CHAPTER 1

Software engineering is concerned with theories, methods and tools for professional software development.

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.

Software costs

The costs of software on a PC are often greater than the hardware cost. Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times development costs. " Software engineering is concerned with cost-effective software development.

There are two kinds of software products:

Generic products

These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them.

Example: databases, word processors, drawing packages, and project-management tools.

Customized (or bespoke) products

These are systems that are commissioned by a particular customer. A software contractor develops the software especially for that customer

Difference:

In generic products, the organization that develops the software controls the software specification.

For custom products, the specification is usually developed and controlled by the organization that is buying the software.

Essential attributes of good software

Maintainability: Software should be written in such a way so that it can evolve to meet the changing needs of customers.

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 utilization, 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 is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
2. Using software engineering methods and techniques for software systems is generally less expensive in the long term compared to simply writing programs as if it were a personal project.

Software process activities:

- **Software specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
- **Software development**, where the software is designed and programmed.
- **Software validation**, where the software is checked to ensure that it is what the customer requires.
- **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.

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.

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.

Application types

Stand-alone applications:

These are 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.

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.

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.

Entertainment systems

These are systems that are primarily for personal use and which are intended to entertain the user.

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.

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.

Systems of systems

These are systems that are composed of a number of other software systems.

Internet software engineering

The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.

Web services allow application functionality to be accessed over the web.

Cloud computing is a way of providing computer services where applications run on the internet instead of your own device

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.

Following should be considered while developing a web application:

- **Software reuse:**

  It means using ready-made software pieces to put together the new system, instead of starting from the beginning. This saves time and effort by using what's already available.

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

SOFWARE ENGEINEERING ETHICS:

- You should uphold normal standards of honesty and integrity.
- You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession.

There are areas where standards of acceptable behaviour are not bound by laws but by the more tenuous notion of professional responsibility:

- **Confidentiality**: You should normally respect the confidentiality of your employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
- **Competence:** You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.
- **Intellectual property rights:** You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.
- **Computer misuse:** You should not use your technical skills to misuse other people's computers.

ACM/IEEE CODE OF ETHICS:

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.

Ethical Dilemma

Chapter 2 – Software Processes

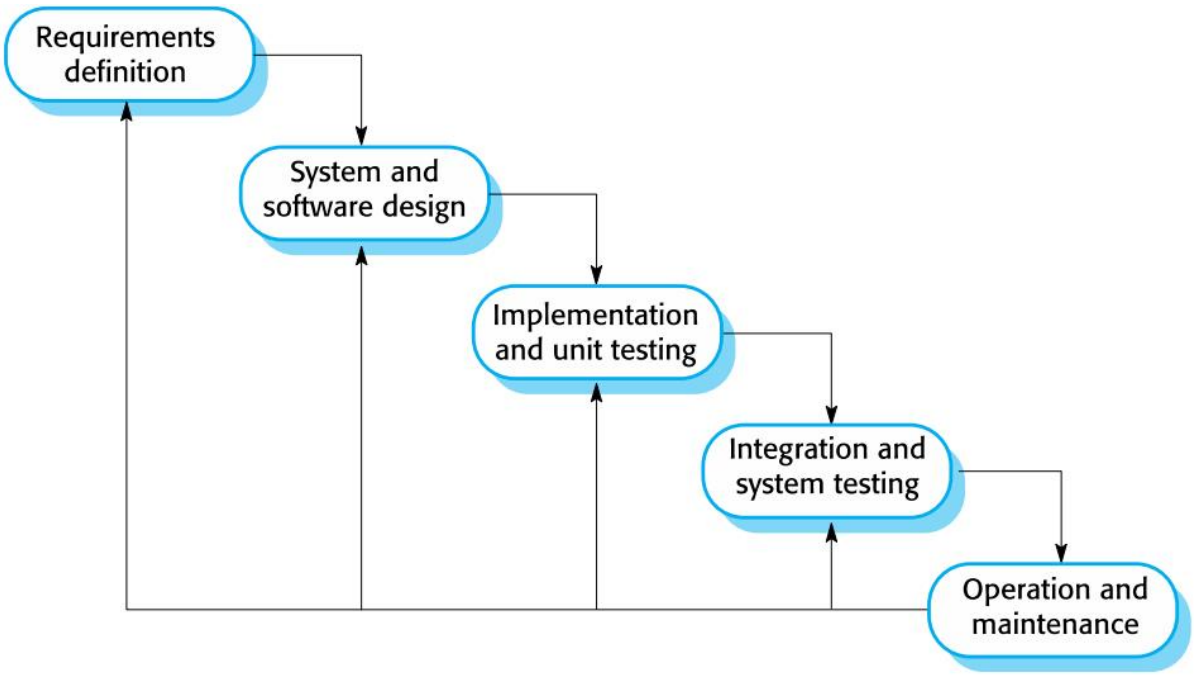
A software process is a set of related activities that leads to the production of a software product.

There are many different software processes but all must include four activities that are fundamental to software engineering:

1. **Software specification**: The functionality of the software and constraints on its operation must be defined.
2. **Software design and implementation**: The software to meet the specification must be produced.
3. **Software validation**: The software must be validated to ensure that it does what the customer wants.
4. **Software evolution**: The software must evolve to meet changing customer needs

Software process models

The waterfall model:



This model is called the "waterfall model" or software life cycle because one phase leads to another like a cascading waterfall.

