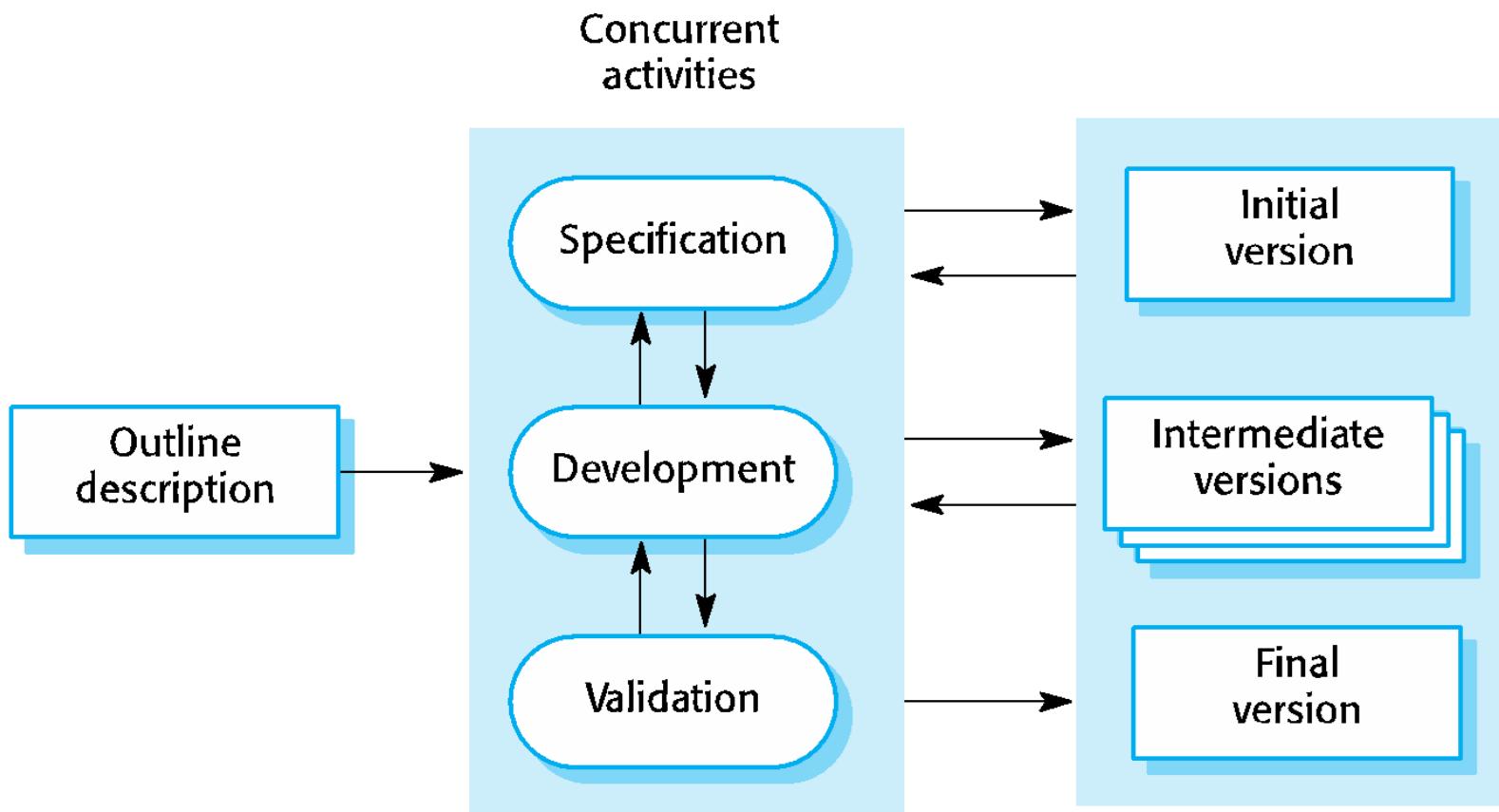
- **Interaction:**

Users have little interaction with project. This feedback is not taken during development. After a development process starts, **changes can not done easily**.

Problems of Waterfall Model

- **Feedback path:**
The waterfall model has no feedback path. In the traditional waterfall model evolution of software from one phase to another phase is like a waterfall. The waterfall model assumes that **no error** is ever committed by developers during any phases. Hence, it **does not incorporate any mechanism for error correction**.
- **Not Flexible:**
Difficult to accommodate change requests. The waterfall model assumes that all the customer requirements can be completely and correctly defined at the beginning of the project, but actually customers' requirements keep on changing with time. **After the requirements specification phase is completed difficult to accommodate any change requests.**

Incremental Development



Benefits of Incremental Development

1. 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.
1. 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.
1. 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

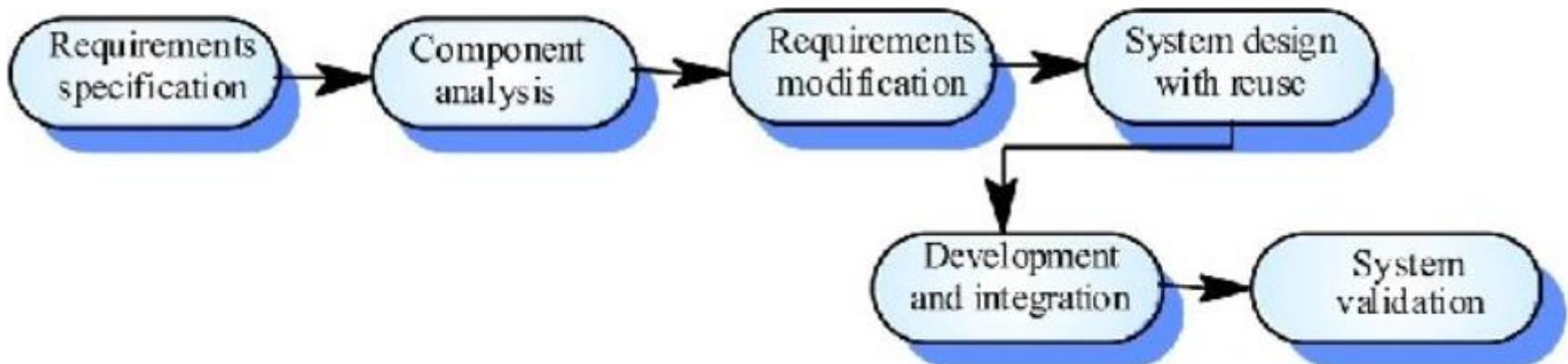
- ❖ The process is not visible.
 - Managers need regular deliverables to measure progress. If systems are developed quickly, it is **not cost-effective** to produce documents that reflect every version of the system.

- ❖ 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**.

Reuse-Oriented Software Engineering

- ❖ Reuse-oriented approaches rely on a large base of **reusable software components** and an **integrating framework** for the composition of these components (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

Reuse-Oriented Software Engineering



Key Process Stages

□ Component Analysis

- Given the requirements specification, a [search](#) is made for components to implement that specification

□ Requirements Modification

- The requirements are analyzed using information about the components. Can search for [alternative solutions](#).

□ System Design with Reuse

- The framework of the system is designed or an existing framework is reused

□ Development and integration

- Software that cannot be externally procured is developed, and the components are integrated to create the new system

Types of Reusable Software

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

Advantages and Disadvantages

Advantages

- Reduced costs and risks as less software is developed from scratch
- Faster delivery and deployment of system

Disadvantages

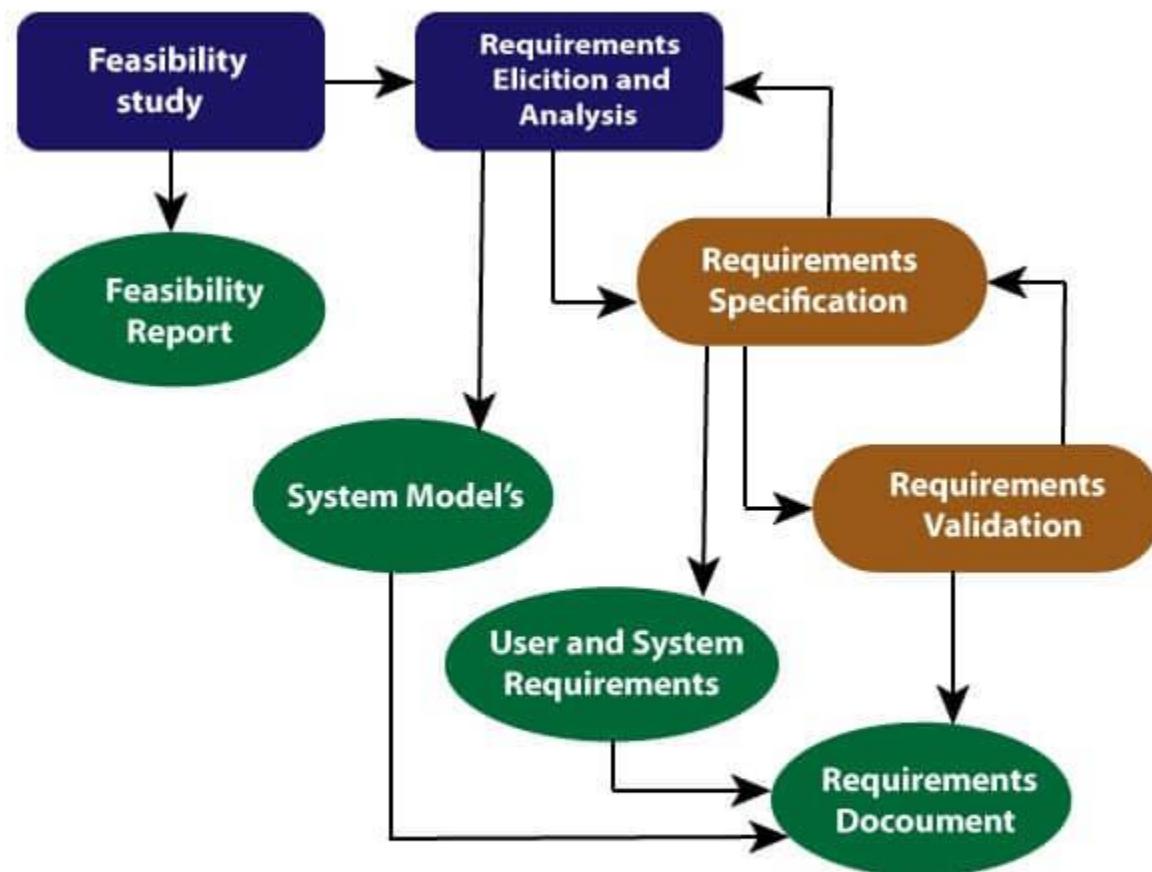
- But requirements compromises are inevitable so system may not meet real needs of users
- Loss of control over evolution of reused system elements as new versions are not under the control of the organization using them

Process Activities

Software Specification

- ❖ Software specification or **requirements engineering** is the process of understanding and defining what **services** are required from the system and the **constraints** on the system's operation and development.
- ❖ There are four main activities in the requirements engineering process:
 - Feasibility study**
 - Whether the requirements can be satisfied with existing technologies? cost-effective and within budget?
 - Requirements elicitation and analysis**
 - What do the system stakeholders require or expect from the system?
Observing existing systems, discussions with potential users, task analysis.
 - Requirements specification**
 - Translating the information gathered during the analysis activity into a document
 - Requirements validation**
 - Checks for realism, consistency, and completeness

The Requirements Engineering Process



Requirement Engineering Process

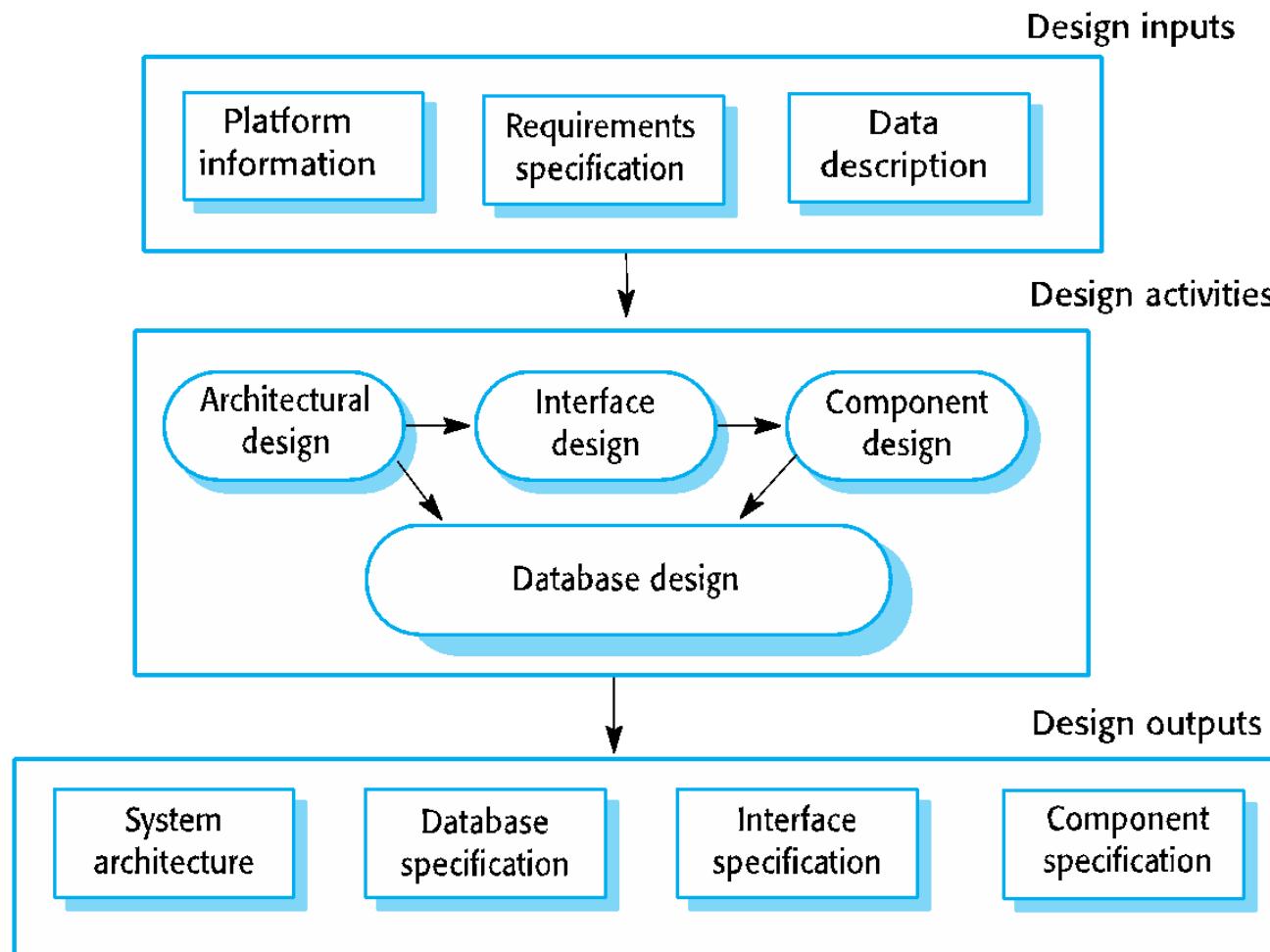
Software Design and Implementation

Converts the system specification into an executable system.

1. Software design
 - Design a software structure that realises the specification;
2. Implementation
 - Translate this structure into an executable program;

The activities of design and implementation are closely related and may be inter-leaved.

A General Model of the Design Process



Design Activities

□ Architectural design

- identifying the **overall structure** of the system, the principal **components** (subsystems or modules), their **relationships** and how they are distributed.

□ Interface design

- defining the interfaces between system components.

□ Component selection and design

- we take each system component and design how it will operate.

□ Database design

- designing the system data structures and how these are to be represented in a database.

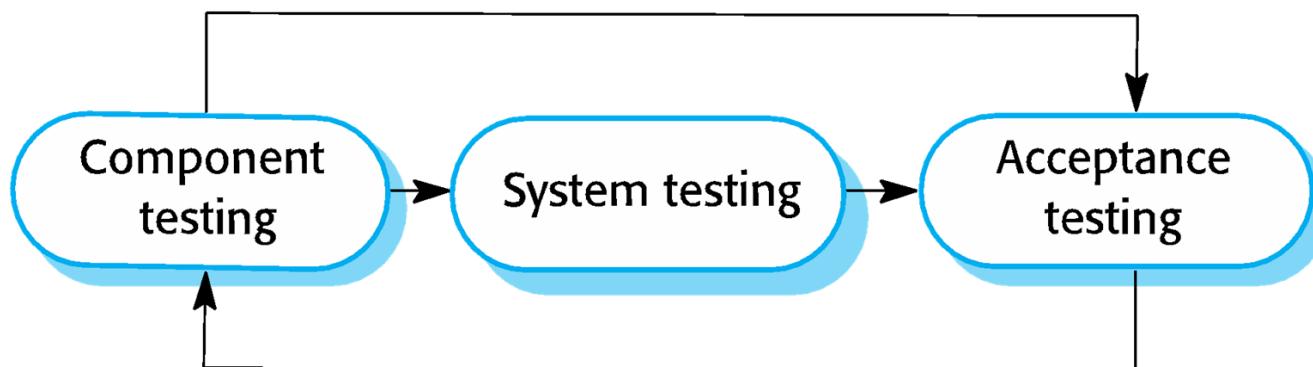
System Implementation

- ❖ The software is implemented either by developing a program or programs or by configuring an application system.
- ❖ Design and implementation are interleaved activities for most types of software system.
- ❖ Programming is an individual activity with no standard process.
- ❖ Debugging is the activity of finding program faults and correcting these faults.

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.

Stages of Testing



Testing Stages

□ Component testing

- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.

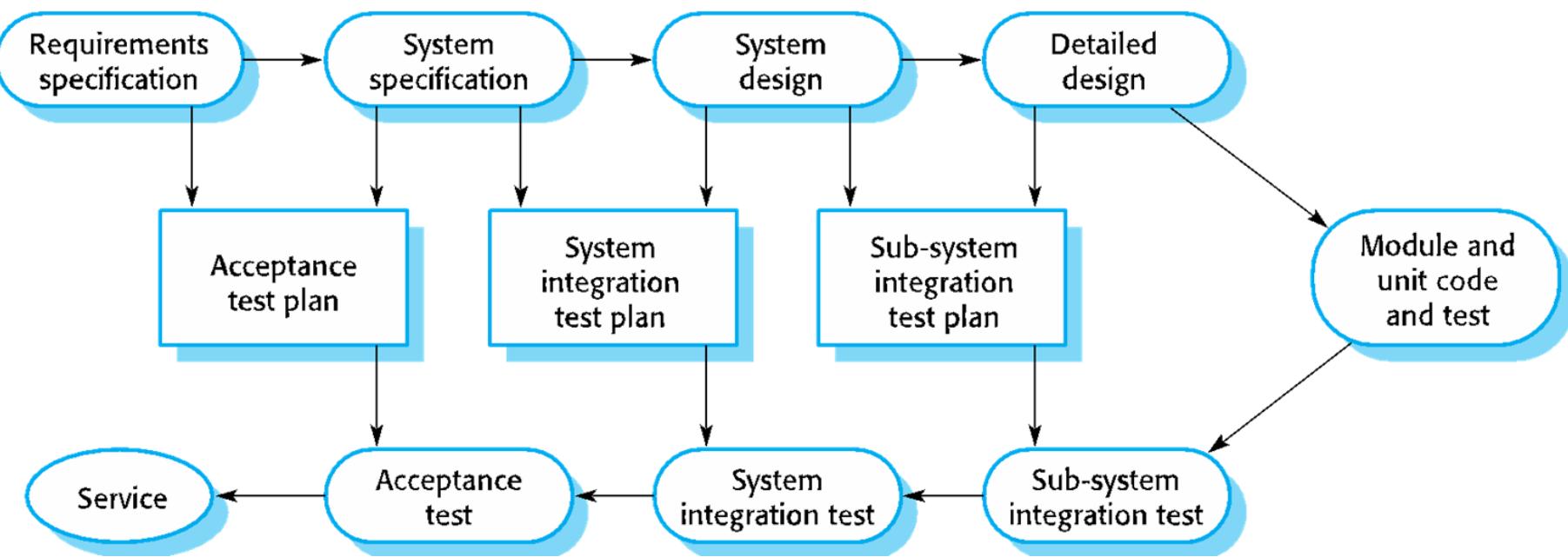
□ System testing

- Testing of the system as a whole. Testing of emergent properties is particularly important.

