

# CPT-203 w2-w7

## W2 - Lecture 1: Introduction

### 1. What is computer software:

Computer software is **the product that software professionals build and then support over the long term**. It encompasses **computer programs and associated documentation** that execute within a computer of any size and architecture. **Software products may be developed for a particular customer or may be developed for a general market.**

### 2. Software systems are abstract and intangible.

### 3. Software engineering 缺点

- (1) Software engineering is **criticized** as inadequate for modern software development.
- (2) Software failures are a consequence of two factors:
  - ① Increasing demands
  - ② Low expectations

### 4. Professional software usually has the following properties:

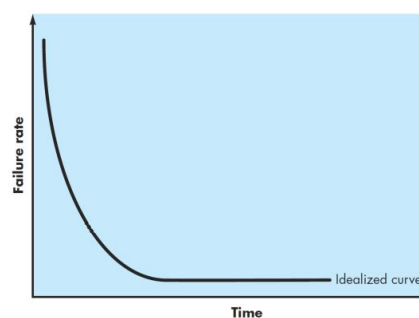
- (1) Strict user requirements
- (2) Required accuracy and data integrity
- (3) Higher security standard
- (4) Stable performance for heavy load
- (5) Required technical support, etc

## 5. 参考下图，具备归类能力

Product characteristics	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 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.

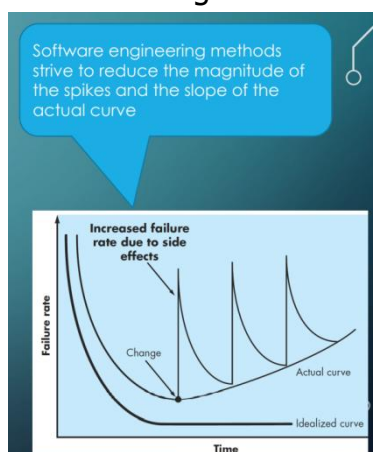
## 6. SOFTWARE DETERIORATION

(1) Software 不会磨损（wear out），理论上它的错误率随时间的变化是下图



(2) 但实际上，software 会恶化（deterioration）。

- ① • During its life, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the “actual curve”.
- ② • Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again.



## 7. **software processes(Lecture 3):**

- (1) Software specification
- (2) Software development
- (3) Software validation
- (4) Software evolution

## 8. **APPLICATION TYPES**

- (1) Stand-alone applications （不需要联网）
- (2) Interactive transaction-based applications
- (3) Embedded control systems
- (4) Batch processing systems （批处理，包含大量的用户输入与输出）
- (5) Entertainment systems
- (6) Systems for modeling and simulation
- (7) Data collection systems （使用传感器搜集数据并且处理的系统）
- (8) Systems of systems （由子系统组成）

## 9. **WEB 开发**

- (1) Software reuse is the dominant approach
- (2) Should be developed and delivered incrementally.
- (3) User interfaces are constrained by the capabilities of web browsers.

10. The **client satisfaction** is the **primary measurement** of success in software project.

- (1) • Feasibility studies
- (2) • Separation of requirement from design
- (3) • Milestones and releases
- (4) • Acceptance and user testing

11. Suggest a few ways to build software to **stop deterioration due to change**.

- The software should be **modular** so that changes will not have a lot of side effects to other part of the software
- The software must be **maintainable**
- **Comprehensive testing** should put in place to reduce errors.
- **Work closely with the stakeholder** to ensure requirements are correctly defined
- **Improve requirement study** approach to achieve better requirements study