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

2. System and software design:

The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture.

Software design involves identifying and describing the fundamental software system abstractions and their relationships.

3. Implementation and unit testing:

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

4. Integration and system testing:

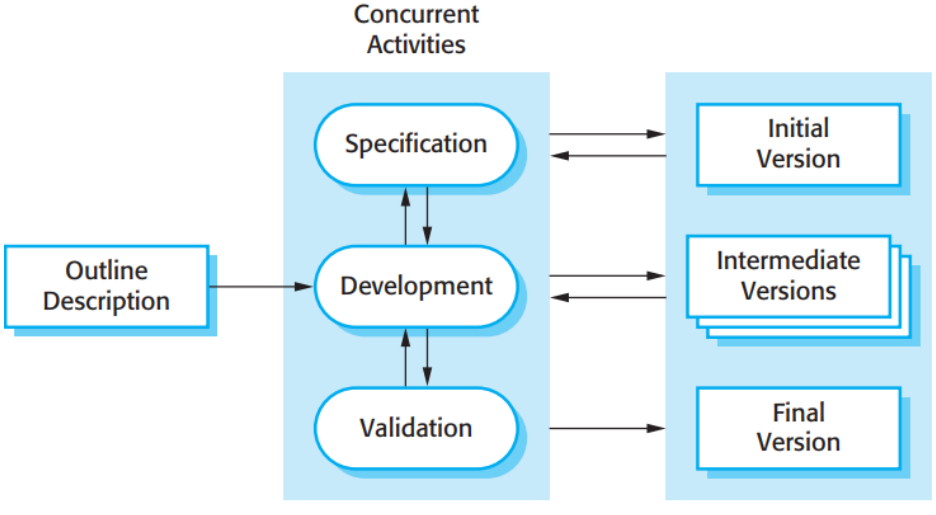
The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. Operation and maintenance:

The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle.

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

Incremental development



Incremental development is based on the idea of developing an initial implementation and then improving it little by little based on feedback from users. This process continues through several versions until a satisfactory final product is achieved. Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.

Each increment or version of the system incorporates some of the functionality that is needed by the customer.

Three important benefits:

- The cost of accommodating changing customer requirements is reduced. The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented.
- More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included.

The incremental approach has two problems:

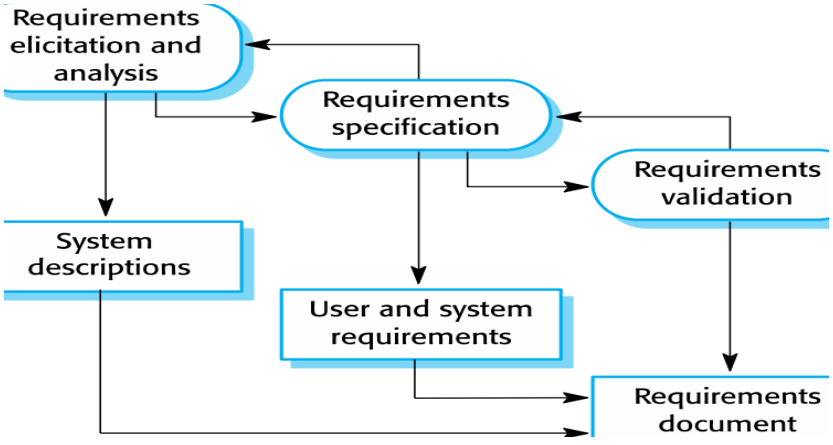
- The process is not visible.
- System structure tends to degrade as new increments are added.

Integration and configuration

Based on software reuse where systems are integrated from existing components or application systems.

Reused elements may be configured to adapt their behaviour and functionality to a user's requirements.

Software specification or requirements engineering



Requirements elicitation and analysis:

This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on.

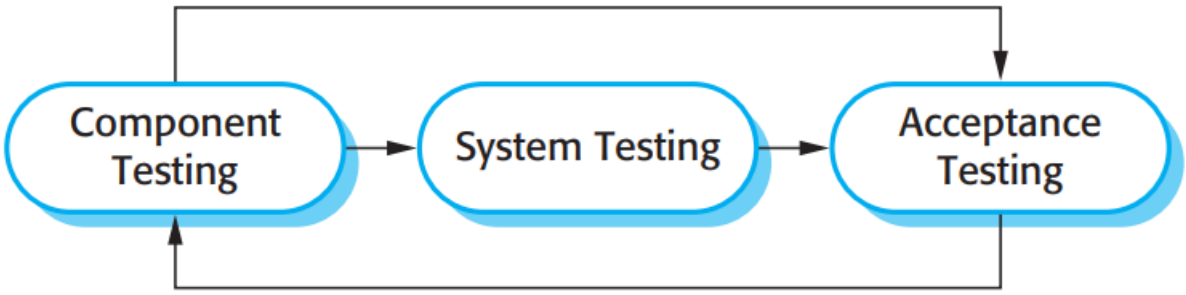
Requirements specification:

Requirements specification is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements.

Requirements validation:

This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered.

STAGES OF TESTING:



Component testing:

The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components.