□ Acceptance 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)



Testing phases in a plan-driven software process (V-model)

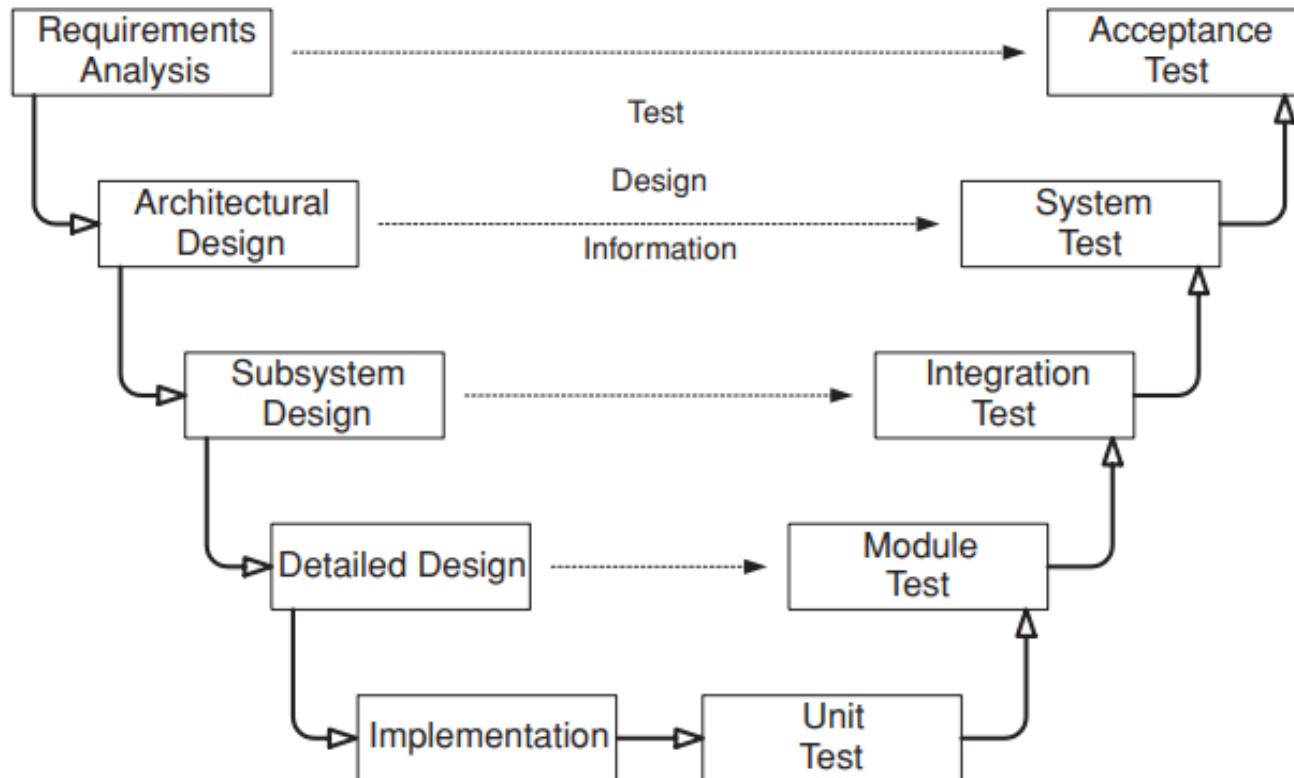


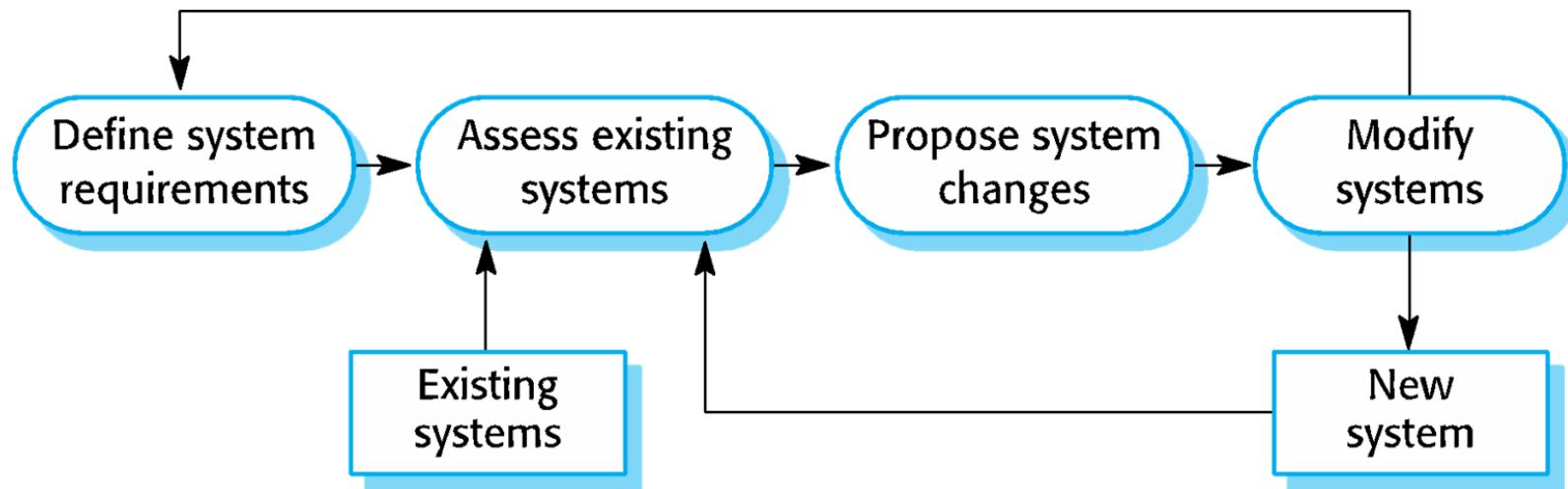
Figure 1.2. Software development activities and testing levels – the “V Model”.

Software Evolution

- ❖ The process of developing software initially, then timely updating it for various reasons, i.e., to add new features or to remove obsolete functionalities etc.

- ❖ The **necessity** of Software evolution:
 - Change in requirement with time
 - Environment change
 - Errors and bugs
 - Security risks
 - For having new functionality and features

Software Evolution



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.
- ❖ Whatever software process model is used, it is essential that it can accommodate changes to the software being developed
- ❖ 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

There are **two related approaches** that may be used to reduce the costs of rework:

❖ Change avoidance

- 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

- the process is designed so that changes can be accommodated at relatively **low cost**.
- normally involves some form of **incremental development**.

Coping with change and changing requirements

□ System prototyping

- 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 avoidance.

□ Incremental delivery

- system increments are delivered to the customer for comment and experimentation. This supports **both** change avoidance and change tolerance.

Software Prototyping

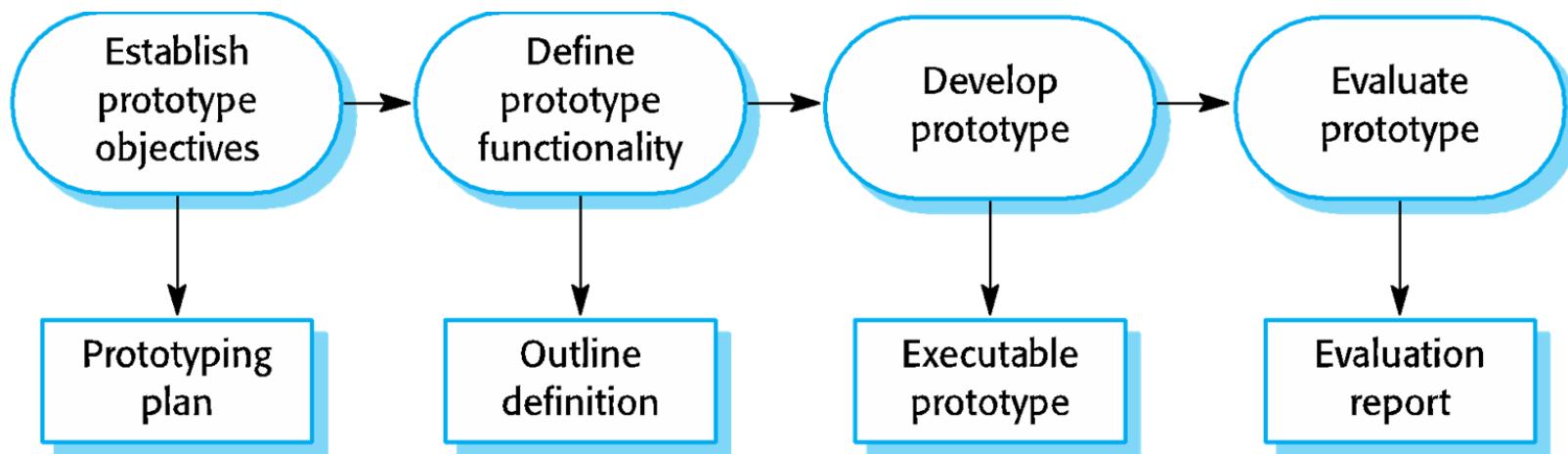
□ A prototype is an initial version of a system used to demonstrate concepts and try out design options.

- get new ideas for requirements
- find areas of strength and weakness in the software.
- propose new system requirements.
- may reveal errors and omissions in the proposed requirements

□ Benefits of prototyping:

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

The process of prototype development



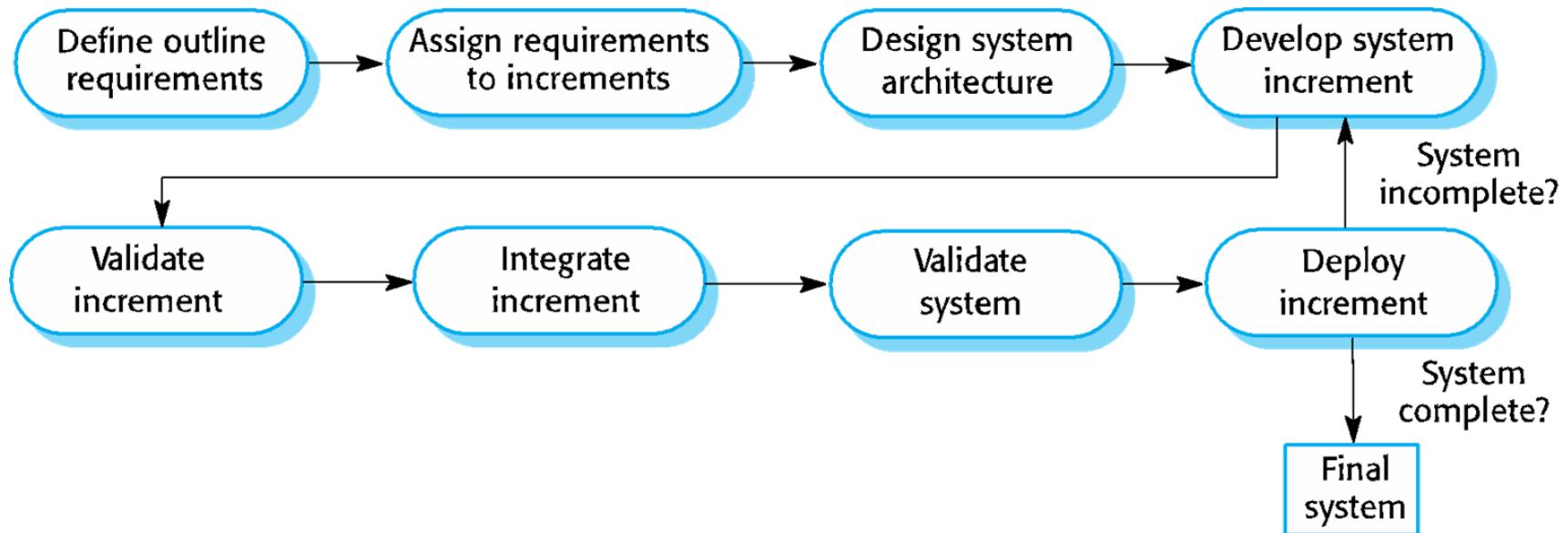
Prototype development

- ❖ 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

Incremental Delivery

- ✧ An approach to software development where some of the developed increments are delivered to the customer and deployed for use in an operational environment
- ✧ 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 prioritized and the **highest priority** requirements are included in early increments.
- ✧ Each increment provides a **subset** of the system functionality.
- ✧ As new increments are completed, they are integrated with existing increments so that the system functionality improves with each delivered increment

Incremental delivery



Advantages of Incremental Delivery

- Customers do not have to wait until the entire system is delivered before they can gain value from it.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- Easy to incorporate changes into the system.
- The highest priority system services tend to receive the most testing.

Problems of Incremental Delivery

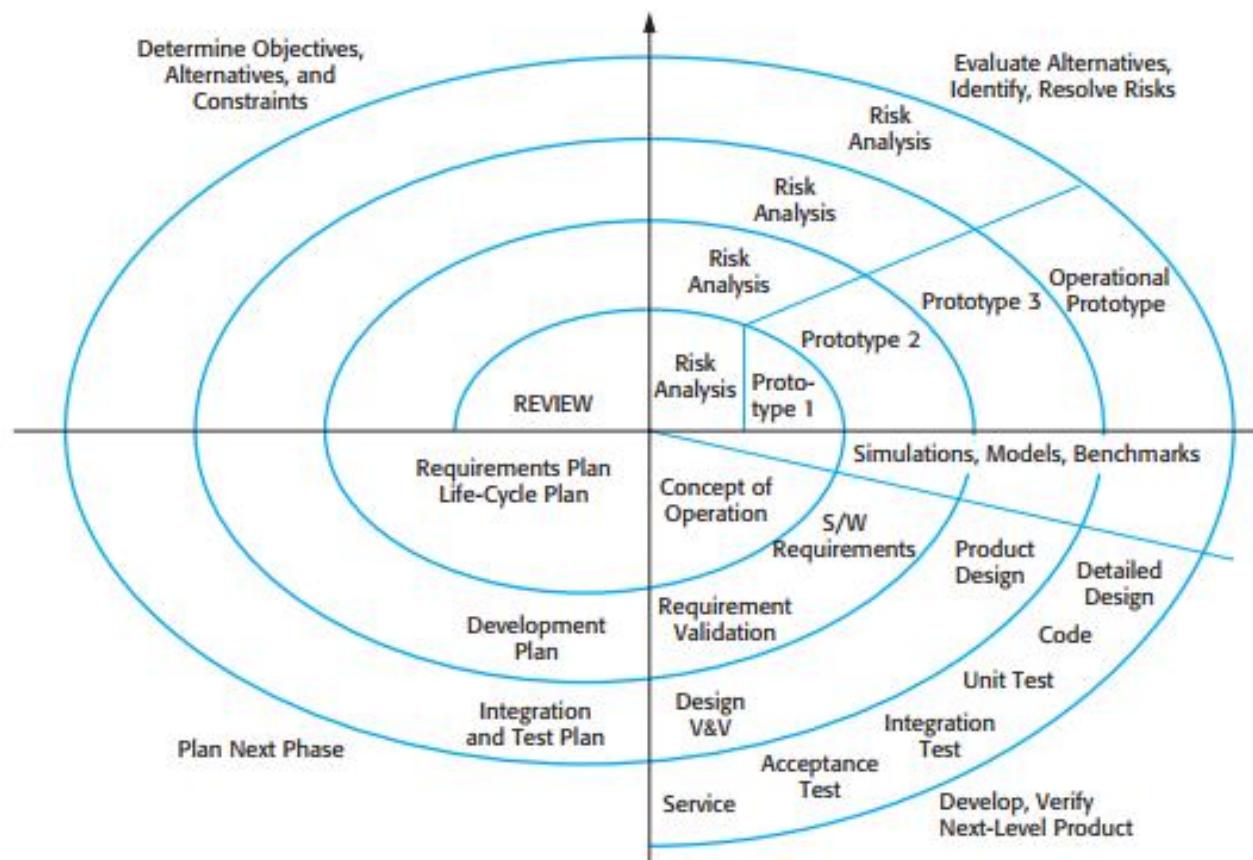
- Most systems require a set of **basic facilities** that are used by different parts of the system.
 - As requirements are not defined in detail until an increment is to be implemented, it can be **hard to identify common facilities** that are needed by all increments.
- Iterative development can also be difficult when a **replacement system** is being developed.
 - Getting useful customer **feedback is difficult**.
- The essence of iterative processes is that the **specification is developed in conjunction** with the software.
 - there is **no complete system specification** until the final increment is specified. government agencies may find difficult to accommodate.

Boehm's Spiral Model

Boehm's Spiral Model

- ✧ The software process is **represented as a spiral**, rather than a sequence of activities with some backtracking from one activity to another.
- ✧ Each loop in the spiral represents a phase of the software process.
- ✧ The innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on.
- ✧ It combines change avoidance with change tolerance
- ✧ It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks

Spiral Model



Spiral Model

Each loop in the spiral is split into four sectors:

- ✧ Objective setting
 - Specific objectives for that phase of the project are defined. Constraints, detailed management plan, risks, alternative strategies are identified.
- ✧ Risk assessment and reduction
 - For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk.
- ✧ Development and validation
 - After risk evaluation, a development model for the system is chosen.
- ✧ Planning
 - The project is reviewed and a decision made whether to continue with a further loop of the spiral.

Agile Software Development

Chapter Outline

- Agile methods
- Plan-driven and agile development
- Agile development techniques
- Agile project management

Rapid Software Development

- ❖ Businesses operate in a global, rapidly changing environment.
- ❖ Software has to evolve quickly to reflect changing business needs.
- ❖ **Plan-driven development** is essential for some types of system but does not meet these business needs.
- ❖ **Rapid software development processes** are designed to produce useful software quickly. The software is not developed as a single unit but as a series of increments, with each increment including new system functionality
- ❖ **Agile development** methods emerged in the late 1990s whose aim was to radically **reduce the delivery** time for working software systems

Fundamentals of Rapid Software Development

- ❑ Program specification, design and implementation are interleaved, no detailed system specification, and design documentation is minimized
- ❑ The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation
- ❑ System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created by drawing and placing icons on the interface
- ❑ 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 Vs Agile development

□ Plan-driven development

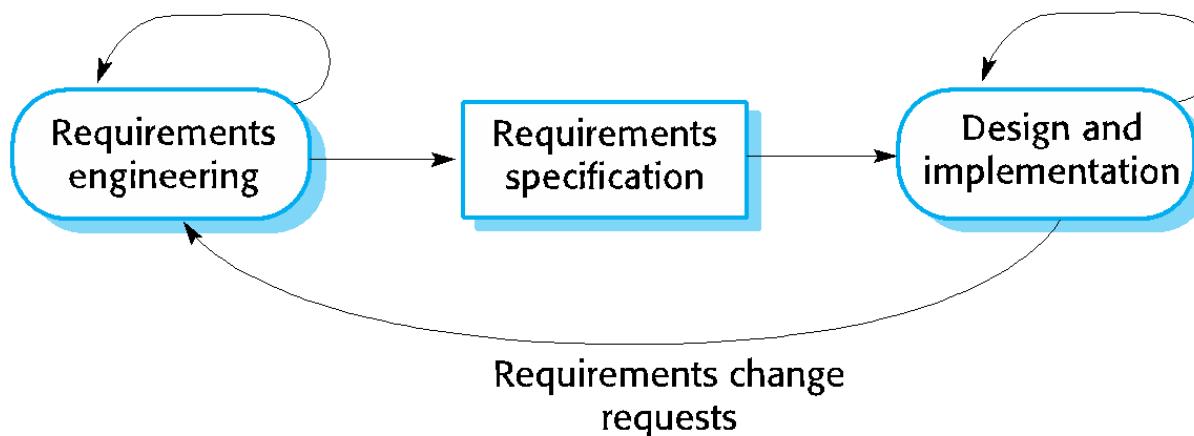
- A plan-driven approach is based around **separate development stages** with the outputs to be produced at each of these stages planned in advance.
- Iteration occurs **within** activities with formal documents used to communicate between stages of the process.
- Good for safety-critical systems

□ Agile development

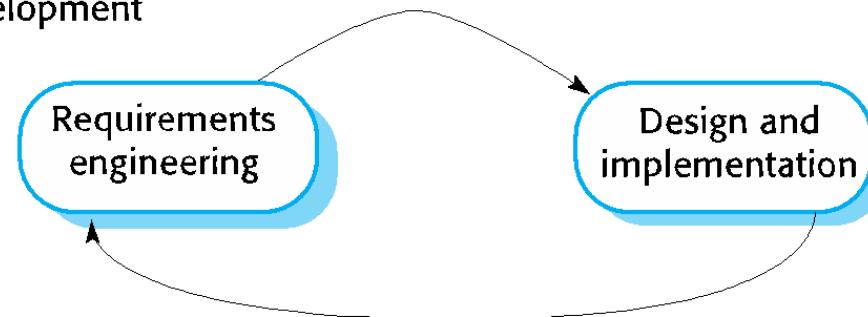
- The software development approach based on iterative development where specification, design, implementation and testing are **interleaved**.
- Good for fast moving business environment
- May include a documentation ‘**spike**’, where, instead of producing a new version of a system, the team produces system documentation

Plan-driven Vs Agile development

Plan-based development



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
 - Based on an iterative approach to software development
 - Aims to **deliver** and **evolve** working software **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.