**TTL - 1** 几道题，直接看

## **W3 - Lecture 2: Software Processes**

1. 四个 **fundamental step** 每个 **process** 都具有：

- (1) Software specification

(2) Software design and implementation

(3) Software validation

(4) Software evolution

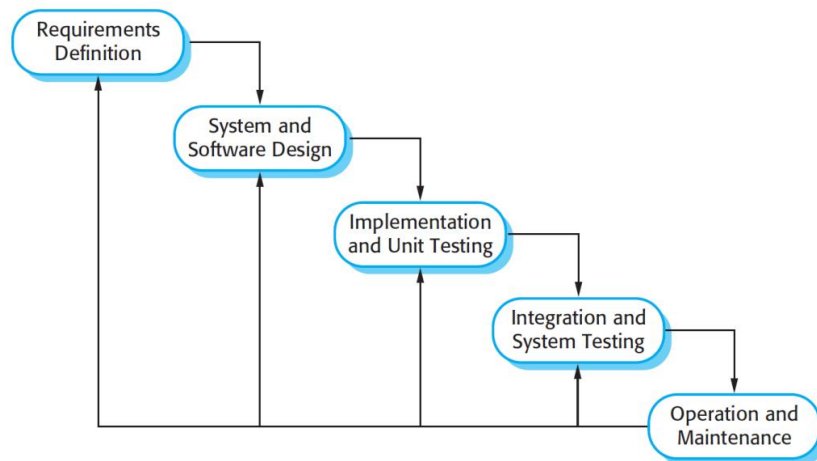
2. software processes are categorized as either **plan-driven** or **agile processes**

### 3. Software Process Model:

#### (1) The waterfall model (plan-driven):

① This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases, such as **requirements specification, software design, implementation, testing**, etc.

② Example:

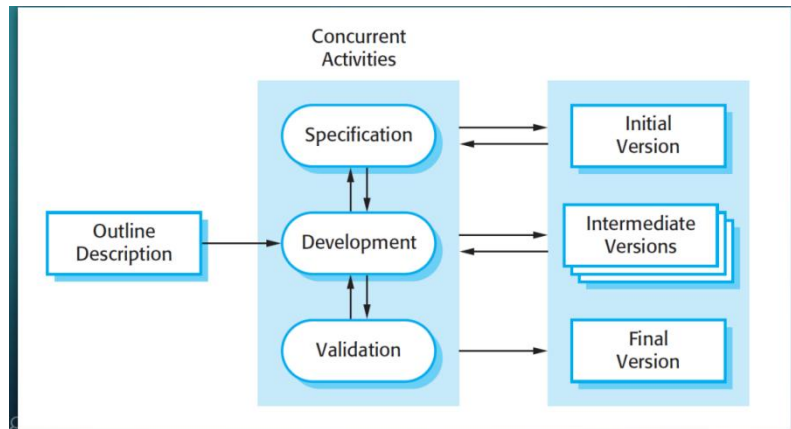


③ 随着开发继续，前面的部分会被冻结，避免迭代造成大量资源消耗。每个阶段都会产生文件，以便管理者查看进度。

④ **waterfall** 缺点：必须在项目早期作出承诺，使得其很难适应要求不断变化的客户。使用瀑布模型只能在需求在被充分理解并且在开发时不太可能发生根本变化的情况下使用。

#### (2) Incremental development

① Incremental development is based on the idea of developing an initial implementation, **exposing this to user comment and evolving it through several versions** until an adequate system has been developed.



②

③ 交错进行 specification, development, and validation activities. 各个部分相互反馈。

④ Incremental software development, which is a fundamental part of agile approaches, is better than a waterfall approach for **most business, e-commerce, and personal systems**

⑤ **Incremental** 优点:

1) The cost of **accommodating changing customer requirements is reduced.**

2) It is easier to **get customer feedback on the development work that has been done.**

3) More **rapid delivery and deployment of useful software to the customer is possible**, even if all of the functionality has not been included.

⑥ **Incremental** 缺点:

1) The process is **not visible**. (不像 waterfall 可以每个阶段产生文件)

2) System structure tends to degrade as new increments are added

3) **large organizations have bureaucratic** (官僚的) **procedures** that have evolved over time and there may be a mismatch between these procedures and a more informal iterative or agile process

4) Formal procedures are **required by external regulations** (e.g., accounting regulations)

5) The problems become **particularly acute for large, complex, long-lifetime systems**, where different teams develop different parts of the system.