System testing:

System components are integrated to create a complete system. This process is concerned with finding errors that result from unexpected interactions between components

Acceptance testing:

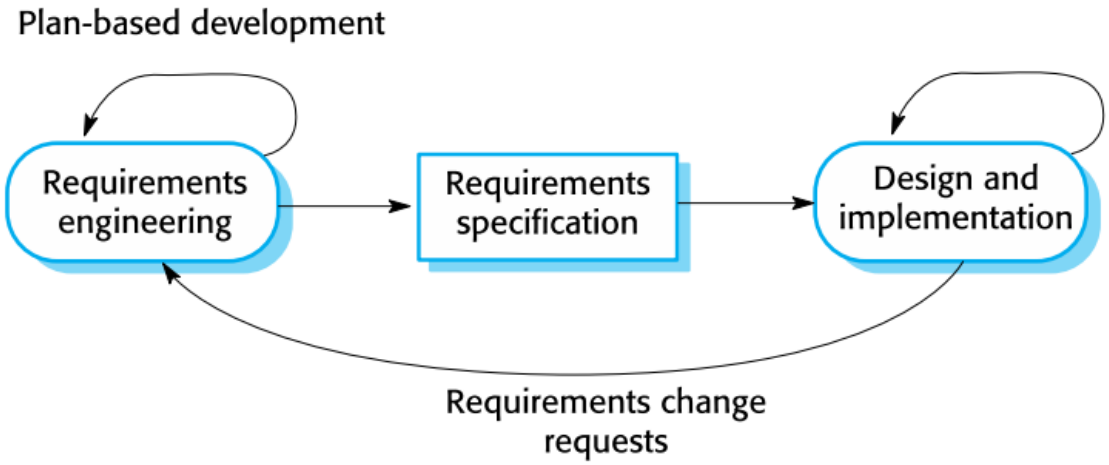
This is the final stage in the testing process before the system is accepted for operational use. The system is tested with customer data to check that the system meets the customer's needs.

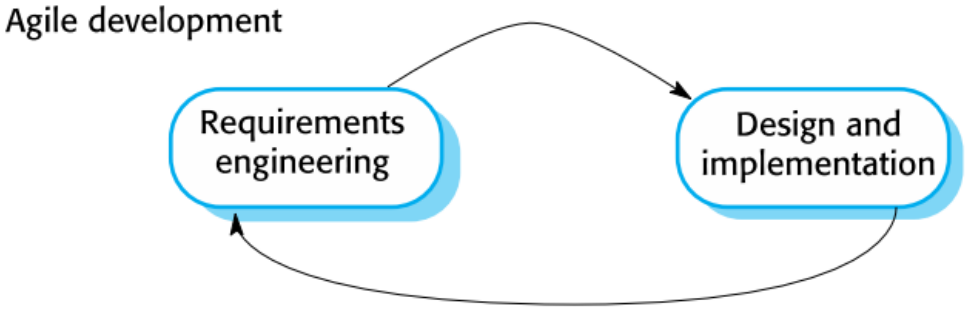
CHAPTER 3

Agile development (fundamental characteristics of rapid software development)

- The processes of specification, design, and implementation are interleaved.
- The system is developed in a series of versions. End-users and other system stakeholders are involved in specifying and evaluating each version.
- 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.
- Agile methods are a way of building things step by step, making small updates to the system every two or three weeks to keep improving it for customers.
- They minimize documentation by using informal communications rather than formal meetings with written documents.

Plan-driven and agile development





In a plan-driven approach to software engineering, the development process is divided into stages, and each stage produces specific results. These results are used to plan the next steps in the process.

iteration occurs within activities.

MANIFESTO:

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

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

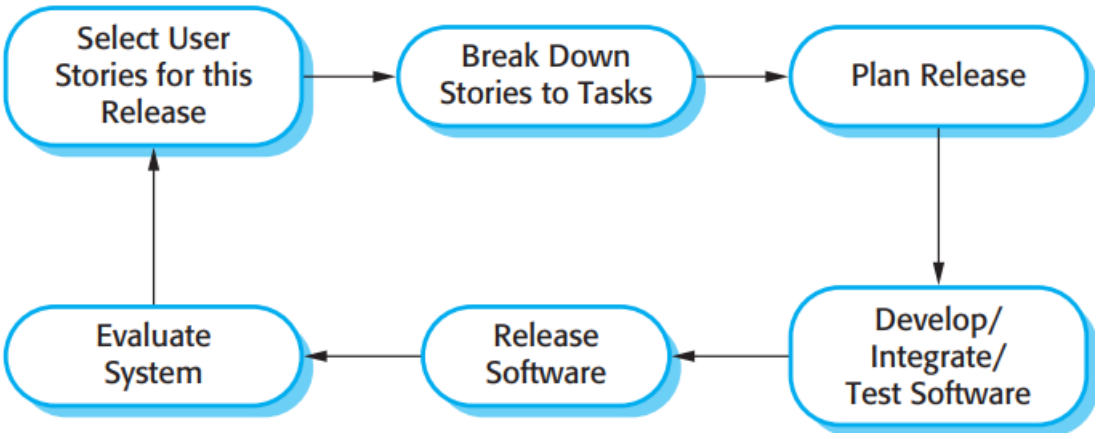
The principles of agile methods

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.

Extreme programming

Extreme Programming (XP) takes an 'extreme' approach to iterative development.

New versions may be built several times per day.



Extreme programming practices:

- Incremental planning
- Small releases
- Simple design
- Test-first development
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- Sustainable pace
- On-site customer

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.

Refactoring

- This means that the programming team look for possible improvements to the software and implement them immediately.
- 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.
- Example:
- Reorganization of a class hierarchy to remove duplicate code.
- Tidying up and renaming attributes and methods to make them easier to understand.