The Principles of Agile Methods

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is to 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 Development Techniques

Agile Methods



Extreme Programming (XP)

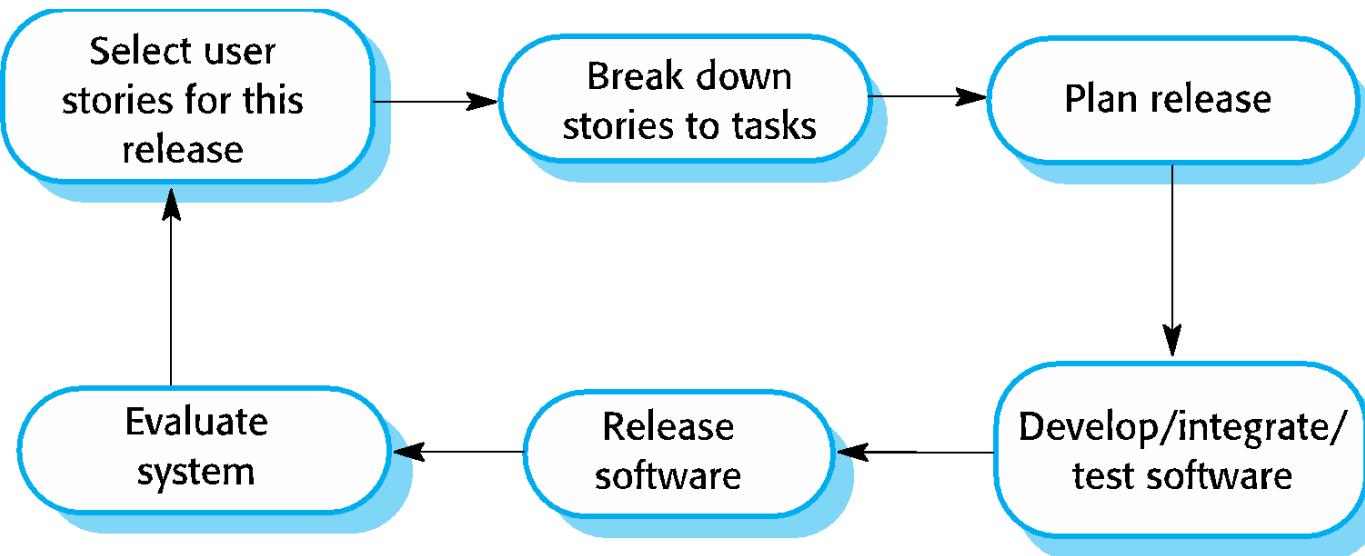
- Emphasizes on values such as **Communication**, **Simplicity**, **Feedback**, **Courage** and **Respect**, and prioritizes customer satisfaction over everything else.

- The approach was developed by pushing recognized good practices, such as iterative development, to ‘extreme’ levels.
 - For example: in XP, several new versions of a system may be developed by different programmers, integrated and tested in a day

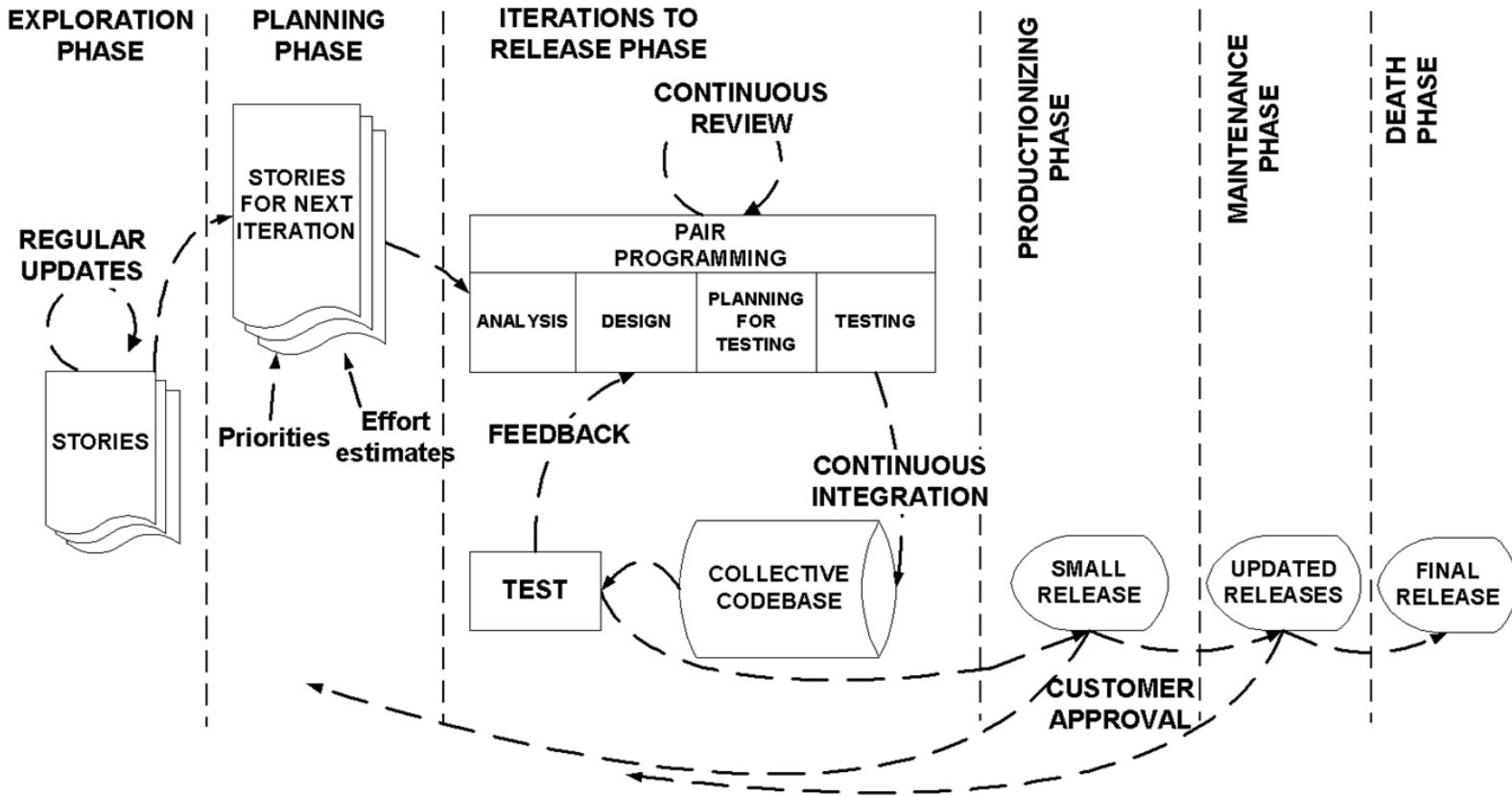
Extreme Programming (XP)

- Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.
 - Requirements are expressed as scenarios (called user stories)
 - Stories are implemented directly as a series of tasks;
 - Programmers develop tests for each task before writing the code.
 - All tests must be successfully executed when new code is integrated into the system

The XP Release Cycle



The Extreme Programming Process



The Extreme Programming Process

✧ Exploration

- The customers write out the story cards
- Project team familiarize themselves with the tools and technology
- Takes between a few weeks to a few months

✧ Iterations to release phase

- Includes several iterations of the systems before the first release
- Each take one to four weeks to implement
- The first iteration creates a system with the architecture of the whole system
- Customer decides the stories to be selected for each iteration

The Extreme Programming Process

✧ Productionizing phase

- Requires extra testing and checking
- New changes may still be found
- The iterations may need to be quickened
- The postponed ideas and suggestions are documented for later implementation, e.g., the maintenance phase.

✧ Death phase

- When the customer no longer have any stories to be implemented.
- System satisfies customer needs
- Necessary documentation of the system is finally written
- Death may also occur if the system is not delivering the desired outcomes, or if it becomes too expensive for further development

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.

Extreme Programming Practices (a)

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 ’. See Figures 3.5 and 3.6.
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 refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable .

Extreme programming practices (b)

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.

Key Practices in XP

- ❖ Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.

- ❖ Key practices
 - User stories for specification
 - Refactoring
 - Test-first development
 - Pair programming

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

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.

Examples of task cards for prescribing medication

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

- ✧ Incremental development tends to degrade the software structure, so changes to the software become **harder** to implement
 - code is often **duplicated**
 - parts of the software are **reused** in inappropriate ways
 - overall **structure degrades** as code is added to the system
- ✧ Refactoring is the process of **restructuring** code, while **not changing** its **original functionality** yet improves its internal structure.
- ✧ The goal of refactoring is to improve internal code by making many **small changes** without altering the **code's external behavior**.
- ✧ When carried out **manually**, refactoring is applied directly to the source code and is generally a **labor-intensive**, ad hoc, and potentially **error-prone process**.
- ✧ Program development environments, such as **Eclipse** (Carlson, 2005), include tools for refactoring which **simplify** the process of **finding dependencies between code sections** and making **global code modifications**

Benefits of Refactoring

- Programming team looks 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 require 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.

Testing with XP

- ✧ 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.
 - Customer involvement in test development and validation.
 - Use of automated testing frameworks

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 shortcuts 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.

Pair programming

- ✧ In pair programming, programmers **sit together** at the same computer to develop the software.
- ✧ **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

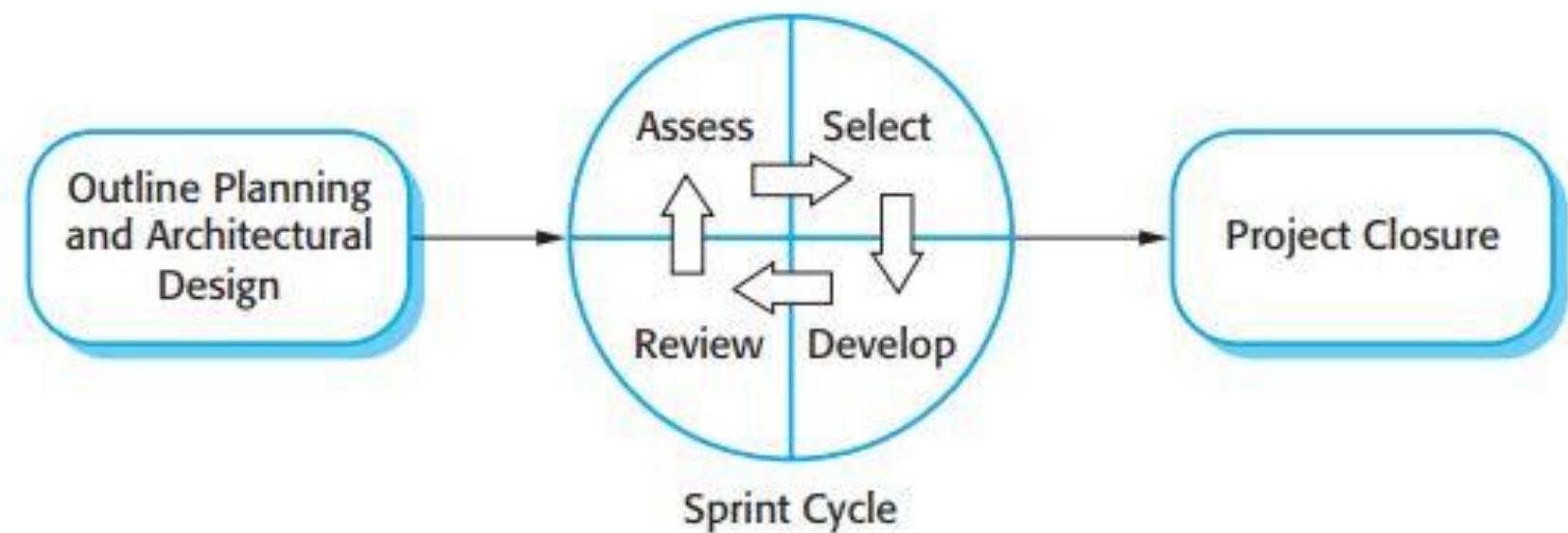
- The principal responsibility of software project managers is to manage the project so that the software is **delivered on time** and within the planned **budget** for the project.
- The **standard approach** to project management is **plan-driven**. Managers draw up a plan for the project showing **what** 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

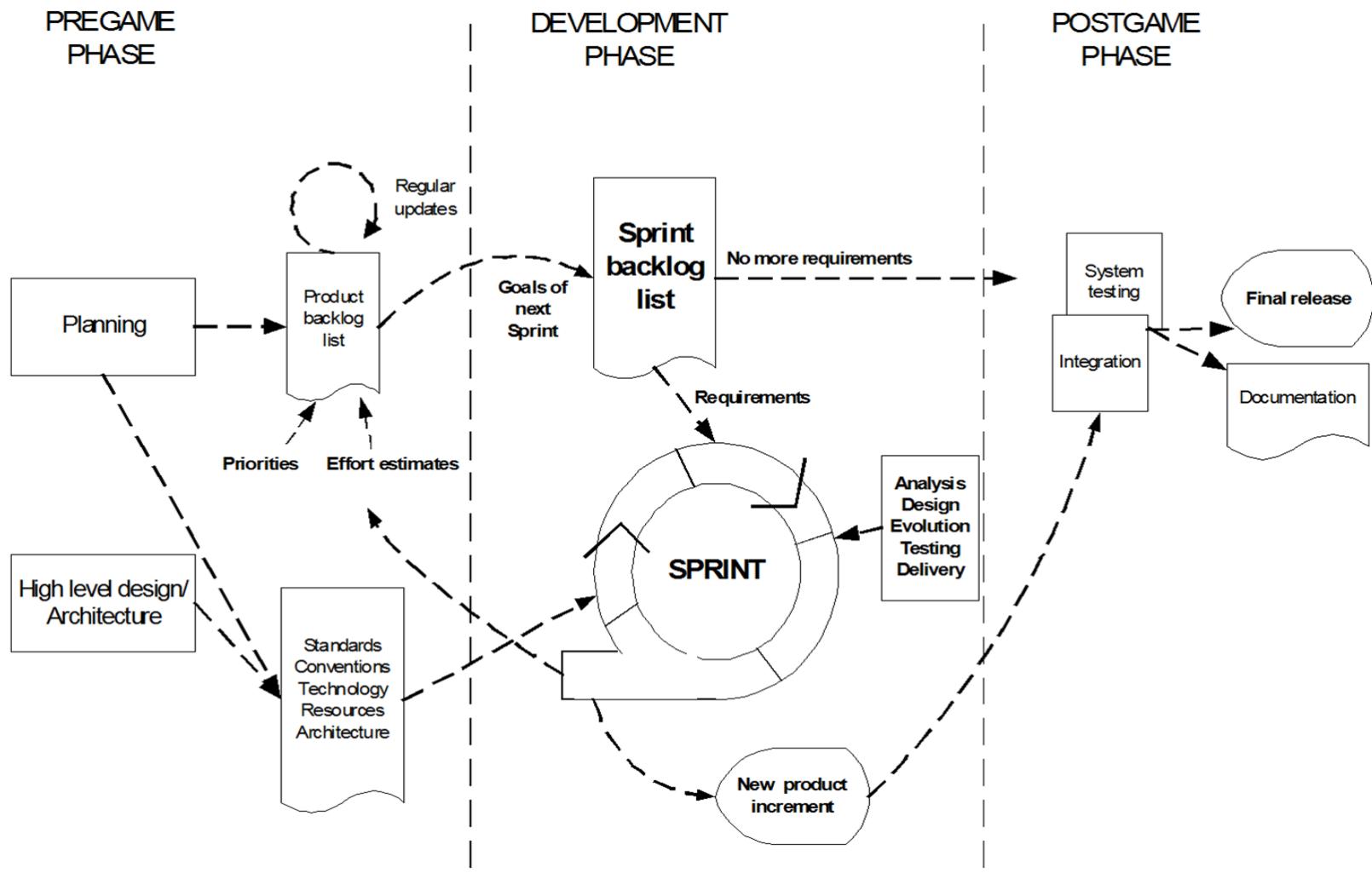
- ❖ Scrum is an agile method that focuses on managing iterative development rather than specific technical approaches to agile software engineering.

- ❖ 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



Scrum Process



Pre-game

- ❖ The pre-game phase includes two sub-phases:
 1. Planning and
 2. Architecture/High level design

Planning

- ❖ Definition of the system being developed.
- ❖ A **Product Backlog** list is created containing all the requirements that are currently known.
- ❖ The requirements are **prioritized** and the **effort** needed for their implementation is estimated.
- ❖ The Product Backlog list is **constantly updated** with new and more detailed items, as well as with more accurate estimations and new priority orders.
- ❖ Planning also includes the definition of the **project team**, **tools** and other **resources**, **risk** assessment and controlling issues, training needs and verification management approval.
- ❖ The customer is closely involved in this process and can introduce new requirements or tasks at the beginning of each sprint

Architecture Design

- ❖ The high level design of the system including the architecture is planned based on the current items in the Product Backlog.
- ❖ In case of an **enhancement** to an existing system, the **changes** needed for implementing the Backlog items are **identified** along with the **problems** they may cause.
- ❖ A **design review meeting** is held to go over the proposals for the implementation and decisions are made on the basis of this review.

Development phase (Game phase)

- ❖ This phase is treated as a "black box" where the unpredictable is expected.
- ❖ The system is developed in Sprints

Sprint

- ❖ Sprints are iterative cycles where the functionality is developed or enhanced to **produce new increments**.
- ❖ Each Sprint **includes the traditional phases** of software development: requirements analysis, design, implementation, evolution and delivery phases.
- ❖ One Sprint is planned to last from one week to one month.

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.
- Once these are agreed, the team organize themselves to develop the software.

The Sprint Cycle

- During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called ‘Scrum master’.
- ‘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.
- At the end of the sprint, the work done is **reviewed** and **presented** to stakeholders. The next sprint cycle then begins.

The Daily Scrum

❖ Parameters

- Daily
- 15 minutes
- Stand-up

❖ Progress update, not status update

❖ These are *not* status updates for the ScrumMaster

- They are commitments in front of peers



Everyone answers 3 questions

1

What progress was done yesterday

2

What is planned for today

3

Any roadblocks or impediments

The Sprint Review

- ✧ Team presents what it accomplished during the sprint
- ✧ Typically takes the form of a demo of new features or underlying architecture
- ✧ Informal
 - 2-hour prep time rule
 - No slides
- ✧ Whole team participates
- ✧ Invite the world



Post-game

- ❖ This phase is entered when an agreement has been made that the environmental variables such as the requirements are completed.
- ❖ In this case, no more items and issues can be found nor can any new ones be invented. The system is now ready for the release and the preparation for this is done during the post-game phase, including the tasks such as the integration, system testing and documentation.

Scrum Terminology (a)

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are 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 Terminology (b)

Scrum term	Definition
Scrum meeting	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
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.

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.

Managing a Digital Land Management System with JIRA

This system aims to digitize land records to improve efficiency, transparency, and reduce corruption in land-related services.

Managing a Digital Land Management System with JIRA

Applying the Scrum Approach

1. Product Backlog Creation:

The team identifies all requirements and divides them into **Epics, Stories, and Tasks** to manage them effectively in a tool like JIRA.

Managing a Digital Land Management System with JIRA

- ✧ **Structure in JIRA:**
- ✧ **Epic:** A large feature or a broader goal in the project.

Epic 1: "Digitization of Land Ownership Records"

Epic 2: "Online Land Mutation Process"

Epic 3: "Citizen Land Query System"

Managing a Digital Land Management System with JIRA

- ✧ **User Stories:** Smaller functionalities or goals derived from the Epics, written from the end-user's perspective.
- ✧ **Epic 1 (Digitization of Land Ownership Records):**
 - **Story 1.1:** "As a landowner, I want to upload my ownership documents so that they are digitized."
 - **Story 1.2:** "As an administrator, I want to verify uploaded documents to ensure accuracy."
- ✧ **Epic 2 (Online Land Mutation Process):**
 - **Story 2.1:** "As a citizen, I want to apply for land mutation online to avoid visiting the land office."
 - **Story 2.2:** "As a mutation officer, I want to track mutation applications to ensure timely approvals."