⑦ **Incremental** 解决方案:

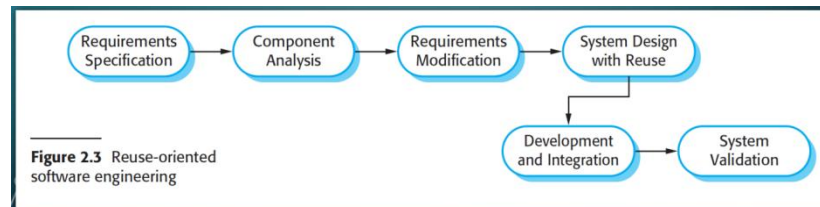
1) Large systems need a **stable framework or architecture and the responsibilities of the different teams** working on parts of the system need to be **clearly defined** with respect to that architecture.

2) This has to be planned in advance rather than developed

incrementally.

### (3) REUSE-ORIENTED SOFTWARE ENGINEERING

① Reuse-oriented approaches rely on a large base of **reusable software components** and an **integrating framework for the composition of these components**. (使用现有的已有的组件来搭建)



②

③ **Reuse** 应用场景:

1) **Web services** that are developed according to service standards and which are available for remote invocation.

2) **Collections of objects** that are developed as a package to be **integrated with a component framework** such as .NET or J2EE.

3) **Stand-alone software systems** that are configured for use in a particular environment.

④ **Reuse** 优点:

1) **reducing the amount of software** to be developed and so reducing cost and risks.

2) usually also **leads to faster delivery** of the software.

⑤ **Reuse** 缺点:

1) However, **requirements compromises** are inevitable, and this may lead to a system that **does not meet the real needs** of users.

2) Furthermore, some control over the system evolution is lost as **new versions of the reusable components are not under the control of the organization using them**.

⑥ 该模型需要进行两次需求工程，一次测试组件和系统的功能，一次测试是否满足客户需要

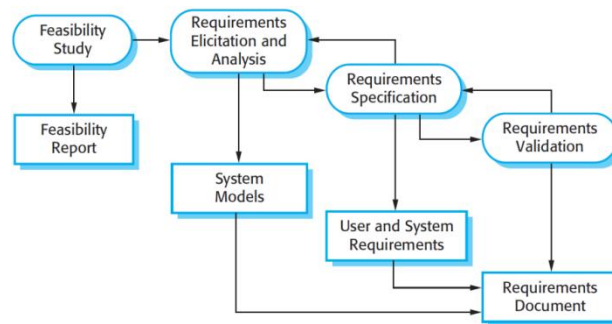
### (4) Software process activities

① Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system.

② Four basic activities:

1) **Specification (Requirement engineering)**

## W5L4)



a.

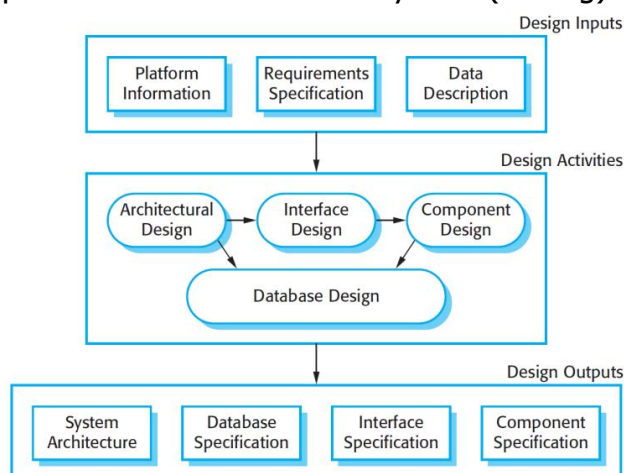
(Four

blocks above are **main** activities of requirement engineering)

b. **End-users** and customers need a **high-level statement** of the requirements; **System developers** need a more **detailed system specification**.