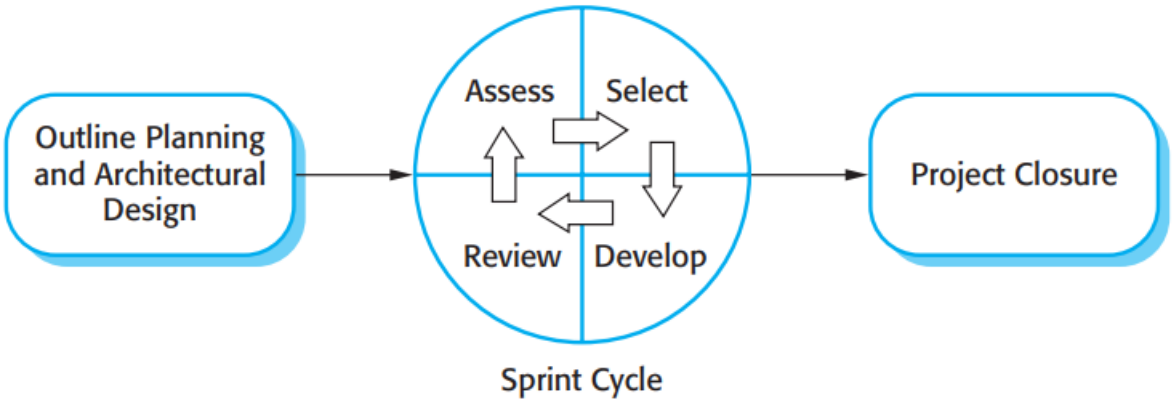
Test automation: Test automation means that tests are written as executable components before the task is implemented.

These testing components should be stand-alone, should simulate the submission of input to be tested and should also verify that the output matches the expected result as specified.

Pair programming

- Pair programming involves programmers working in pairs, developing code together.
- 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.
- It supports the idea of collective ownership and responsibility for the system.
- It acts as an informal review process because each line of code is looked at by at least two people.
- It encourages refactoring as the whole team can benefit from improving the system code.

The **Scrum approach** is a general agile method but its focus is on managing iterative development rather than specific technical approaches to agile software engineering.



There are three phases in Scrum:

- The first 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.
- Finally, the project closure phase wraps up the project, completes required documentation

The Scrum sprint cycle

1. During the selection phase, the entire project team collaborates with the customer to choose the features and functionality from the product backlog that will be developed in the current sprint.
2. Once these are agreed, the team organize themselves to develop the software. During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called 'Scrum master'.
3. At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

Chapter 4 – Requirements Engineering

Types of requirements:

User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

System requirements are more detailed descriptions of the software system's functions, services, and operational constraints.

The system requirements document should define exactly what is to be implemented.

Functional and non-functional requirements

Software system requirements are often classified as functional requirements or non-functional requirements:

Functional requirements

- These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.
- The functional requirements for a system describe what the system should do.
- These requirements depend on the type of software being developed, the expected users of the software.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.

Requirements imprecision

- Problems arise when functional requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.

Example:

Consider the term 'search' in the following requirement:

A user shall be able to search the appointments lists for all clinics.

- User intention – search for a patient name across all appointments in all clinics;
- Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

Non-functional requirements

As the name suggests, these are requirements that are not directly concerned with the specific services delivered by the system to its users.

These define system properties such as reliability, response time and storage requirements and Constraints such as I/O device capability, system representations, etc.

Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

TYPES OF NON-FUNCTIONAL REQUIREMENTS:

Product requirements

These requirements specify or constrain the behaviour of the software.

For example, they can include requirements like how fast the system needs to work, how much memory it can use, how dependable it should be in terms of failures, security requirements, and usability requirements

Organizational requirements

These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization.

Examples:

- **operational process requirements** that define how the system will be used.
- **Development process requirements** that specify the programming language, the development environment or process standards to be used.
- **Environmental requirements** that specify the operating environment of the system.

External requirements

This broad heading covers all requirements that arise from factors which are external to the system and its development process.

These may include **regulatory requirements** that set out what must be done for the system to be approved for use by a regulator;

Legislative requirements that must be followed to ensure that the system operates within the law; and

Ethical requirements that ensure that the system will be acceptable to its users and the general public.

Metrics for specifying nonfunctional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Requirements engineering processes

It involves activities for defining and managing software or system requirements.

It comprises four main steps:

A feasibility study to assess alignment with business goals;

Elicitation and analysis of requirements from stakeholders,

Specification of these requirements in a standardized format, and

Validation to confirm accuracy.

----------OR----------

Feasibility study: assessing if the system is useful to the business.

Elicitation and analysis: discovering requirements.

Specification: converting these requirements into some standard form.

Validation: checking that the requirements actually define the system that the customer wants.

However, in practice, requirements engineering is an iterative process in which the activities are interleaved.