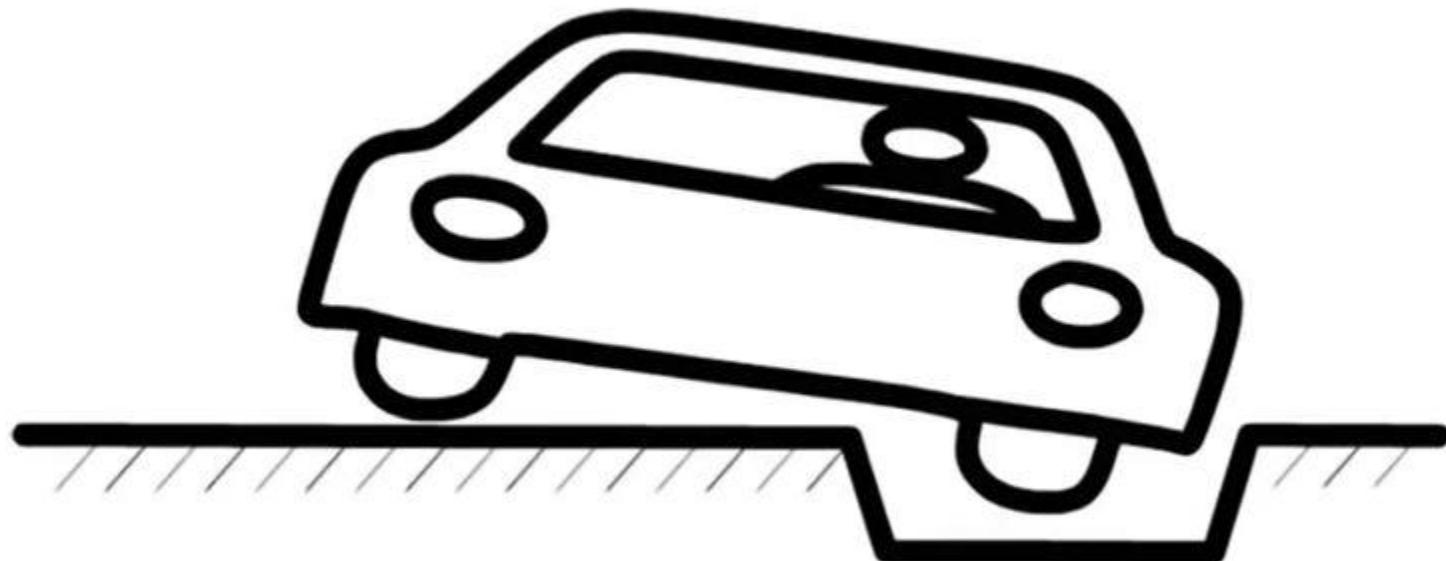
Managing a Digital Land Management System with JIRA

- ❖ **Tasks:** Smaller, actionable items required to complete a story.
- ❖ **For Story 1.1 ("As a landowner, I want to upload my ownership documents so that they are digitized."):**
 - Task 1.1.1: Design the document upload interface.
 - Task 1.1.2: Test the upload feature with sample documents.
 - Task 1.1.3: Implement file storage in a secure database.
 - Task 1.1.4: Develop backend APIs for uploading files.

THANK YOU

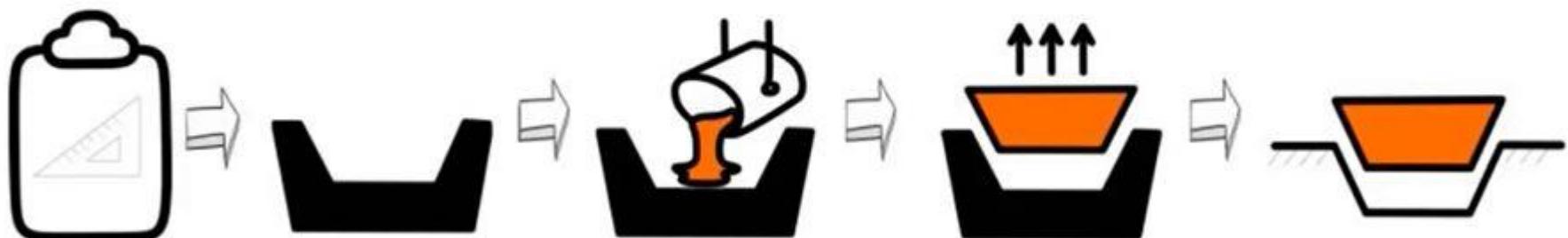
Software Process Models with Examples

Process Models



Process Models

WATERFALL APPROACH



Process Models

ITERATIVE APPROACH

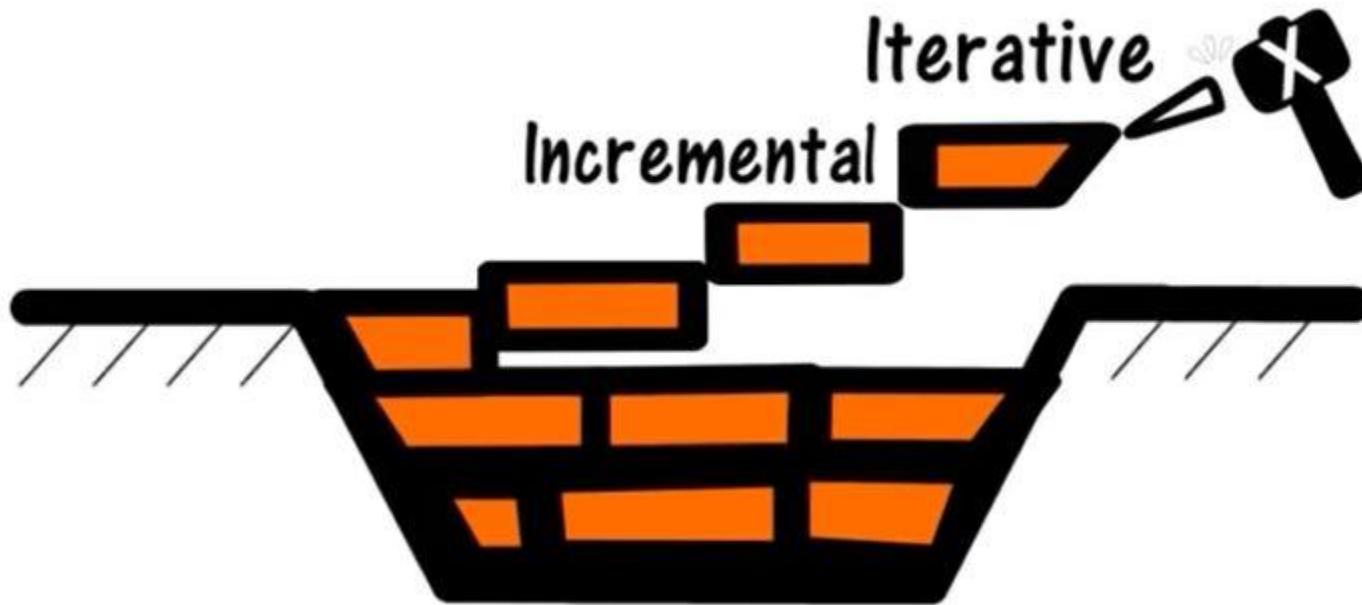


Process Models

INCREMENTAL APPROACH



Process Models

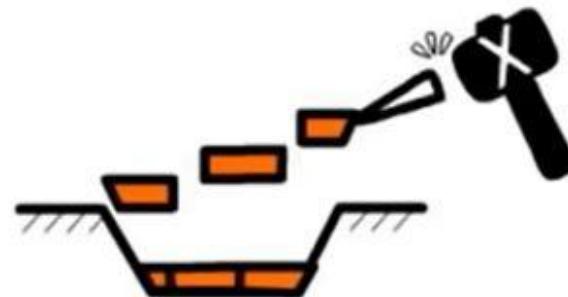


Process Models

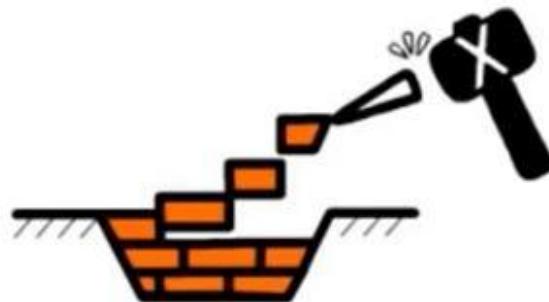
AGILE APPROACH



Increments + Iterations



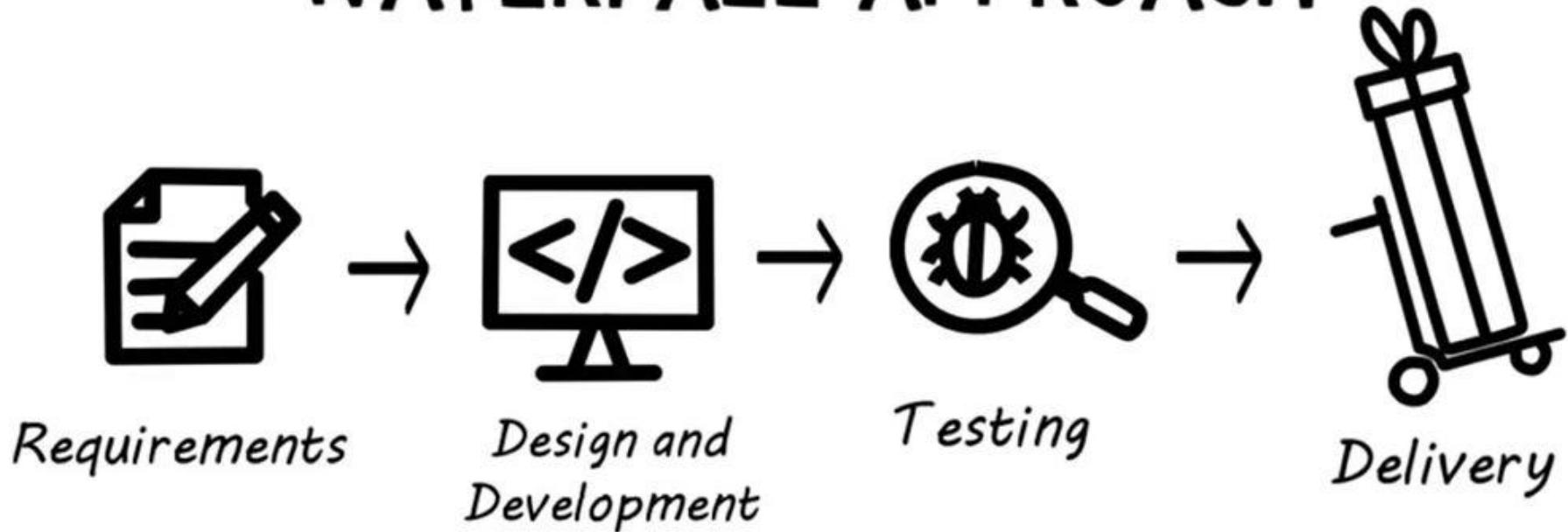
Increments + Iterations



Increments + Iterations

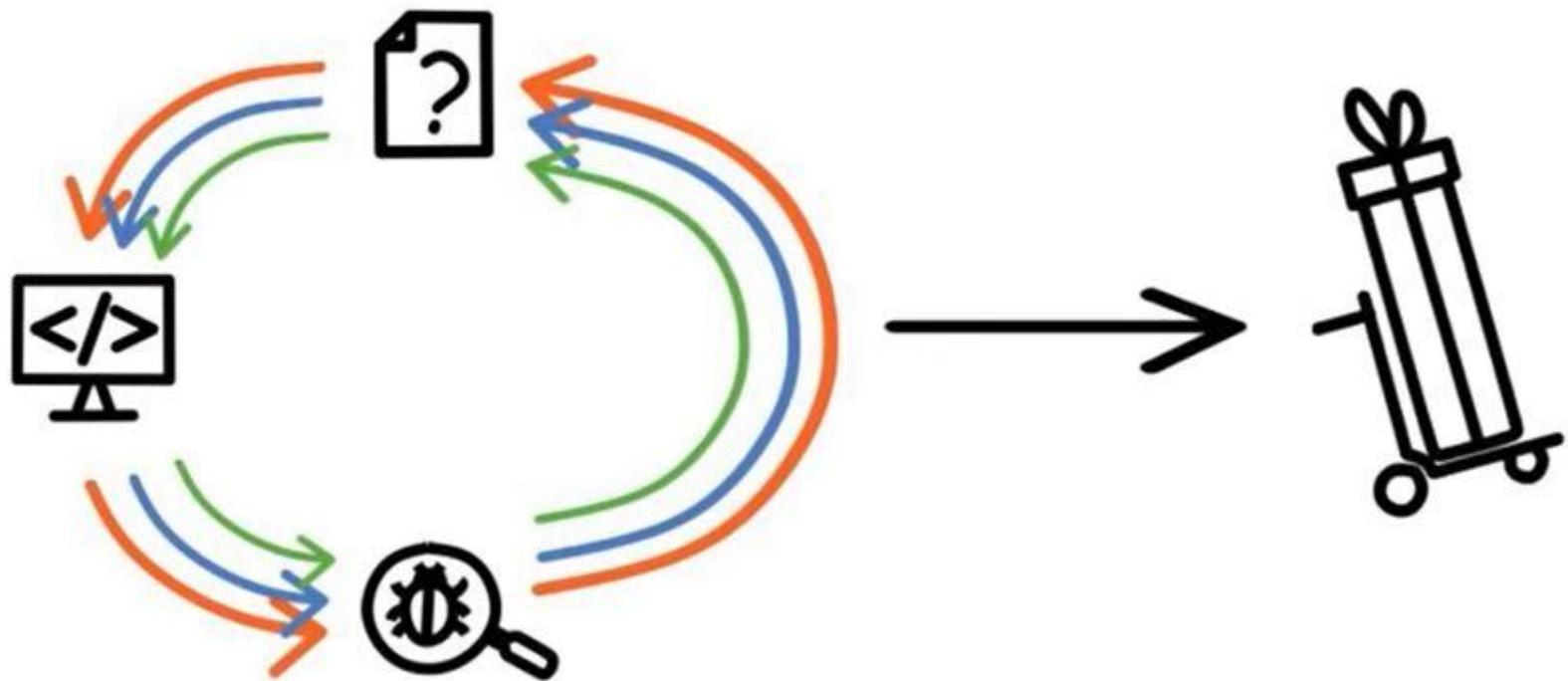
Example of a Web-based Project

WATERFALL APPROACH



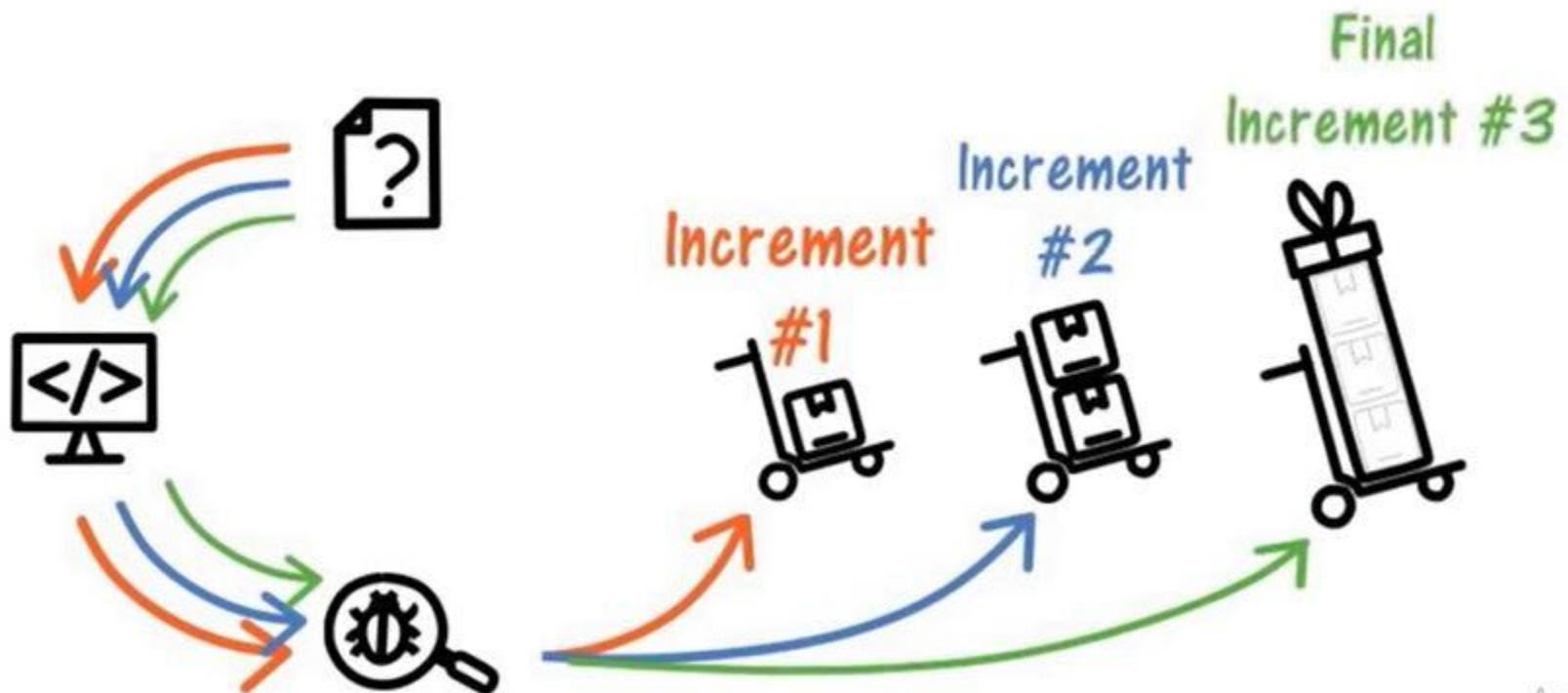
Example of a Web-based Project

ITERATIVE APPROACH

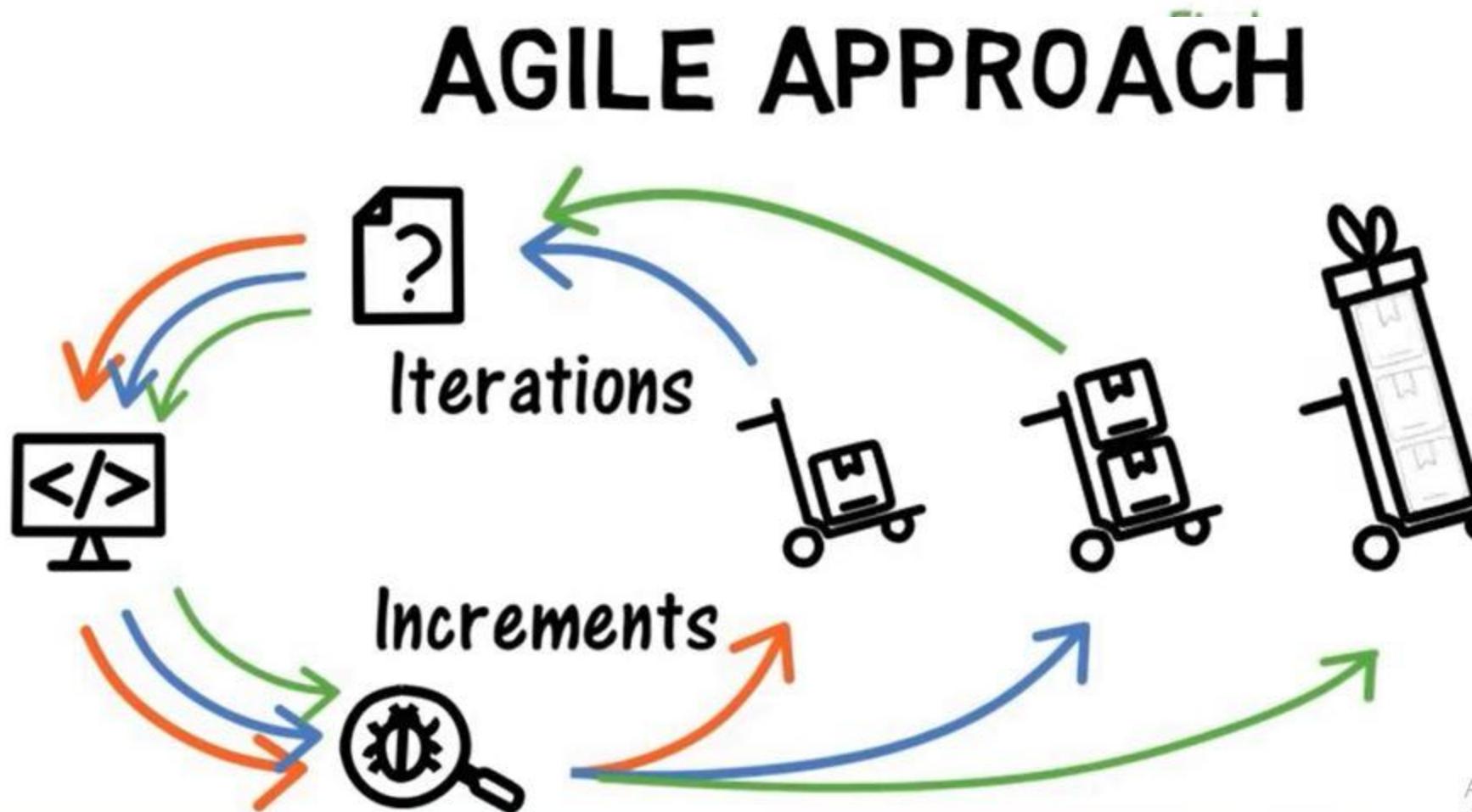


Example of a Web-based Project

INCREMENTAL APPROACH



Example of a Web-based Project



Choosing a Software Process

Changes during the software development process are expensive.

- If a big software system has many interrelated components, avoid major changes to the design of a system during development., such as the modified waterfall model.
- If the market for the software is poorly understood, use a process that gets operational software in front of customers as quickly as possible. Incremental, agile.
- If the requirements are poorly understood, or expected to change, select a process that keeps flexibility. Iterative, agile.

Exercise

Chess Game

It's a well-designed game built to play in a very interactive way where pieces get moved from block to block by the system itself. This project is built to predict (human) moves and accordingly, it takes action. This app can be so satisfying and will make you feel like you're playing in the real world. The best part is that you can even play with friends (remotely) and with the computer as well.

It would require you to set up a MySQL database so that user's details can be stored (including name, score, ranking, etc.).