## 2) Development (Design and Implementation)

a. Specification -> Executable system (Coding)



b.

c. A software design is a **description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components** and, sometimes, **the algorithms used**.

d. Design activities:

- Architectural design
- Interface design
- Component design
- Database design

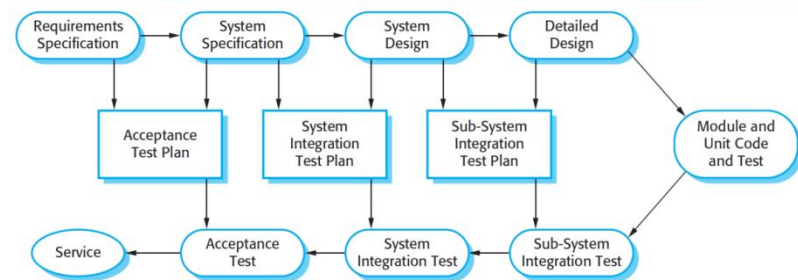
e. **Model-driven development (MDD):**

Software development tools may be used to generate a skeleton program from MDD. This includes code to define

and implement interfaces.

### 3) Validation

a. 确保其符合 specification 与 costumer's requirement.



b.

c. A three-stage testing process involved testing for **system components** then the **testing for an integrated system** and, finally, the **testing of the system with the customer's data**.

d. **Stages:**

- a) Development testing(开发人员进行组件独立测试)
- b) System testing(集成为系统测试)
- c) Acceptance testing(由系统客户提供数据进行验收测试)

### 4) Evolution

## TTL - 2 题型

1. 问管理系统开发的基础:

(1) 简单说题目要求的类型

(2) 因此使用开发模式 (**plan**[大量前期准备]、**agile**[多接口, 频繁换要求、复杂、]、**reuse**[要求被熟知、与其他系统结合使用])

(3) 具体模型 (**waterfall**)

1 (2) 也可以作为模型的优点

2. 实时系统用 **plan-driven** 开发, 因为需要大量硬件, 硬件不便更改。



## W4 - Lecture 3: Agile method

### 1. Why RAD?

- (1) Businesses now operate in a global, rapidly changing environment. They **have to respond to new opportunities and markets, changing economic conditions, and the emergence of competing products** and services
- (2) Impossible to derive a complete set of stable software requirements. The initial requirements inevitably change.
- (3) Plan-driven 不能做到快速开发软件

### 2. RAD Fundamental characteristics:

- (1) The processes are interleaved.
- (2) Minimum documentation
- (3) Developed in a series of versions, or increments, with system stakeholders involvement.
  
- (4) System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created.

### 3. 其他 RAD 方法

- (1) Adaptive software development
- (2) Agile methodologies
- (3) Spiral model
- (4) Unified software development process

### 4. Plan-driven:

- (1) **Large, long lived software - careful project planning, formalized quality assurance**, the use of analysis and design methods supported by **CASE tools**, and **controlled and rigorous software development processes**
- (2) **Characteristic:**
  - ① Work of multiple development teams has to be well coordinated
  - ② The system is a **critical system**
  - ③ Many different people will be involved in maintaining the software **over long period of time**

### 5. Agile methods

- (1) Agile allowed the development team to focus on the software itself

rather than on its design and documentation. Universally rely on an incremental approach to software specification, development, and delivery. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system.

## (2) Agile manifesto:

- ① Individuals and interactions over processes and tools
- ② Working software over comprehensive documentation
- ③ Customer collaboration over contract negotiation
- ④ Responding to change over following a plan

## (3) Principle

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

## (4) Challenges - In practice, the principles underlying

### agile methods are sometimes difficult to realize:

- ① Its success depends on having a customer who is willing and able to spend time with the development team and **who can represent all system stakeholders.**
- ② Individual team members may not have suitable **personalities** for the intense involvemem.
- ③ **Prioritizing changes** can be extremely difficult, especially in systems for which there are many stakeholders.
- ④ **Maintaining simplicity** requires extra work.
- ⑤ It is difficult for some organization to **accept informal processes** defined by development teams.
- ⑥ 合同难写
- ⑦ 出现问题是谁的锅（客户和开发者谁要为浪费的时间和资

源负责? )