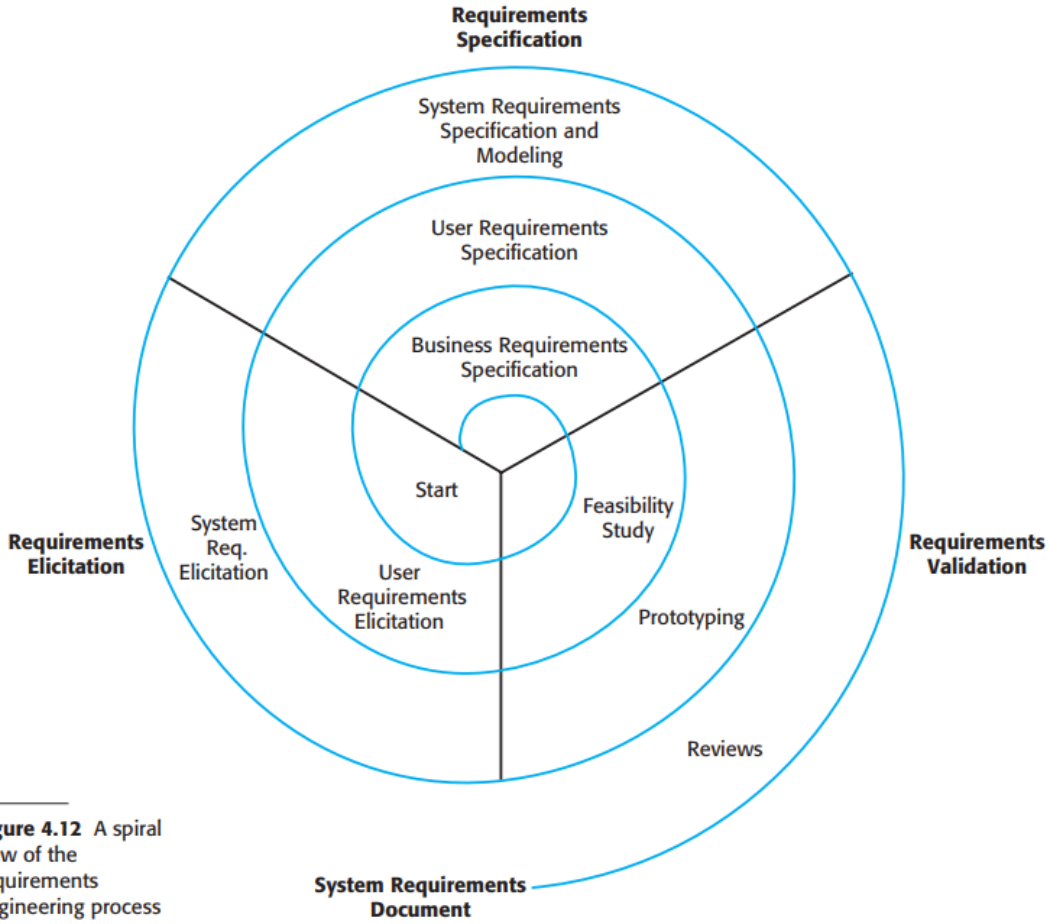
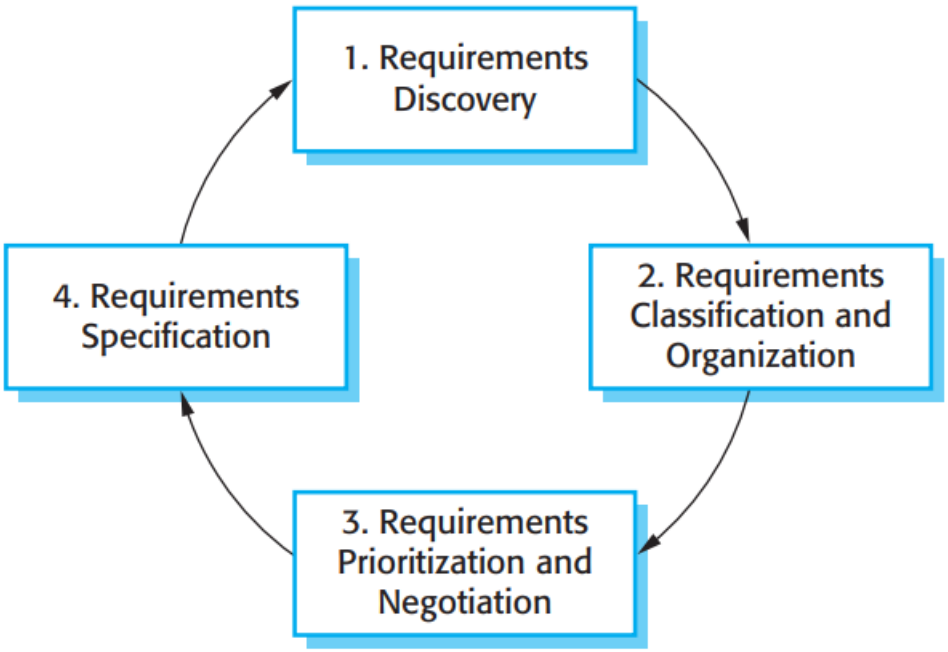


Figure 4.12 A spiral view of the requirements engineering process

- The activities are organized as an iterative process around a spiral, with the output being a system requirements document.
- Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system.
- Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the detailed system requirements.
- The spiral model can adapt to different ways of developing requirements at different levels of detail.
- The number of iterations around the spiral can vary so the spiral can be exited after some or all of the user requirements have been elicited.

Requirements elicitation and analysis



In this activity, software engineers work with customers and system end-users to find out about

- the application domain,
- what services the system should provide,
- the required performance of the system,
- hardware constraints, and so on.

- **Requirements discovery**
  - This is the process of interacting with stakeholders of the system to discover their requirements.
  - Domain requirements from stakeholders and documentation are also discovered during this activity.

- **Requirements classification and organization**

This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters.