Exercise

An appropriate software engineering process model would be the Iterative and Incremental Development Model, with a specific focus on Agile methodologies. Here's why this model is suitable for this project:

- **Complexity of Chess Rules:** Chess is a complex game with intricate rules. An **iterative approach** allows you to gradually build and refine the game's logic and rules. You can start with basic functionality and gradually add advanced features, ensuring the correctness of each component before moving on.
- **User-Centric Design:** The description highlights the importance of an interactive and satisfying user experience. Agile methodologies emphasize frequent user feedback and involvement throughout the development process, **allowing you to adapt the game to user preferences** as it evolves.

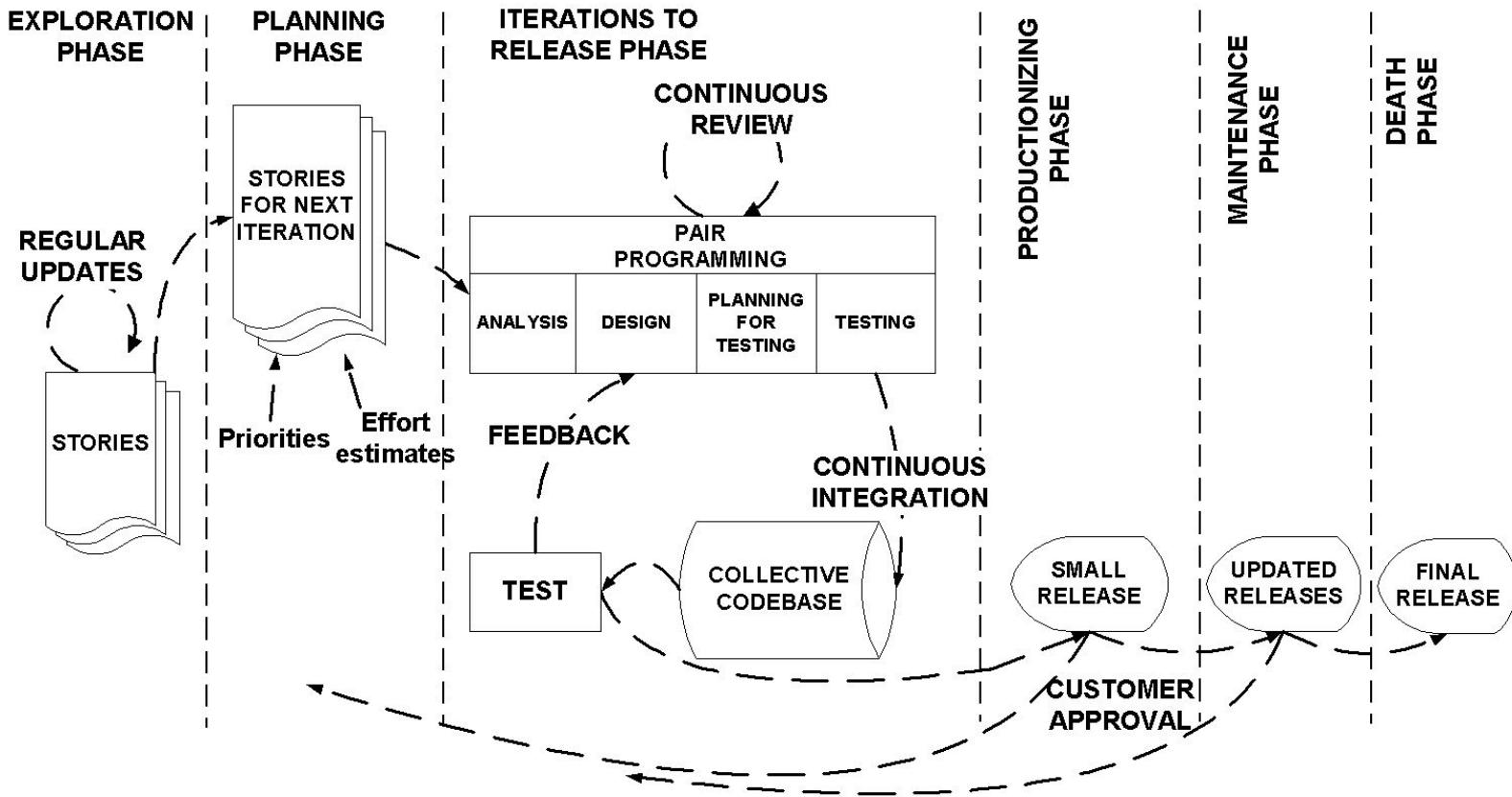
Exercise

Remote Multiplayer Support: The ability to play remotely with friends and computer opponents is a key feature. Agile development encourages **incremental** enhancements, making it easier to integrate and test multiplayer functionalities step by step.

Database Integration: Setting up a MySQL database for storing user details is part of the project. Agile development allows you to prioritize and implement this feature **early and incrementally** while continuously testing and improving it.

Flexibility for Change: Agile methodologies allow for flexibility in adapting to changing requirements or incorporating new features as the project progresses. This is particularly important for games, as user feedback can lead to new ideas and improvements.

The Extreme Programming Process



Architectural Design

Chapter Outline

- Architectural design decisions
- Architectural views
- Architectural patterns (styles)
- Application architectures

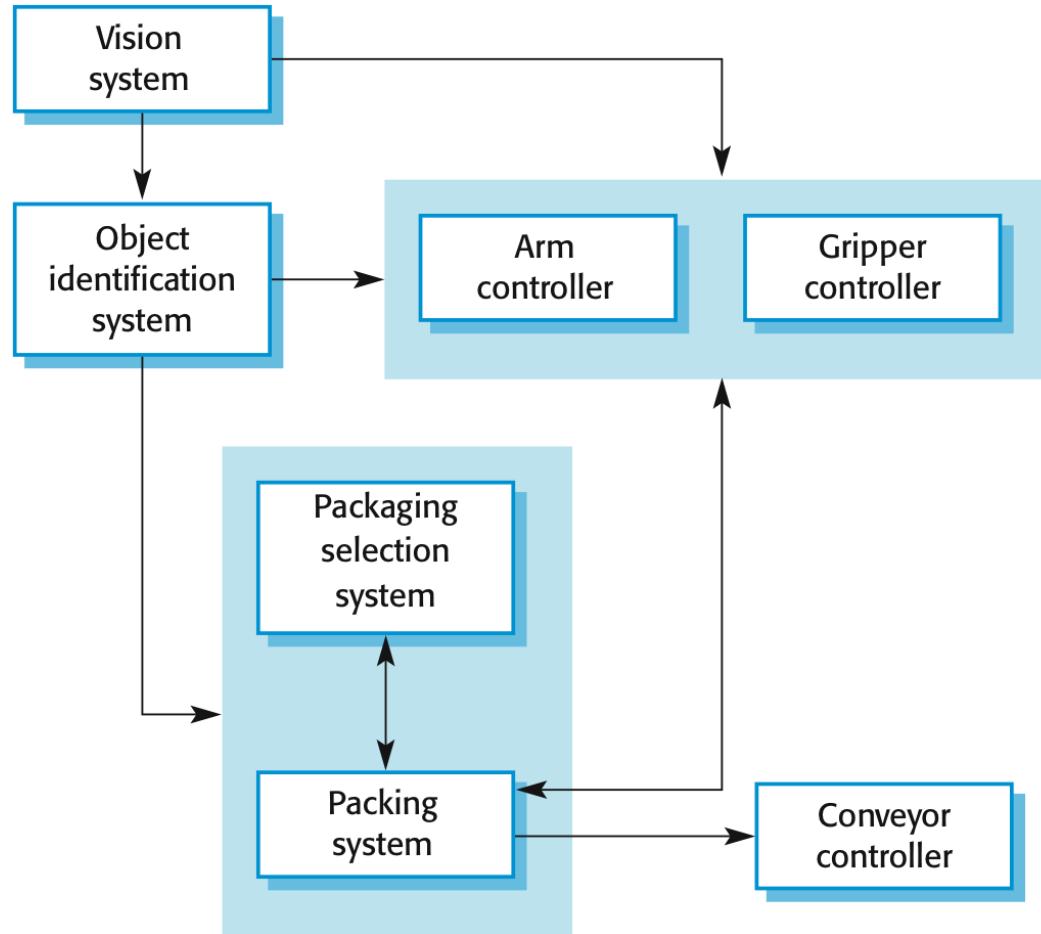
Software Architecture

- ❖ The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is architectural design.
- ❖ The output of this design process is a description of the software architecture.

Architectural Design

- An early stage of the system design process.
- Represents the **link** between requirement engineering and design processes.
- It **identifies** the **main structural components** in a system and the **relationships** between them
- Associates groups of system **functions** or features with **large-scale components** or sub-systems.

The architecture of a packing robot control system



Architectural Abstraction

- Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of explicit architecture

✧ Stakeholder communication

- Architecture may be used as a focus of discussion by system stakeholders.

✧ System analysis

- Means that analysis of whether the system can meet its non-functional requirements is possible.

✧ Large-scale reuse

- The architecture may be reusable across a range of systems
- Product-line architectures may be developed.

Architectural representations

- ❖ Simple, informal **block diagrams** showing *components* and *relationships* are the most frequently used method for documenting software architectures.
 - Each box represents a component
 - Boxes within boxes indicate that the component has been decomposed to sub-components
 - Arrows mean that data and control signals are passed from component to component in the direction of the arrows
- ❖ But these have been **criticized** because they lack semantics, do not show the **types** of relationships between components nor the externally visible properties of the components.

Use of Architectural Models

- ✧ As a way of **facilitating discussion** about the system design
 - A high-level architectural view of a system is useful for **communication** with system stakeholders and **project planning** because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- ✧ As a way of **documenting** an architecture that has been designed
 - The aim here is to produce a complete system model that shows the different **components** in a system, their **interfaces** and their **connections**.

Architectural Design Decisions

Architectural Design Decisions

- ✧ Architectural design is a creative process. The activities within the process depends on:
 - type of system being developed,
 - background
 - experience of the system architect, and the
 - specific requirements for the system .
- ✧ However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

Architectural Design Decisions

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Architecture Reuse

- Systems in the **same domain** often have similar architectures that reflect domain concepts.
 - Example: **Application product lines** are built around a core architecture with variants that satisfy particular customer requirements.

- The architecture of a system may be designed around one of more architectural **patterns** or **styles**.
 - Find common patterns, their strengths and weaknesses
 - Choose the most appropriate structure, such as client–server or layered structuring
 - Decide a strategy for decomposing components into sub-components
 - Finally, make decisions about how the execution of components is controlled.

Architecture and system characteristics

❖ Performance

- Localize critical operations and minimize communications. Use large rather than fine-grain components.

❖ Security

- Use a layered architecture with critical assets in the inner layers.

❖ Safety

- Localize safety-critical features in a small number of sub-systems.

❖ Availability

- Include redundant components and mechanisms for fault tolerance.

❖ Maintainability

- Use fine-grain, replaceable components.

Architectural Views

Architectural Views

Issues related to architectural design:

- ❖ What **views** or **perspectives** are useful when designing and documenting a system's architecture?
- ❖ What **notations** should be used for describing architectural models?
- ❖ Each architectural model only shows one view or perspective of the system
 - It might show how a system is **decomposed into modules**, how the **run-time processes interact** or the different ways in which **system components are distributed** across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

4 + 1 View Model of Software Architecture

✧ Logical view

- shows the key abstractions in the system as objects or object classes.

✧ Process view

- shows how, at run-time, the system is composed of interacting processes.

✧ Implementation view

- shows how the software is decomposed for development.

✧ Deployment (physical) view

- shows the system hardware and how software components are distributed across the processors in the system.

✧ All the view above related through a user view (+1)

4+1 Views Architecture

The “4 + 1 views” architecture, Fig. 1.13 [Arlow & Neustadt 2005]

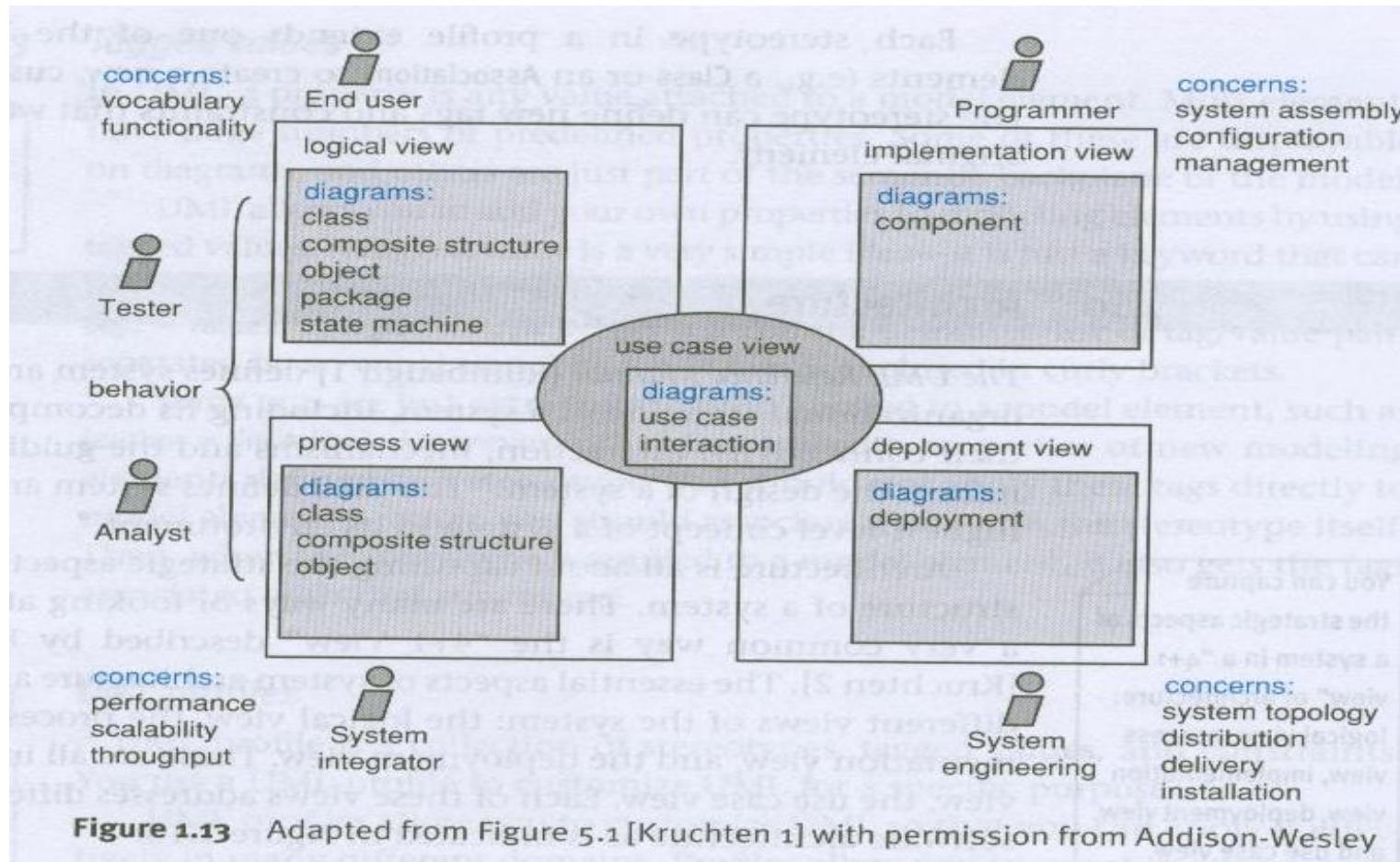


Figure 1.13 Adapted from Figure 5.1 [Kruchten 1] with permission from Addison-Wesley

Architectural Patterns

Architectural Patterns

- ❖ Patterns are a means of representing, sharing and reusing knowledge about software systems .
- ❖ The architectural pattern captures the design structures of various systems and elements of software so that they can be reused
- ❖ Patterns should include information about when they are and when they are not useful.
- ❖ Patterns may be represented using tabular and graphical descriptions.

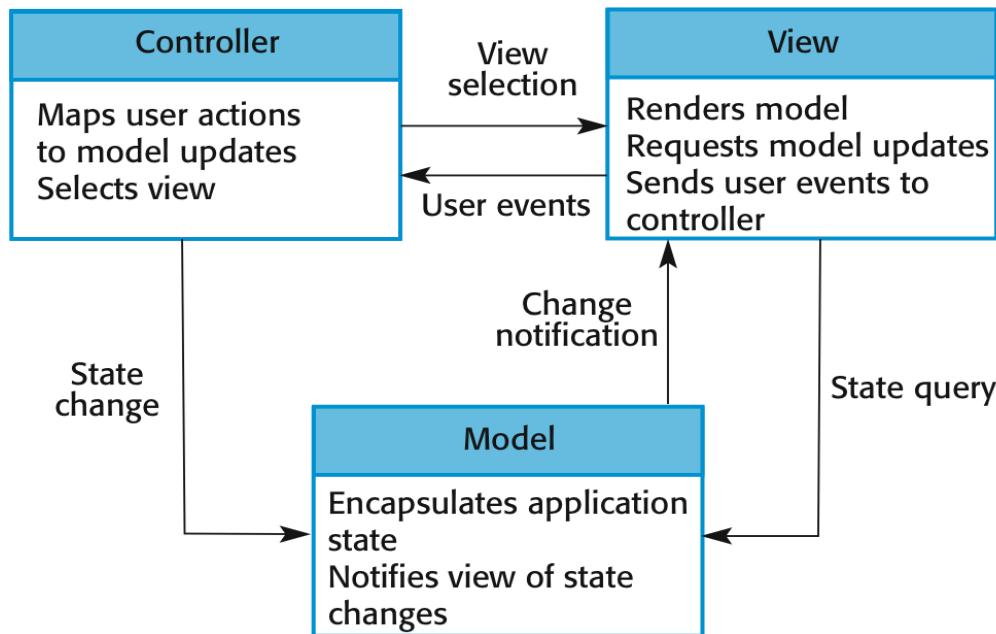
Different Architectural Patterns

1. Model-View-Controller (MVC) pattern
2. Layered Architecture
3. Repository Architecture
4. Client-Server Architecture
5. Pipe and Filter Architecture

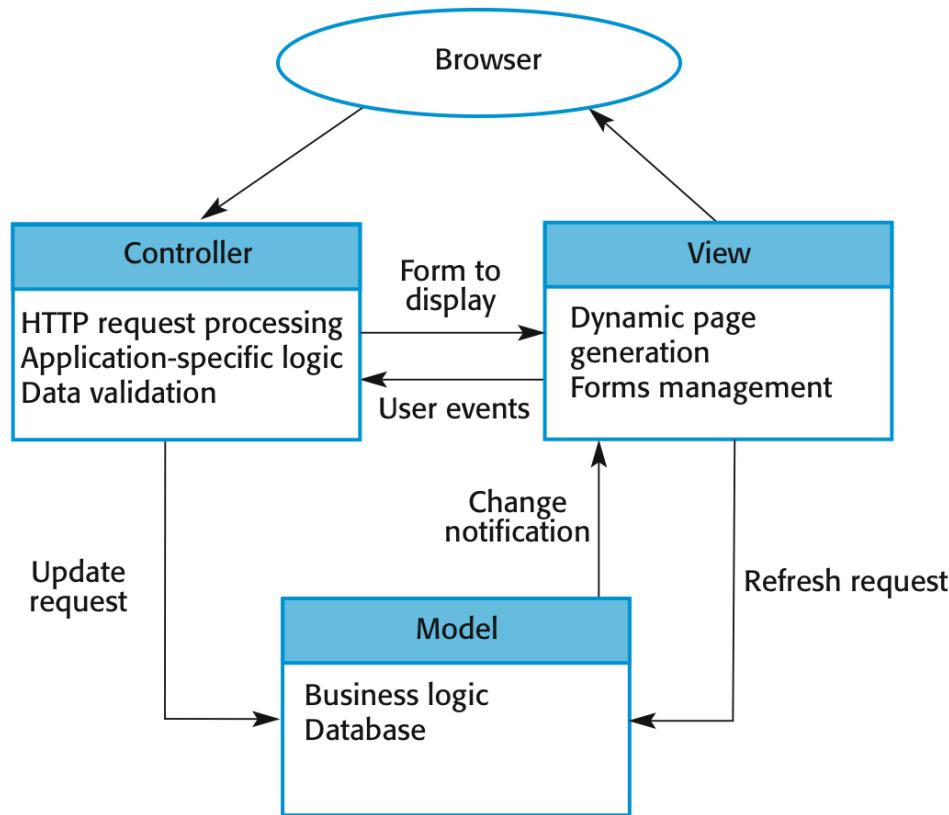
The Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown .
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