⑧ 难以让客户参与进来

⑨ 因为依赖程序员的个人认知，所以可能不能保证团队认知一致性

**(5) maintenance(敏捷专家认为):**

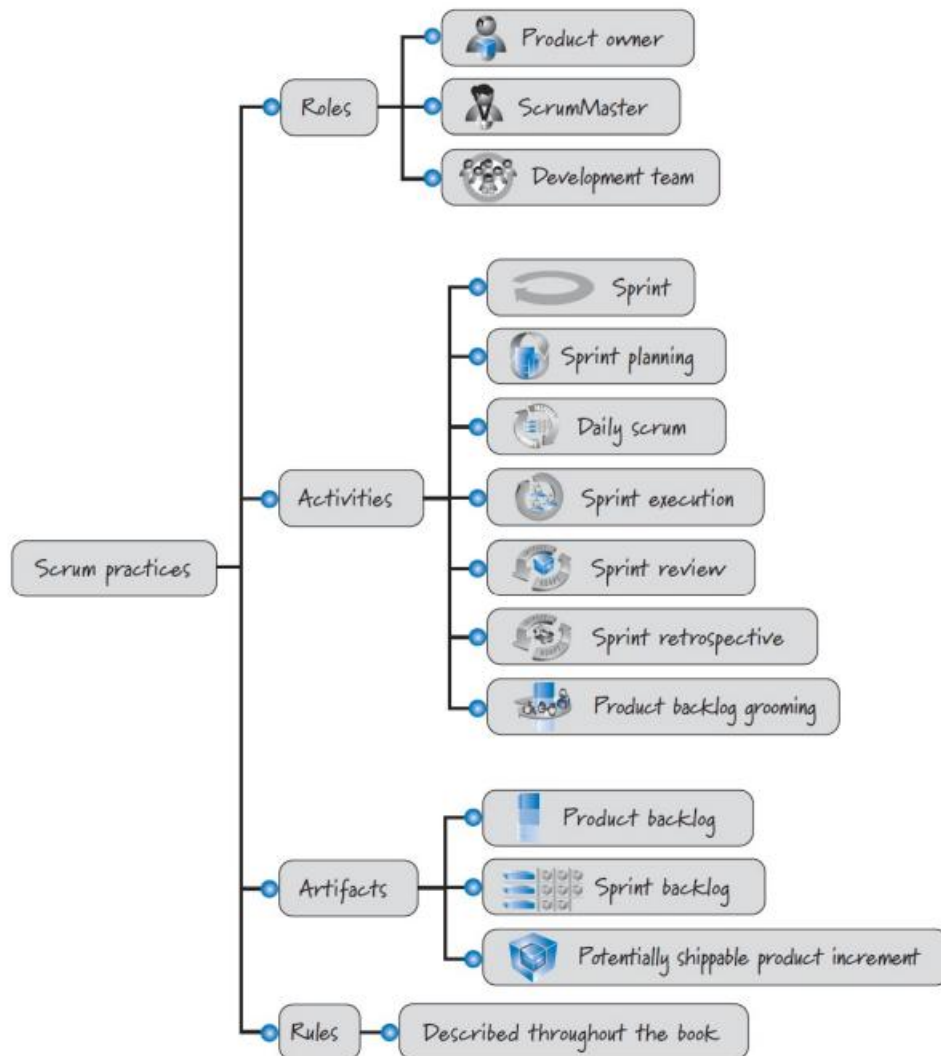
- ① The key to implementing maintainable software is to produce high-quality, readable code.
- ② The key document is the system requirements document, which tells the software engineer what the system is supposed to do.

**(6) 如何选择 plan-driven 和 agile?**

- |   |                                     |
|---|-------------------------------------|
| • Detail specification and design needed? | • Available technologies and tools? |
| • Is incrementally strategy realistic?    | • Organization of the team?         |
| • How large is the system?                | • Cultural issues?                  |
| • What type of system being developed?    | • Available skillsets?              |
| • System life span?                       | • External regulation?              |