- **Requirements prioritization and negotiation**
  - This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.
  - Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

- **Requirements specification**

The requirements are documented and input into the next round of the spiral.

Requirements specification

It is the process of writing down the user and system requirements in a requirements document.

Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent.

User requirements have to be understandable by end users and customers who do not have a technical background.

System requirements are more detailed requirements and may include more technical information.

Ideally, the system requirements should simply describe the external behaviour of the system and its operational constraints.

Requirements validation

Requirements validation is the process of checking that requirements actually define the system that the customer really wants.

Requirements validation is important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document.

These checks include:

1. Validity checks:

A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required.

Validity check is done to check whether the system provides function that best supports the customer's needs.

2. Consistency checks

Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.

3. Completeness checks

The requirements document should include requirements that define all functions and the constraints intended by the system user.

4. Realism checks

Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented within a given budget.

5. Verifiability

To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable.

This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

Requirements validation techniques

- **Requirements reviews:** The requirements are analysed systematically by a team of reviewers who check for errors and inconsistencies.
- **Prototyping:** an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
- **Test-case generation:** Requirements should be testable. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered.

Requirements management

Requirements management is the process of managing changing requirements during the requirements engineering process and system development.

Once a system has been installed and is regularly used, new requirements inevitably emerge.

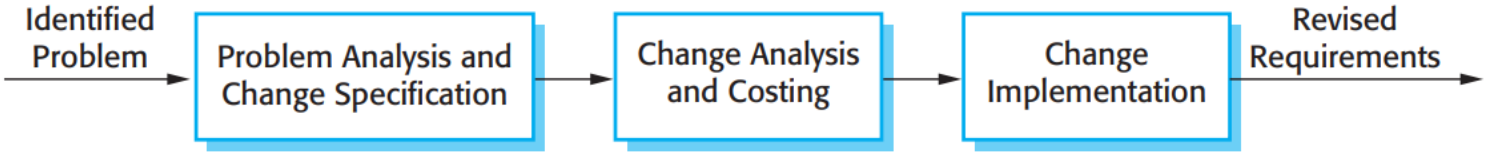
You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes.

Requirements management planning

Requirements management decisions:

a) **Requirements identification** Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
b) **A change management process** This is the set of activities that assess the impact and cost of changes.
c) **Traceability policies** These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
d) **Tool support** Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements change management



There are three principal stages to a change management process:

1. **Problem analysis and change specification**

- During this stage, the problem or the change proposal is analysed to check that it is valid.
- This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

2. **Change analysis and costing**

- The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements.
- The cost of making the change is estimated.
- Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

3. **Change implementation**

- The requirements document and, where necessary, the system design and implementation, are modified.

CHAPTER 5: SYSTEM MODELLING

System modelling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.

Models are used during the requirements engineering process to help derive the requirements for a system.

The most important aspect of a system model is that it leaves out detail. A model is an abstraction of the system being studied rather than an alternative representation of that system.

Ideally, a representation of a system should maintain all the information about the entity being represented.

System perspectives

a) An **external perspective**, where you model the context or environment of the system.
b) An **interaction perspective** where you model the interactions between a system and its environment or between the components of a system.
c) A **structural perspective**, where you model the organization of a system or the structure of the data that is processed by the system.
d) A **behavioural perspective**, where you model the dynamic behaviour of the system and how it responds to events.

CONTEXT MODELS

- It shows what lies outside the system boundaries.
- Does not show process or how the systems interact.
- At an early stage in the specification of a system, you should decide on the system boundaries.

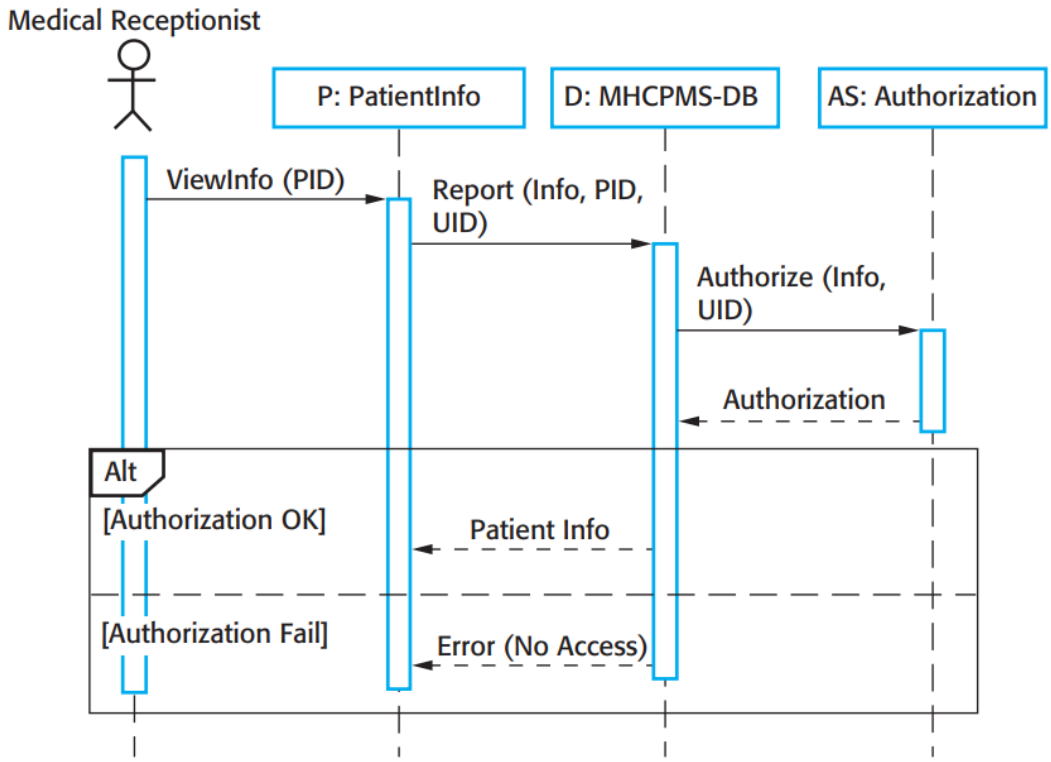
- This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system's environment.