The organization of the Model-View-Controller



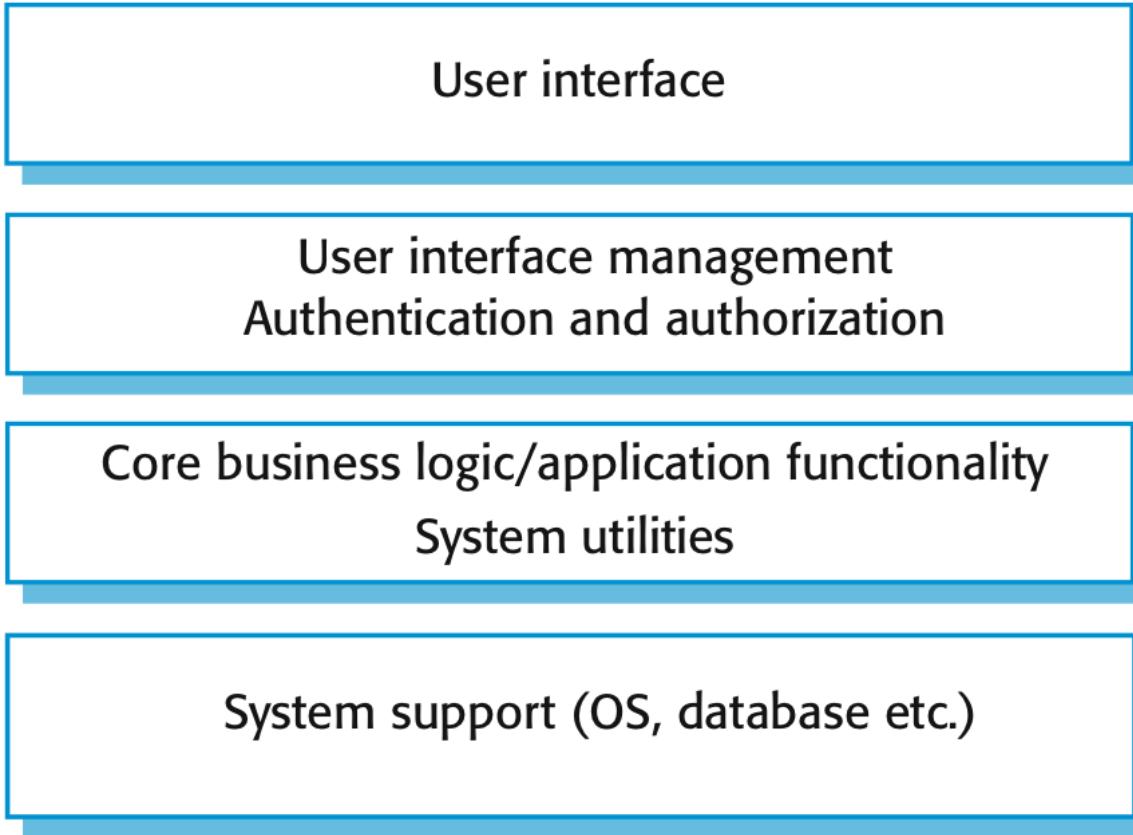
Web application architecture using the MVC pattern



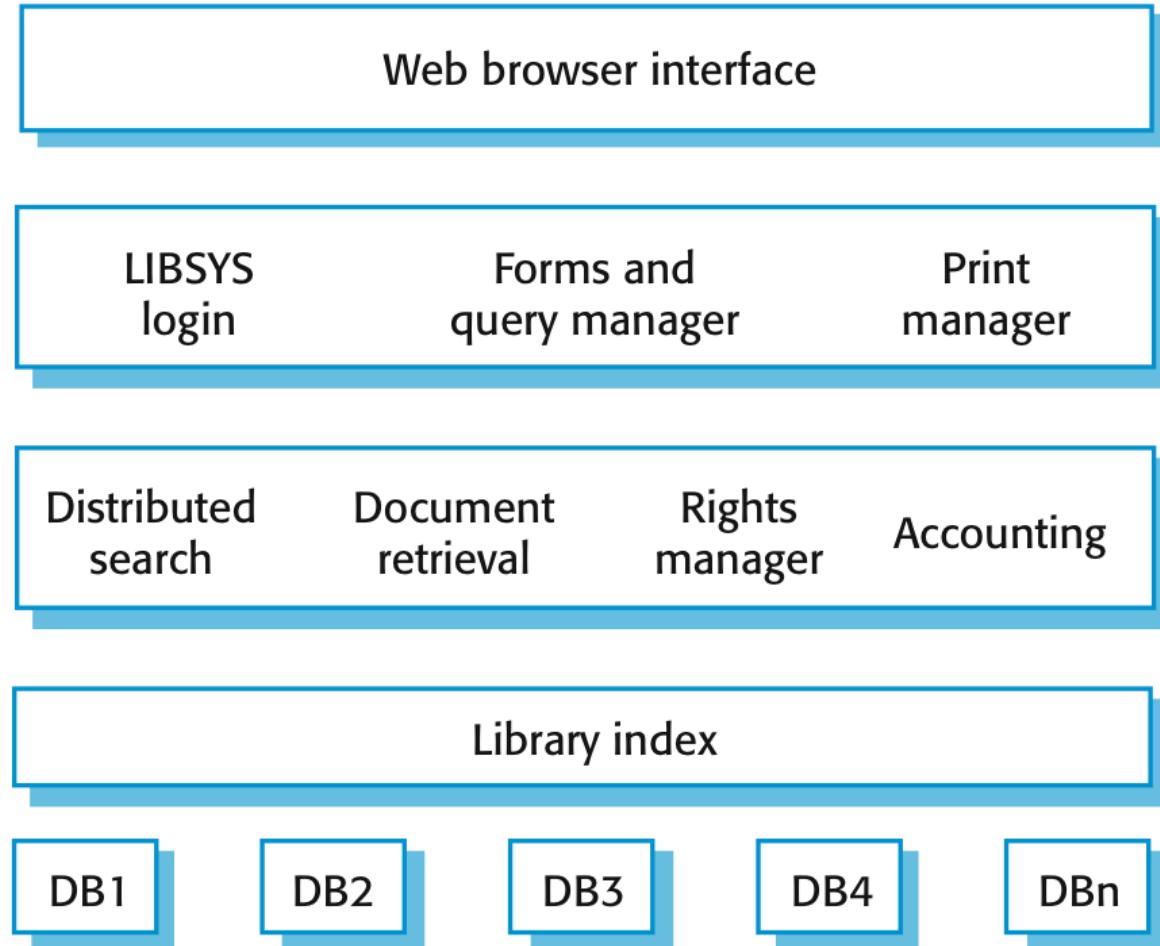
Layered Architecture

- Organizes the system into **layers** with related functionality associated with each layer.
- A layer provides services to the layer above it. The lowest-level layers represent **core services** that are likely to be used throughout the system
- Supports the incremental development of sub-systems in different layers.
- The architecture is **changeable** and **portable**.
- When to use
 - when building new facilities on top of existing systems;
 - when the development is spread across several teams
 - when there is a requirement for multi-level security.

A generic layered architecture



The architecture of the LIBSYS system



Advantages & Disadvantages

✧ Advantages

- Allows **replacement** of entire layers so long as the interface is maintained.
- **Redundant** facilities (e.g., authentication) can be provided in each layer to increase the **dependability** of the system

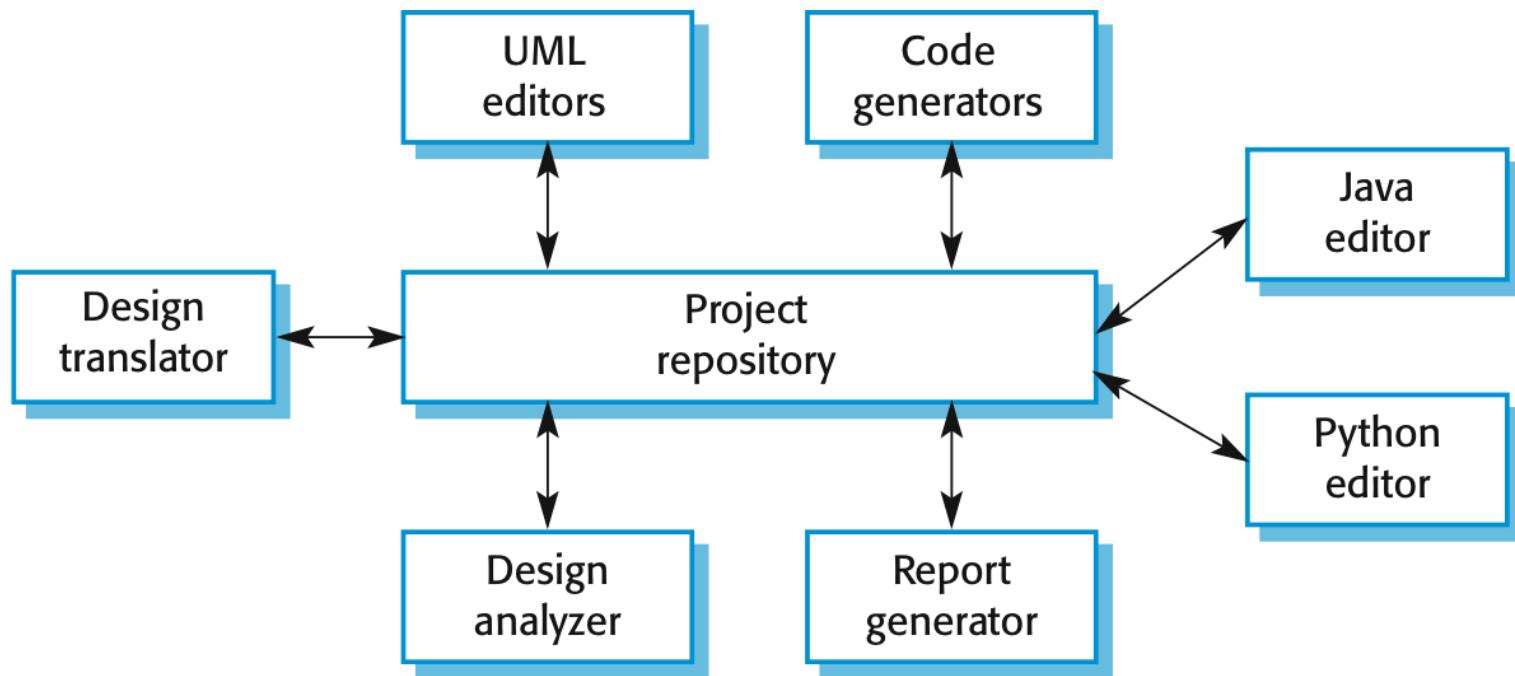
✧ Disadvantages

- Providing a **clean separation** between layers is often difficult
- A high-level layer may have to **interact directly** with lower-level layers rather than through the layer immediately below it
- **Performance** can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

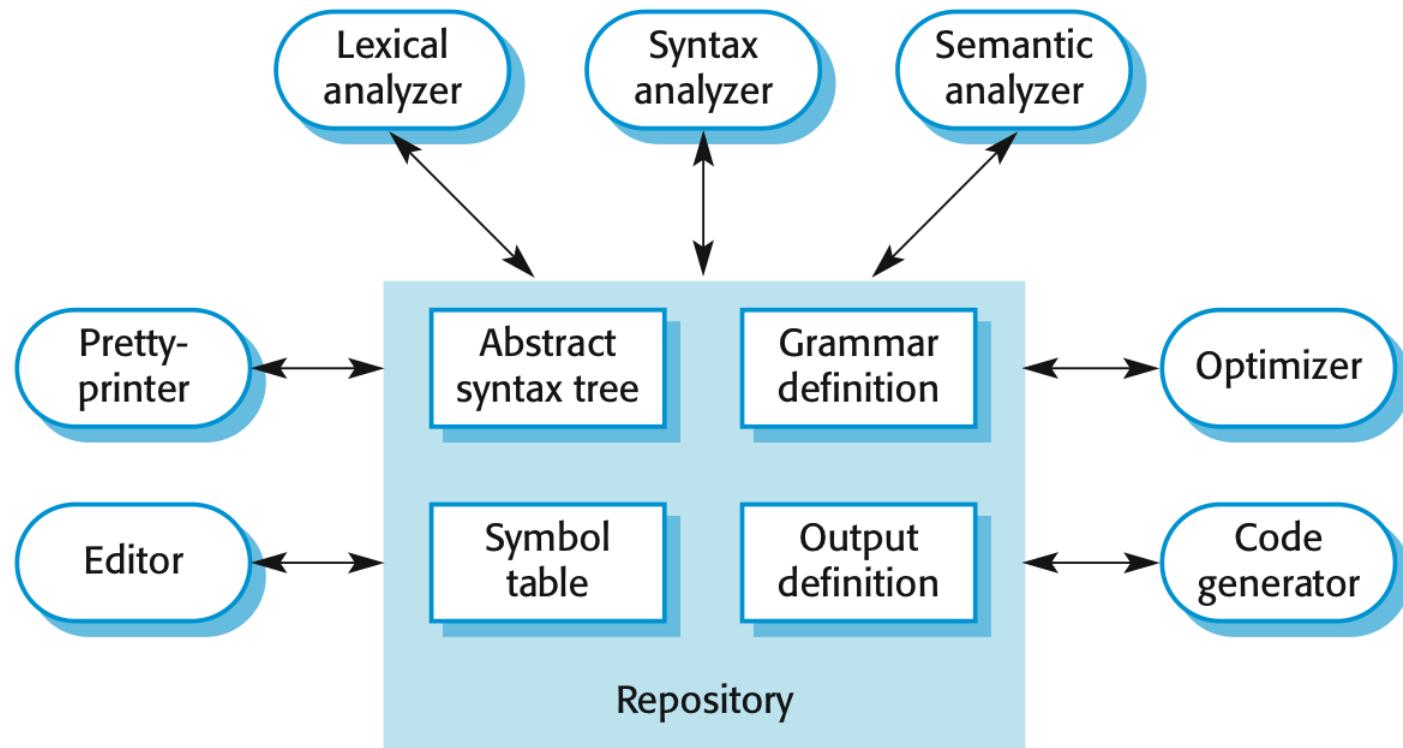
Repository Architecture

- Describes how a set of interacting components can **share data**
- All data in a system is managed in a **central repository** that is accessible to all system **components**
- **Examples:** command and control systems, language processing system, interactive development environments (IDE) for software etc.
- When to use
 - systems in which **large volumes of information** are generated that has to be stored for a long time
 - systems where data is generated by one component and used by another

A repository architecture for an IDE



A repository architecture for a language processing system



Advantages & Disadvantages

✧ Advantages

- Components can be **independent**.
- **Changes** made by one component can be **propagated** to all components.

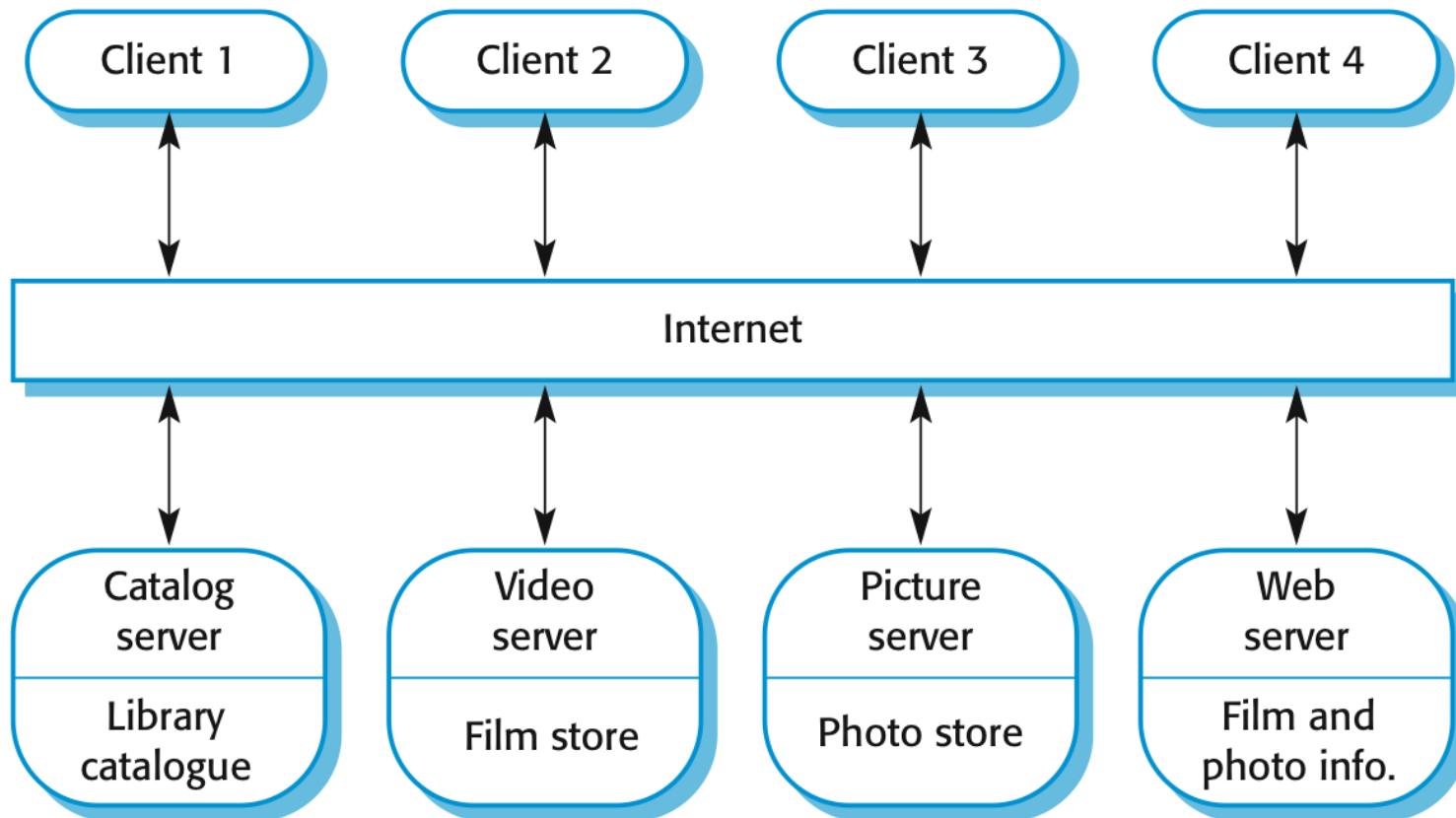
✧ Disadvantages

- The repository is a single point of failure.
- Distributing the repository across several computers may be difficult.

Client-Server Architecture

- The functionality of the system is organized into **services**, with each service delivered from a separate server
- Set of **clients** which call on these services.
- Set of stand-alone **servers** which provide specific services such as printing, data management, etc.
- **Network** which allows clients to access servers.
- **Distributed system** model which shows how data and processing is distributed across a range of components.
- When to use
 - data in a **shared database** has to be **accessed** from a **range of locations**

A client–server architecture for a film library



Client-Server Architecture

✧ Advantages

- Servers can be **distributed** across a network.
- Easy to **add** a new server and **integrate** it with the rest of the system or to upgrade servers

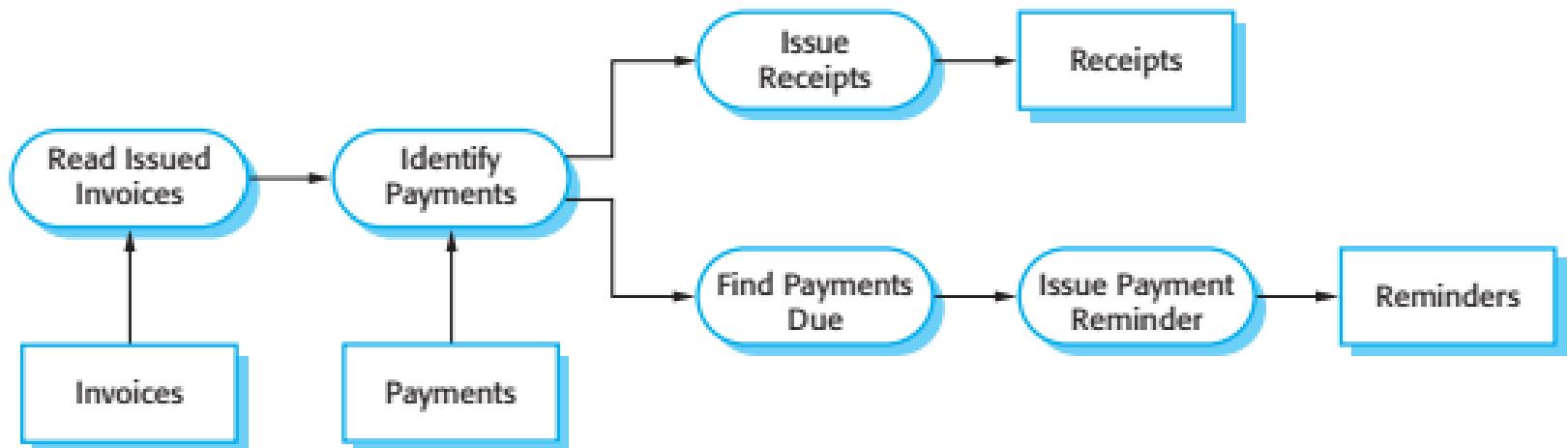
✧ Disadvantages

- Each service is a single point of **failure**
- **Performance** may be **unpredictable** because it depends on the network
- Management problems

Pipe and Filter Architecture

- Functional transformations process their inputs to produce outputs.
- Example: graphics processing, invoicing system etc.
- Data can be processed by each transform item by item or in a single batch.
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.
- When to use
 - for data processing applications

Pipe and Filter Architecture



Advantages & Disadvantages

✧ Advantages

- Easy to understand and supports transformation **reuse**.
- **Workflow** style matches the structure of many business processes.
- **Evolution** by adding transformations is straightforward.
- Can be implemented as either a **sequential** or **concurrent** system.

✧ Disadvantages

- The **format** for data transfer **has to be agreed** upon between communicating transformation. This increases system **overhead** and **reusability difficult**.

Application Architectures

Application Architectures

- Application systems are intended to meet a business or organizational need.
- All businesses have much in **common**. Their application systems also tend to have **common architecture** reflecting their requirements.
- The application architecture may be re-implemented when developing new systems but, for many business systems, application reuse is possible without re-implementation
- A generic system architecture is configured and adapted to create a specific business application.

Use of Application Architectures

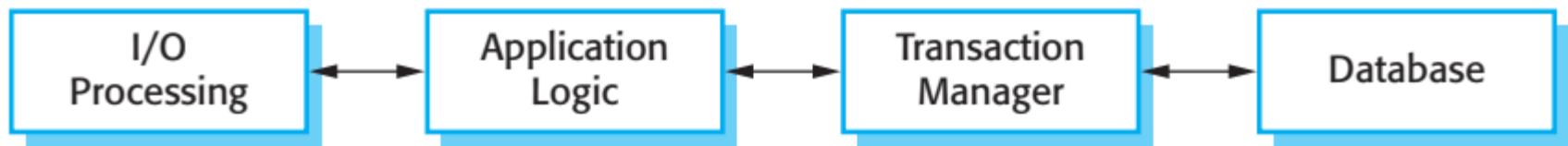
1. As a starting point for the architectural design process
2. As a design checklist.
3. As a way of organizing the work of the development team
4. As a means of assessing components for reuse
5. As a vocabulary for talking about types of applications

Examples of Application Types

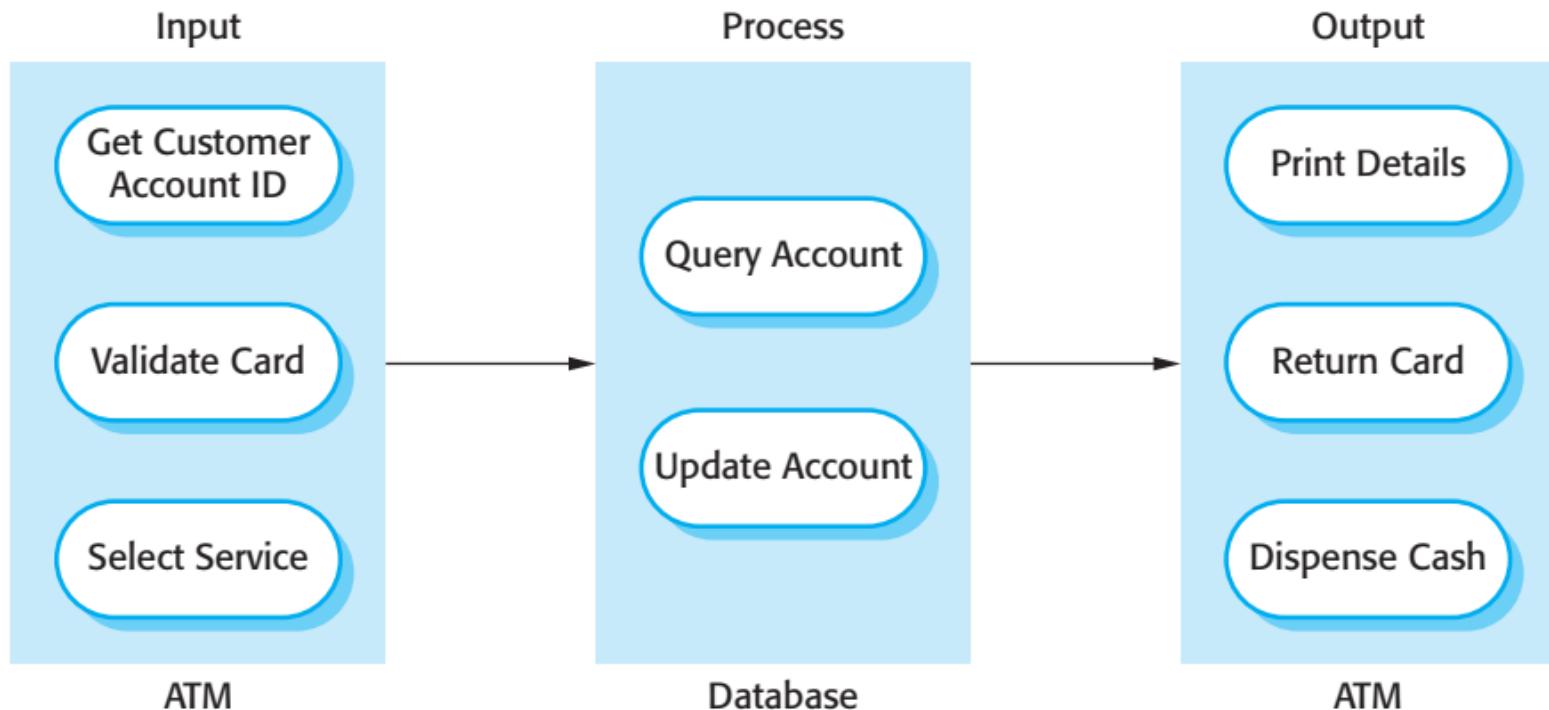
Two very widely used generic type applications are:

- ❖ Transaction processing applications
 - database-centered applications
 - process user requests for information and update the information in a database.
 - **Example:** e-commerce systems, booking systems
- ❖ Language processing systems
 - user's intentions are expressed in a formal language
 - **Example :** Compilers, Command Interpreters

The Structure of Transaction processing applications



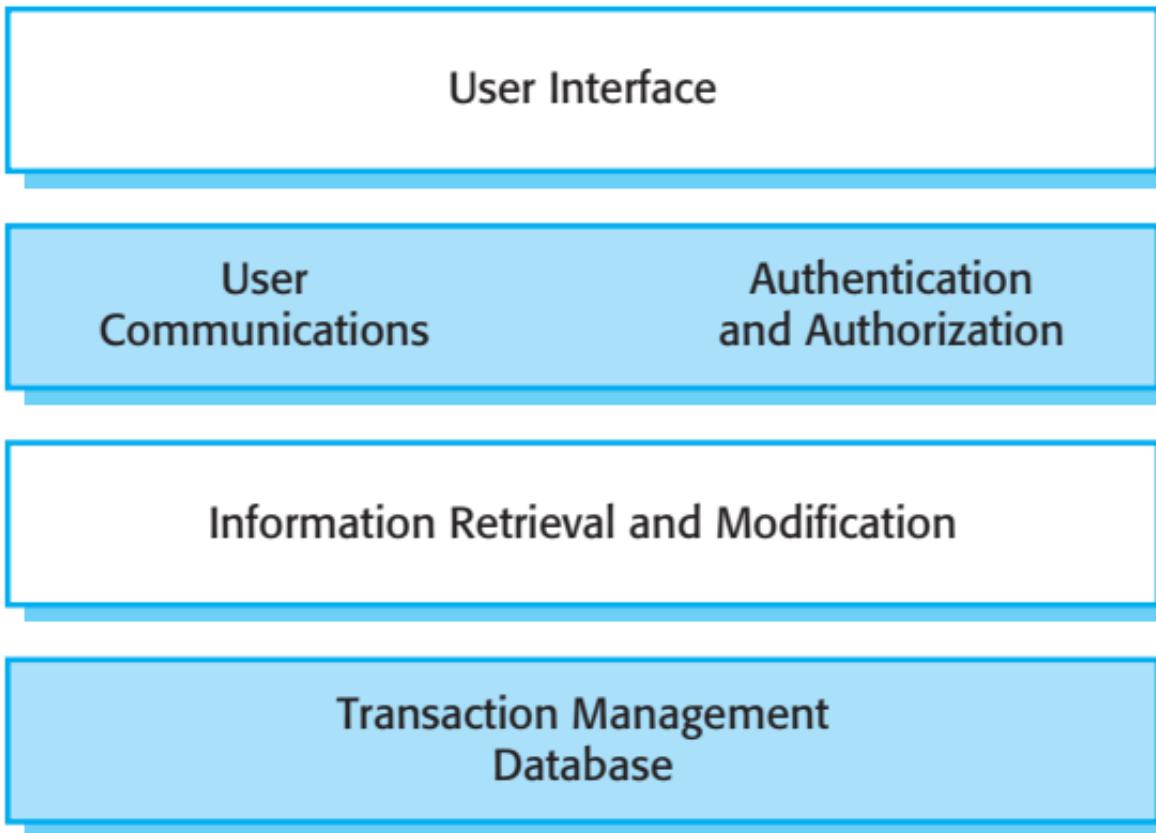
The Software Architecture of an ATM system



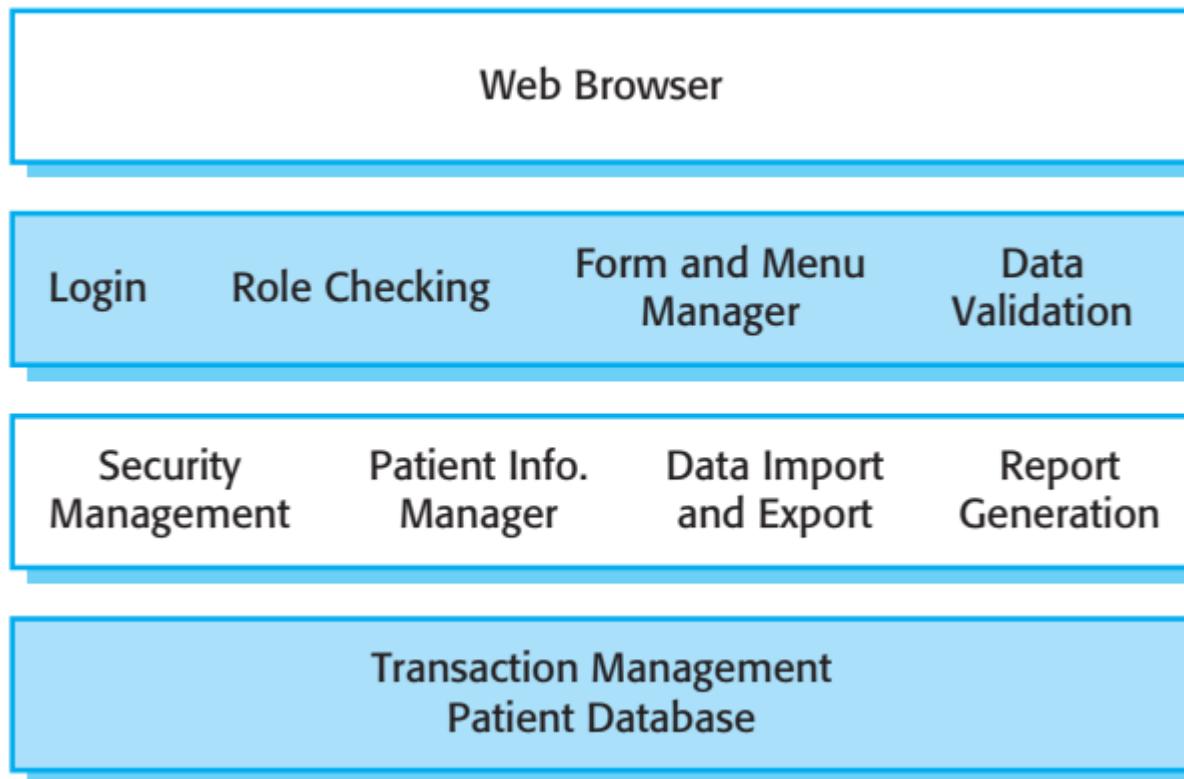
Information Systems Architecture

- An information system allows controlled access to a **large base of information**
- Information Systems have a **generic** architecture that can be organized as a layered architecture
- **Example:** library catalog, flight timetable, the records of patients in a hospital

Layered Information System Architecture



The architecture of the MHC-PMS



Class Work

Draw an architectural model for Temperature Monitoring System

Software Testing

Referred Books

- Software Engineering
 - Ian Sommerville
- Introduction To Software Testing
 - Jeff Offutt

Software Testing

- ❖ Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- ❖ To execute a program, we use **artificial data**.
- ❖ Testing can only reveal the presence of errors, not their absence

Testing Goals

- ❖ To demonstrate to the developer and the customer that the software **meets** its requirements.
 - Custom software testing
 - Generic software testing
- ❖ To **discover** situations in which the behavior of the software is **incorrect, undesirable**, or does **not conform** to its specification
 - Defect Testing: system crashes, unwanted interactions with other systems, incorrect computations

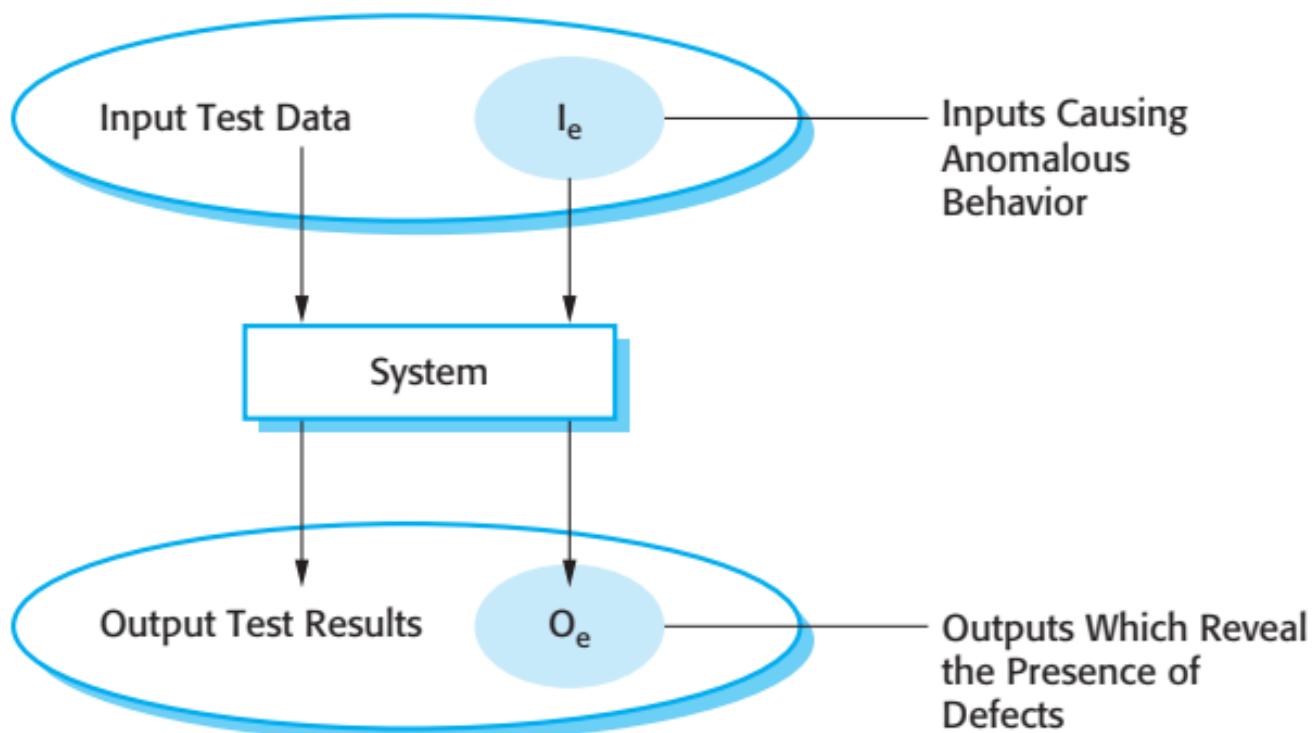
Validation and Defect Testing

- The first goal leads to validation testing
 - The test cases are designed to demonstrate that the software works as intended
 - A successful test: the system performs correctly
- The second goal leads to defect testing
 - The test cases are designed to expose defects
 - A successful test: makes the system perform incorrectly and expose faults

A Sample Test Case

TC#	Test Case	Test Data	Test Steps	Expected Result
1.	Verify Login	Id: test pass: 1234	1. Go to Login page 2. Enter UserId 3. Enter Password 4. Click Login	Login Successful
2.	Verify Login	Id: \$@#d pass: 1234	1. Go to Login page 2. Enter UserId 3. Enter Password 4. Click Login	Login Unsuccessful

An input-output model of program testing



Testing Goals

- ❖ Immediate Goals
- ❖ Long-term Goals
- ❖ Post-Implementation Goals



[https://www.geeksforgeeks.org/
goals-of-software-testing/](https://www.geeksforgeeks.org/goals-of-software-testing/)

Fig : Software Testing Goals

Software Testing Terminology

Software Testing Terminology

- **Verification:** The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase.
- **Validation:** The process of evaluating software at the end of software development to ensure compliance with intended usage.

Verification Vs. Validation

	Verification	Validation
1.	Are we building the product right?	Are we building the right product?
2.	Checks whether an artifact conforms to its previous artifact	Checks the final product against specification
3.	Done by developer	Done by Tester
4.	Involves reviews, inspections, unit testing and integration testing	Involves system testing
5.	Comprises of both Static and Dynamic procedures	Comprises of only Dynamic procedures

Verification and Validation

- The ultimate **goal** of verification and validation processes is to establish **confidence** that the software system is '**fit for purpose**'.
- The **level** of required **confidence** depends on:
 - **Software purpose**
 - how critical the software is to the organization
 - **User expectations**
 - users have low expectations of certain kinds of software
 - **Marketing environment**
 - in a competitive environment, users may be willing to tolerate a lower level of reliability for cheap products

Software Faults, Errors & Failures

- ✧ Software Fault : A static defect in the software
- ✧ Software Error : An incorrect internal state that is the manifestation of some fault
- ✧ Software Failure : External, incorrect behavior with respect to the requirements or other description of the expected behavior

Software Faults, Errors & Failures

- A patient gives a doctor a list of symptoms
 - Failures
- The doctor tries to diagnose the root cause, the ailment
 - Fault
- The doctor may look for anomalous internal conditions (high blood pressure, irregular heartbeat, bacteria in the bloodstream)
 - Errors

A Concrete Example

```
public static int numZero (int [ ] arr)
{ // Effects: If arr is null throw NullPointerException
// else return the number of occurrences of 0 in arr
int count = 0;
for (int i = 1; i < arr.length; i++)
{
    if (arr [ i ] == 0)
    {
        count++;
    }
}
return count;
```

Test 1
[2, 7, 0]
Expected: 1
Actual: ?

Test 2
[0, 2, 7]
Expected: 1
Actual: ?

A Concrete Example

```
public static int numZero (int [ ] arr)
{ // Effects: If arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
  int count = 0;
  for (int i = 1; i < arr.length; i++)
  {
    if (arr [ i ] == 0)
    {
      count++;
    }
  }
  return count;
}
```

Fault: Should start searching at 0, not 1

Test 1
[2, 7, 0]
Expected: 1
Actual: 1

Error: i is 1, not 0,
on the first iteration
Failure: none

Test 2
[0, 2, 7]
Expected: 1
Actual: 0

Error: i is 1, not 0
Error propagates to the variable count
Failure: count is 0 at the return statement

Static Testing vs Dynamic Testing

Differences between Static and Dynamic Testing

- ❖ Static testing finds defects in work products **directly** rather than identifying **failures** caused by defects when the software is run.
- ❖ Static testing can be used to improve **internal quality** of the software, while dynamic testing focuses on **externally visible behaviors**.

Static Testing (Inspection)

- ✧ Inspections involve a team of people reading or visually inspecting a program
- ✧ These are so-called ‘**static**’ V & V technique
- ✧ Do not require execution of a system
- ✧ Focuses on **static system representations** such as requirements, design model, source code etc. to discover defects

Inspection (Cont.)

General Procedure:

❖ Inspection Team:

- Moderator
- Programer
- Program's Designer
- Test Specialist

❖ Moderator's Duties:

- Distributing materials for, and scheduling, the inspection session.
- Leading the session.
- Recording all errors found.
- Ensuring that the errors are subsequently corrected

Inspection (Cont.)