INTERACTION MODELS

All systems involve interaction of some kind.
This can be user interaction, which involves user inputs and outputs;
Interaction between the system being developed and other systems;
Interaction between the components of the system.

a) Use case modelling
   - Mostly used to model interactions between a system and external actors
   - Each use case represents a discrete task that involves external interaction with a system.
   - In its simplest form, a use case is shown as an ellipse with the actors involved in the use case represented as stick figures.
   - Use case diagrams give a fairly simple overview of an interaction so you have to provide more detail to understand what is involved. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram.

b) Sequence diagrams
   - Used to model interactions between system components, although external agents may also be included.
   - As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
   - The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
   - Interactions between objects are indicated by annotated arrows.
   - The rectangle on the dotted lines indicates the lifeline of the object.



STRUCTURAL MODELS

Structural models of software display the organization of a system.

Structural models may be static models, which show the structure of the system design.

Or dynamic models, which show the organization of the system when it is executing.

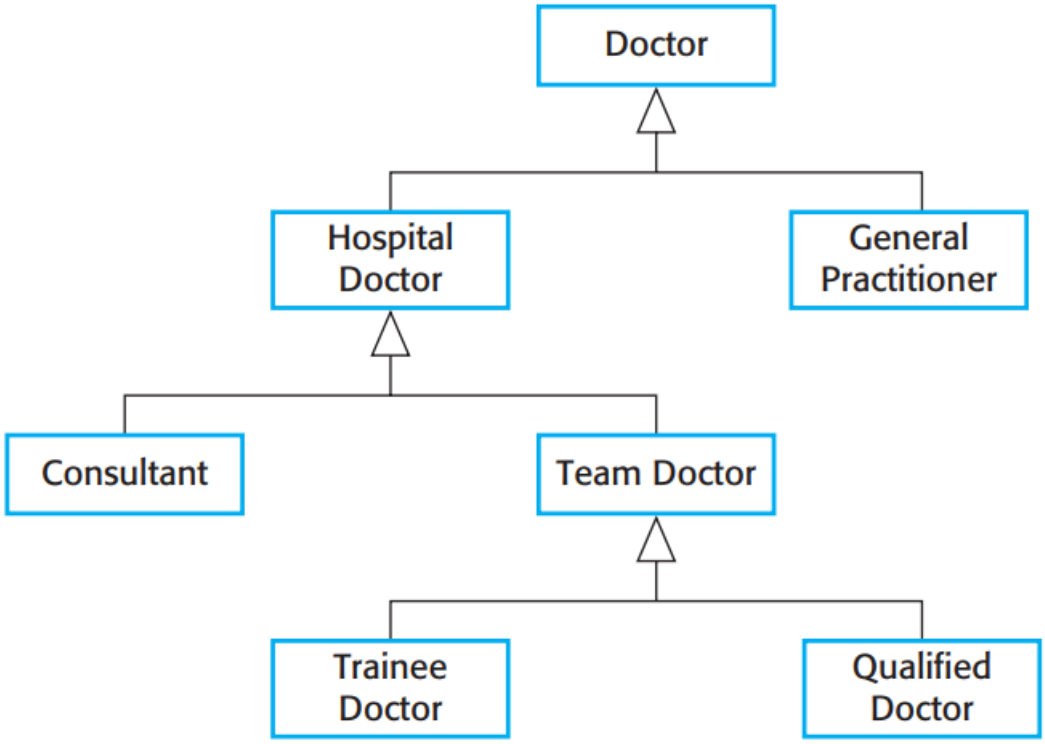
a) Class diagrams

- Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- an object class can be thought of as a general definition of one kind of system object.
- An association is a link between classes that indicates that there is some relationship between these classes.

b) Generalization

- Generalization is an everyday technique that we use to manage complexity.
- Rather than learn the detailed characteristics of every entity, we place them in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.

- In modelling systems, it is often useful to examine the classes in a system to see if there is scope for generalization.



c) Aggregation

- An aggregation model shows how classes that are collections are composed of other classes.
- The UML provides a special type of association between classes called aggregation that means that one object (the whole) is composed of other objects (the parts).
- To show this, we use a diamond shape next to the class that represents the whole.

BEHAVIORAL MODELS

They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.

2 types:

- ***Data*** Some data arrives that has to be processed by the system.
- ***Events*** Some event happens that triggers system processing.

a) Data-driven modelling

- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.

- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

b) Event-driven modelling

- Event-driven modelling shows how a system responds to external and internal events.
- It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

MODEL-DRIVEN ENGINEERING

- Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
- The programs that execute on a hardware/software platform are then generated automatically from the models.
- This raises the level of abstraction in software engineering.