✧ Inspection Agenda:

- The programmer narrates, statement by statement, the logic of the program. And other participants raise questions
- The program is analyzed with respect to checklists of historically common programming errors

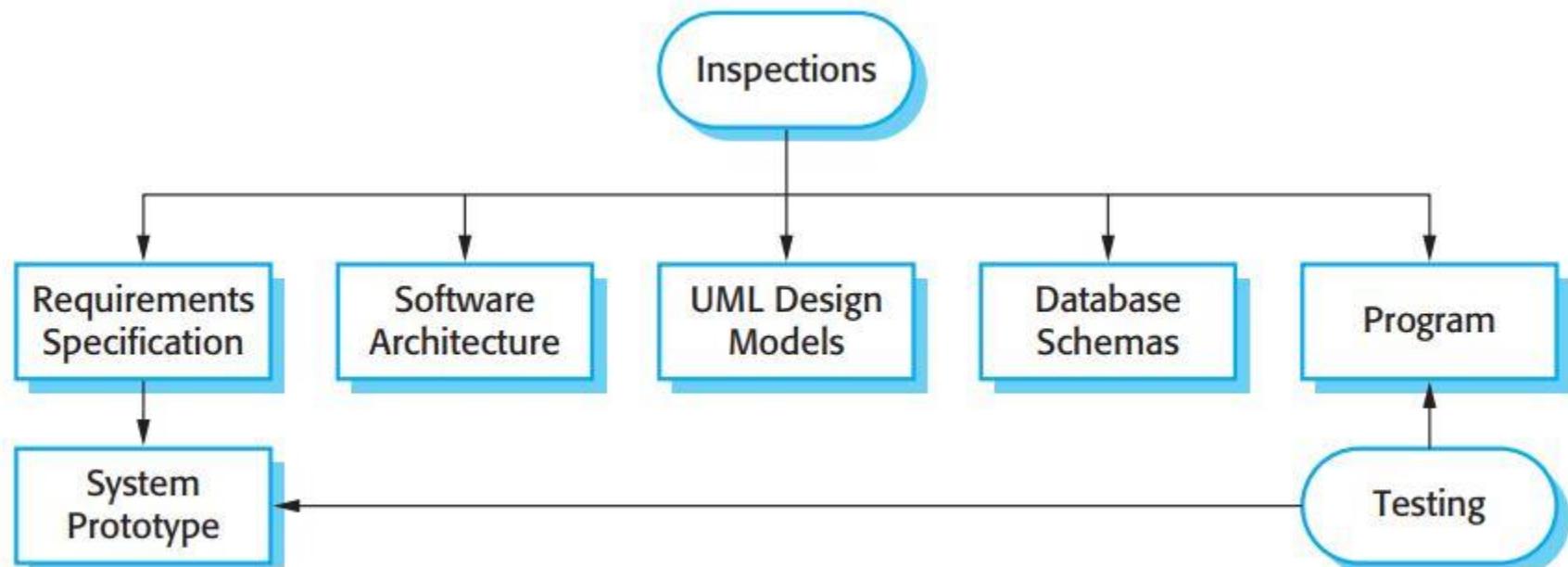
✧ Human Agenda:

- The **testing group** must adopt an appropriate attitude to make the inspection process effective
- The **programmer** must take the process in a positive and constructive way

Dynamic Testing (Software Testing)

- Dynamic V &V technique
- Focuses on executing the implementation of the system with test data and observes software behavior.

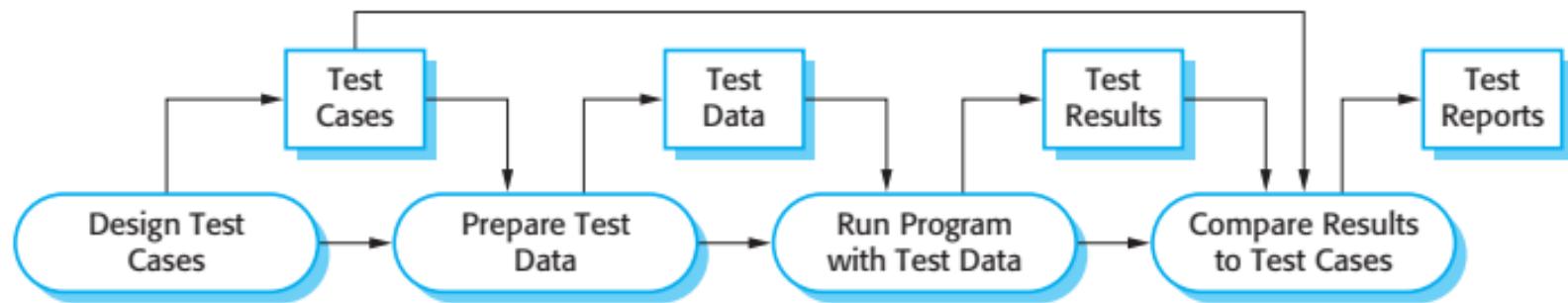
Inspection and Testing



Advantages of Inspections over Testing

- ❖ During testing, errors can **mask** (hide) other errors. Because inspection is a static process, you don't have to be concerned with **interactions** between errors.
- ❖ **Incomplete versions** of a system can be inspected without additional costs
- ❖ Compared with dynamic testing, typical defects that are **easier** and **cheaper** to **find** and **fix** through static testing include:
 - Requirement defects
 - Design defects
 - Coding defects
 - Deviations from standards
 - Incorrect interface specifications
 - Gaps or inaccuracies in test basis traceability or coverage

A model of the software testing process



Testing Methods

Selection of test cases

Testing is expensive and time consuming, so it is important to choose **effective** test cases.

Effectiveness, in this case, means two things

1. The test cases should show that, the components does what it is supposed to do.
2. Should reveal defects in the component, if there are any

These leads to two types of unit test cases

- Tests reflecting normal operation of a program
- Tests based on testing **experience** of where common problems arise. should use **abnormal inputs** to check that these are properly processed and **do not crash**

Testing Methods

❖ White Box Testing

- focuses on the **internal specifications** (e.g., algorithms, logic paths) of software units (i.e., modules)
- performed by developers

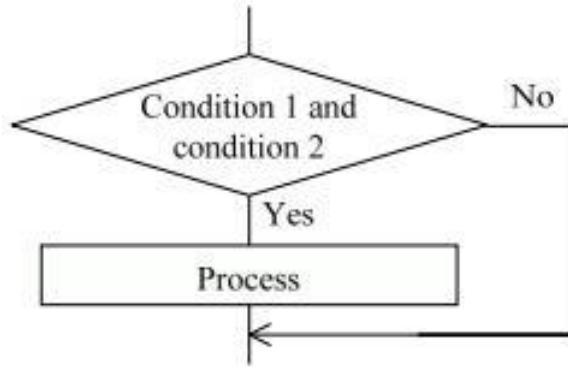
❖ Black Box Testing

- focuses on the **external specifications** (i.e., functional specifications and input/output (interface) specifications) of software units (i.e., modules)

White box testing

- **Statement coverage**
 - This designs test cases that execute every statement in a unit (i.e., module) at least once.
- **Decision condition coverage (branch coverage)**
 - This designs test cases that execute TRUE/FALSE at least once for every “decision condition” (i.e., branch) in a unit (i.e., module).
- **Condition coverage**
 - This designs test cases that execute TRUE/FALSE at least once for every “condition” used as a decision condition.
- **Decision condition / condition coverage**
 - This designs test cases that fulfill both decision condition coverage (i.e., branch coverage) and condition coverage.
- **Multiple-condition coverage**
 - This designs test cases that include every combination of TRUE/FALSE for every condition.

White box testing



(Truth value chart)

	Condition 1	Condition 2	Decision condition
I	TRUE	TRUE	TRUE
II	TRUE	FALSE	FALSE
III	FALSE	TRUE	FALSE
IV	FALSE	FALSE	FALSE

Coverage criteria	Test cases that fulfill coverage criteria
Statement coverage	(I)
Decision condition coverage (branch coverage)	(I and II) or (I and III) or (I and IV)
Condition coverage	(I and IV) or (II and III)
Decision condition /condition coverage	(I and II and III) Note: Avoid (I and IV).
Multiple-condition coverage	(I and II and III and IV)

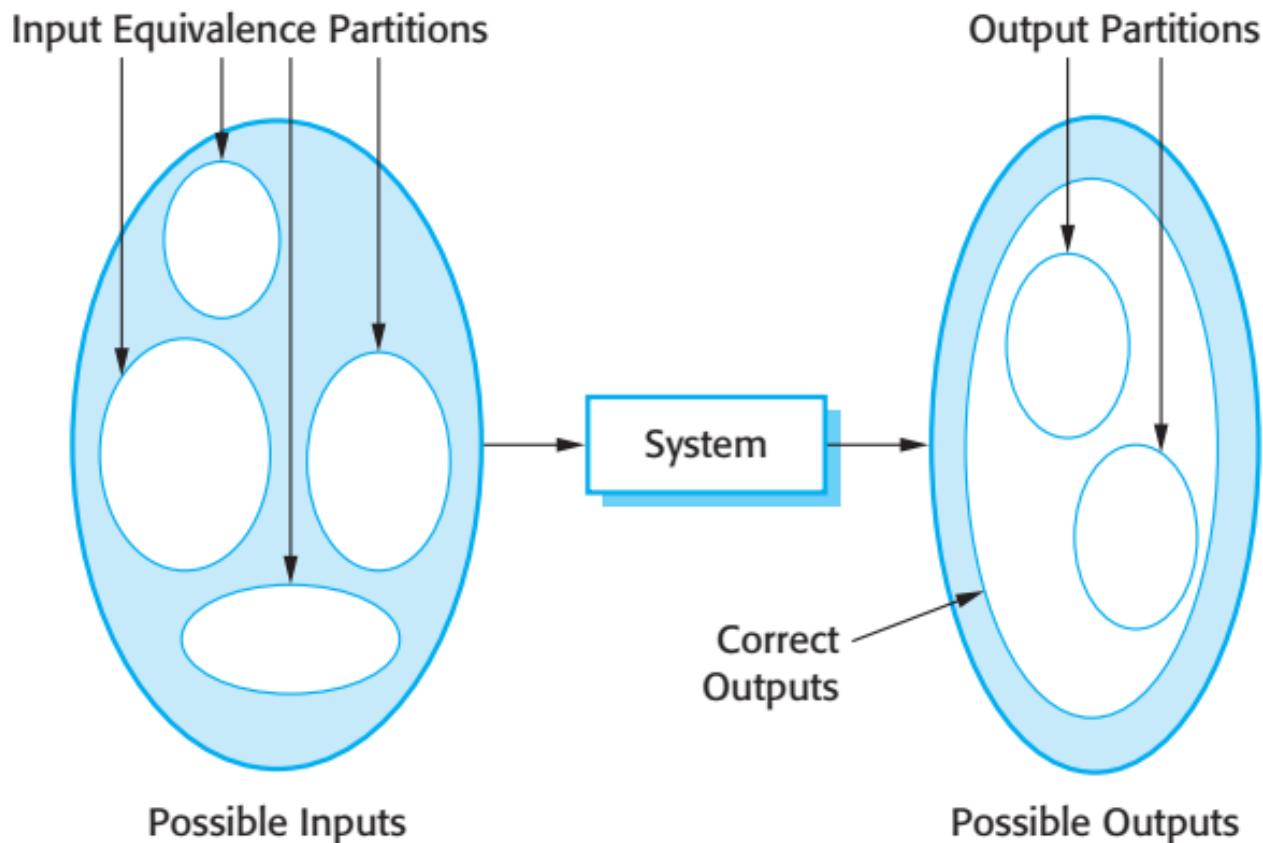
Black box testing

- Equivalence Partitioning
- Boundary Value Analysis

Equivalence Partitioning

- An **exhaustive** input test of a program is **impossible**.
- We are limited to a small subset of all possible inputs.
- We want to select the “right” subset, that is, the subset with the **highest probability** of finding the **most errors**.

Equivalence Partitioning



Identifying the Equivalence Classes

Guidelines for constructing equivalence classes:

- If an input condition specifies a **range of values**, identify one valid equivalence class and two invalid equivalence classes
- If an input condition specifies **the number of values**, identify one valid equivalence class and two invalid equivalence classes

Practice

11. In an Examination a candidate has to score minimum of 24 marks in order to clear the exam. The maximum that he can score is 40 marks. Identify the Valid Equivalence values if the student clears the exam. I

- A. 22, 23, 26
- B. 21, 39, 40
- C. 29, 30, 31
- D. 0, 15, 22

Practice

13. A program validates a numeric field as follows: values less than 10 are rejected, values between 10 and 21 are accepted, values greater than or equal to 22 are rejected. Which of the following input values cover all of the equivalence partitions? *
- A. 10, 11, 21
 - B. 3, 20, 21
 - C. 3, 10, 22
 - D. 10, 21, 22

Boundary Value Analysis

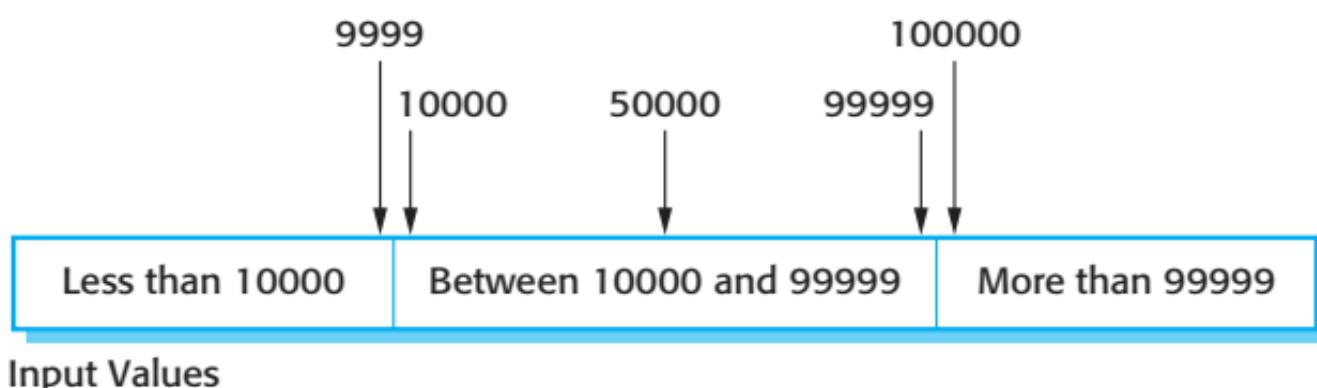
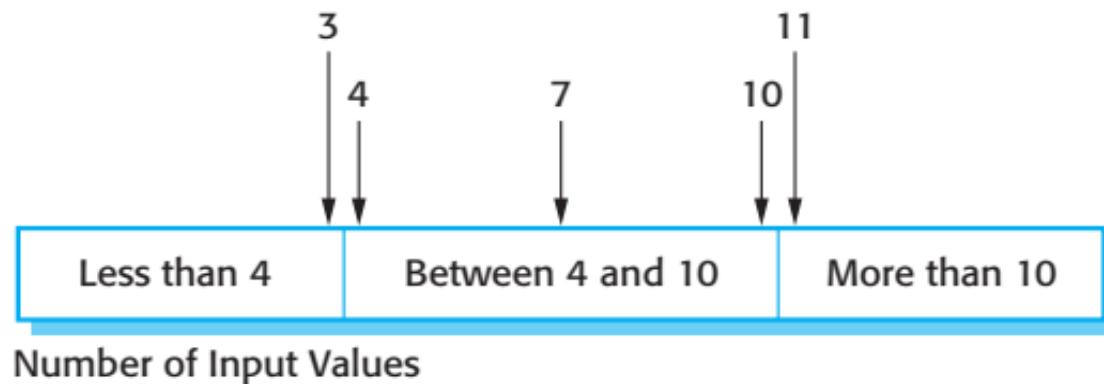
- ❖ Boundary conditions are those situations directly **on**, **above**, and **beneath** the edges of equivalence classes.
- ❖ General Guidelines:
 - If an input condition specifies a **range of values**, write test cases for the **ends of the range**, and invalid-input test cases for situations just **beyond the ends**.
 - If an input condition specifies a **number of values**, write test cases for the **minimum** and **maximum** number of values and one **beneath** and **beyond** these values.

Example

For example, say a program specification states that the program accepts 4 to 8 inputs which are five-digit integers (ex. 10,000). You use this information to identify the input partitions and possible test input values.

Example

For example, say a program specification states that the program accepts 4 to 8 inputs which are five-digit integers (ex. 10,000). You use this information to identify the input partitions and possible test input values.



General Testing Guidelines

- Choose inputs that force the system to generate all error messages
- Design inputs that cause input buffers to overflow
- Repeat the same input or series of inputs numerous times
- Force invalid outputs to be generated
- Force computation results to be too large or too small

Stages of Testing

Testing Procedure

❖ Manual testing

- A tester runs the program with some test data and compares the results to their expectations.

❖ Automated testing

- The tests are encoded in a program that is run each time the system under test (SUT) is to be tested
- Faster than manual testing
- Testing can never be completely automated
- Practically impossible to test systems that depend on how things look (UI)

Automated Testing

An automated test has three parts:

1. A **setup part**, where you **initialize** the system with the test case, namely the **inputs** and **expected** outputs
2. A **call part**, where you **call** the object or method to be tested.
3. An **assertion part**, where you **compare** the result of the call with the **expected result**. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

Automated Testing

```
package LAS;

import static org.junit.jupiter.api.Assertions.*;

class Test {
    private LoanApprovalSystem LS ;

    @org.junit.jupiter.api.BeforeEach
    public void setUp() throws Exception {
        LS = new LoanApprovalSystem();
    }

    @org.junit.jupiter.api.Test
    public void test1() throws Exception {
        LS.Receive(12000);
        assertEquals(12000, LS.getamount());
        LS.Approver(12000);
        assertEquals(false, LS.getapprove());
        LS.Reply(false, false);
        assertEquals("Loan Rejected", LS.getresult());
    }
}
```

Stages of Testing

❖ Development testing

- where the system is tested during development to discover bugs and defects.
- Done by System designers and programmers

❖ Release testing

- a complete version of the system is tested before it is released to users
- Done by a separate testing team

❖ User testing

- users or potential users of a system test the system in their own environment

Development Testing

Development testing includes all testing activities that are carried out by the team developing the system.

Testing may be carried out at **three levels of granularity**:

❖ Unit testing:

- individual program units or object classes are tested.

❖ Component testing

- where several individual units are integrated to create composite components

❖ System testing

- some or all of the components in a system are integrated and the system is tested as a whole

Unit Testing

Unit testing is the process of testing individual program components in isolation.

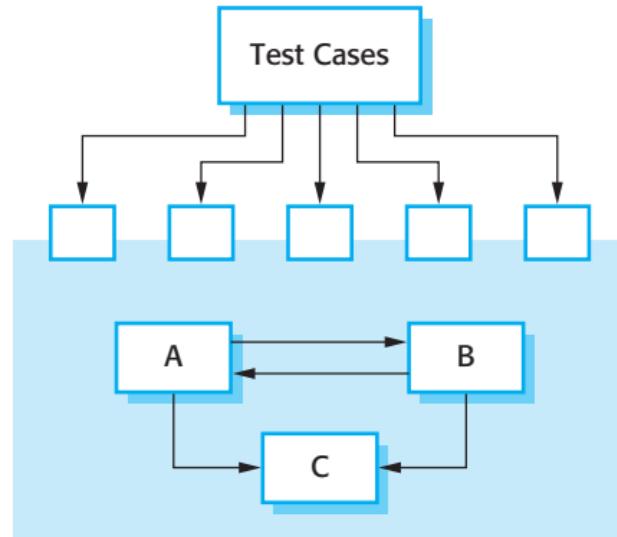
- ❖ Units may be:
 - Individual **functions** or **methods** within an object.
 - **Object classes** with several **attributes** and **methods**
 - Composite components with defined interfaces used to access their functionalities
- ❖ Testing is performed by calling these units with **different input** parameters

Object Class Testing

- Testing should be designed in a way which will provide **coverage** to all the **features** of the objects.
- Complete test coverage of a class involves:
 - testing all **operations** associated with an object
 - setting and checking the values of all **attributes** associated with the object
 - putting the object into all possible **states**

Component Testing

- ❑ After individual objects have been unit tested in isolation, they are **combined** (integrated) to form “composite components”.
- ❑ The functionalities of these objects are accessed through their defined component interfaces
- ❑ **Component testing** focuses on showing that the component interface **behaves** according to its specification



Interface Testing

✧ Parameter interfaces

- data are passed from one component to another.

✧ Shared memory interfaces

- a block of memory is shared between components.

✧ Procedural interfaces

- one component encapsulates a set of procedures that can be called by other components.

✧ Message passing interfaces

- one component requests a service from another component by passing a message to it.

Interface Errors

✧ Interface misuse

- A calling component calls some other component and makes an error in the use of its interface. Example: **wrong type** of parameters or be passed in the **wrong order**, or the **wrong number** of parameters may be passed

✧ Interface misunderstanding

- A calling component **misunderstands** the **specification** of the interface of the called component and makes **assumptions** about its behavior .

✧ Timing errors

- The called and calling components **operate at different speeds** and **out-of-date information** is accessed.

Interface Testing Guidelines

- Design tests in which parameters of the called components are at the extreme ends of their ranges.
- always test pointer parameters with null pointer
- Design tests which cause the component to fail
- Use stress testing in message passing systems.
- For shared memory, design tests that vary the order in which these components are activated

System Testing

- ✧ System testing during development involves **integrating components** to create a version of the system and then testing the integrated system
- ✧ Focuses on testing the **interactions** between the components
- ✧ Checks whether the components are **compatible**, **interacts correctly** and **transfer data correctly** across their interfaces.

System Testing Vs Component Testing

- ❖ During system testing, **reusable components** that have been separately developed and **off-the-shelf systems** may be integrated with newly developed components. The complete system is then tested
- ❖ Components developed by **different team members** or **groups** may be **integrated** at this stage. System testing is a **collective** rather than an **individual** process.