Model driven architecture (used by MDE)

- It is a model-focused approach to software design and implementation that uses a sub-set of UML models to describe a system.
- MDA method recommends that three types of abstract system model:
  a) Computation independent model (CIM):
     These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
  b) Platform independent model (PIM):
     These model the operation of the system without reference to its implementation.
  c) Platform specific models (PSM):
     These are transformations of the platform-independent model with a separate PSM for each application platform.

COMPONENT BASED SOFTWARE ENGINEERING (CBSE)

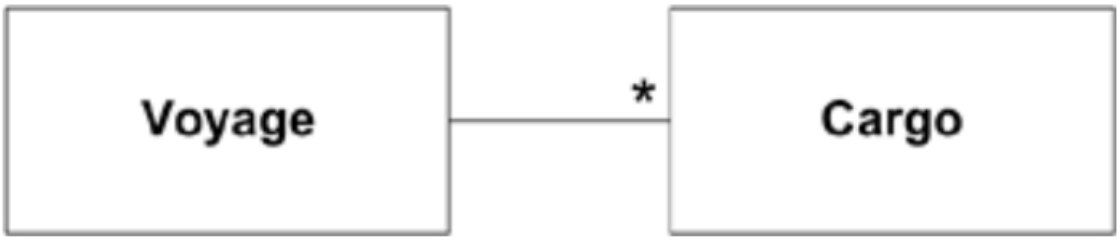
- It is an approach to software systems development based on reusing software components.
- CBSE is the process of defining, implementing, and integrating or composing loosely coupled, independent components into systems.
- The only way that we can cope with complexity and deliver better software more quickly is to reuse rather than reimplement software components.

Essentials:

- There should be a clear separation between the component interface and its implementation.
- Component standards that facilitate the integration of components.
- Middleware that provides software support for component integration.
- A development process that is geared to component-based software engineering.

Q7. INNOVATIVE:

Let's take a simple domain model that could be the basis of an application for booking cargos onto a voyage of a ship.



We can state that the booking application's responsibility is to associate each Cargo with a Voyage, recording and tracking that relationship.

Somewhere in the application code there could be a method like this:

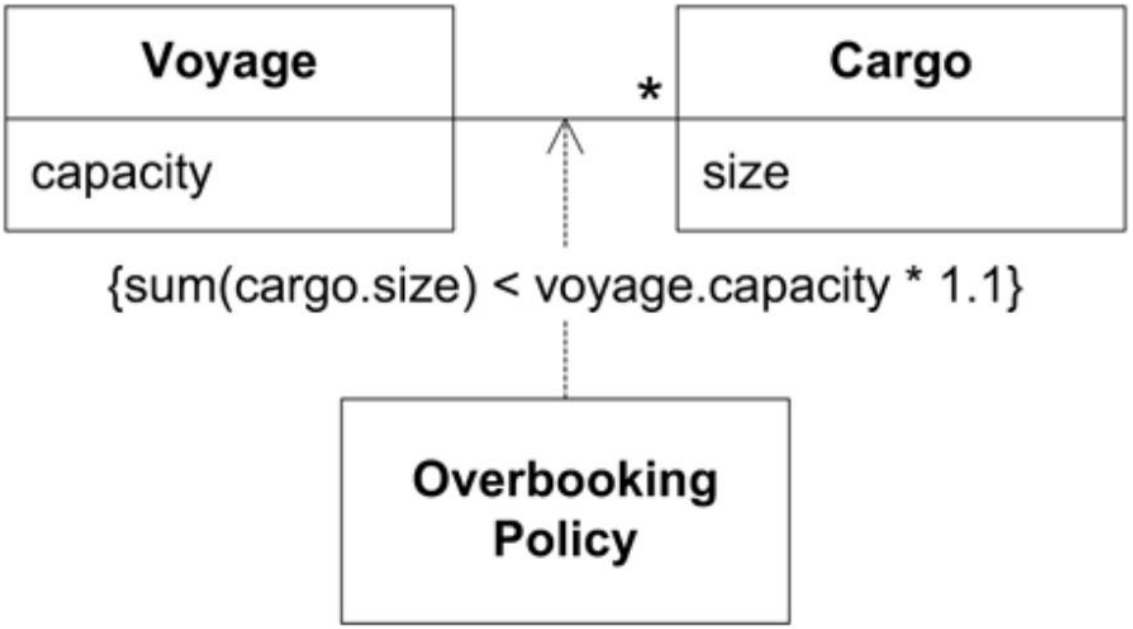
```
public int makeBooking(Cargo cargo, Voyage voyage) {
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}
```

Because there are always last-minute cancellations, standard practice in the shipping industry is to accept more cargo than a particular vessel can carry on a voyage. This is called "overbooking." Sometimes a simple percentage of capacity is used, such as booking 110 percent of capacity.

The requirements document contains this line:

> Allow 10% overbooking.


```
public int makeBooking(Cargo cargo, Voyage voyage) {
    double maxBooking = voyage.capacity() * 1.1;
    if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
        return -1;
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}
```



The code is now:

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   if (!overbookingPolicy.isAllowed(cargo, voyage)) return -1;
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

The new **Overbooking Policy** class contains this method:

```
public boolean isAllowed(Cargo cargo, Voyage voyage) {
   return (cargo.size() + voyage.bookedCargoSize()) <=
         (voyage.capacity() * 1.1);
}
```