## W4 - Lecture 3: Scrum framework

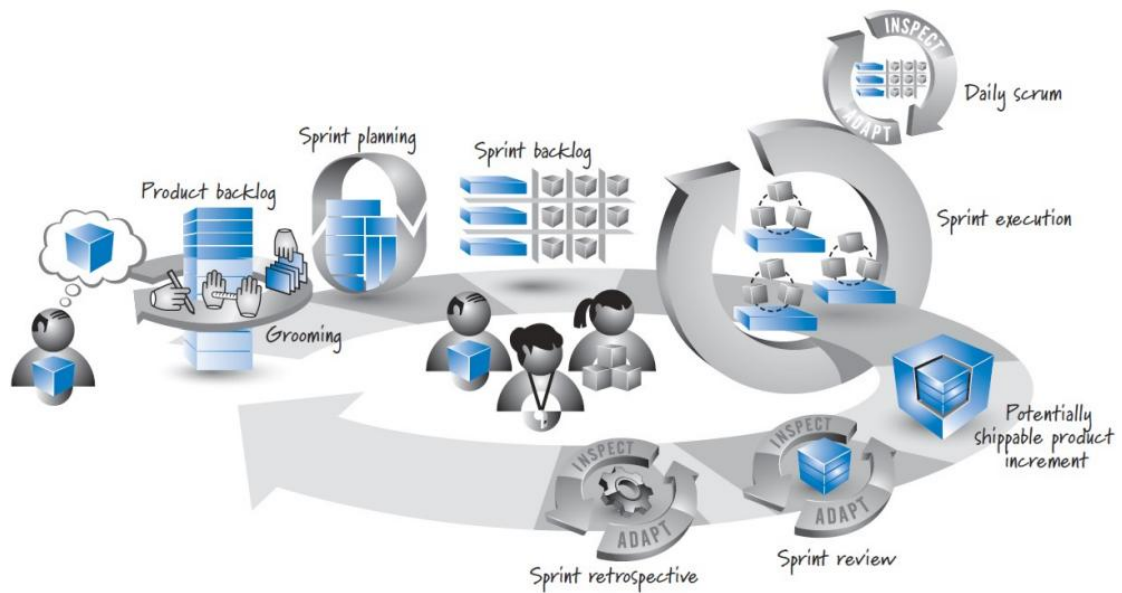
**1. Scrum is not a standardized process but a framework for organizing and managing work.**



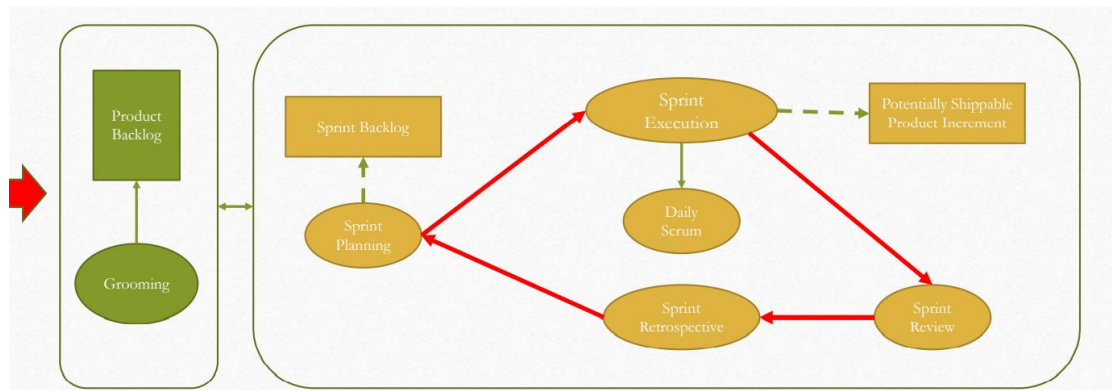
2. **FIGURE 2.1** Scrum practices

### 3. Responsibilities:

- (1) The **product owner** is responsible for what will be developed and in what order. (Manager)
- (2) The **Scrum Master** is responsible for guiding the team in creating and following its own process based on the broader Scrum framework.(Coach / Pusher /Leader)
- (3) The **development team** is responsible for determining how to deliver what the product owner has asked for.



4. **FIGURE 2.3** Scrum framework



5.

## 6. Sprint:

- (1) In Scrum, **work is performed in iterations or cycles** of up to a calendar month called sprints
- (2) The work completed in each sprint should **create something of tangible value** to the customer or user
- (3) Sprints are timeboxed so they always **have a fixed start and end date**, and generally they should all be of the same duration
- (4) A new sprint **immediately follows** the completion of the previous sprint
- (5) **Sprint Planning:**
  - ① During sprint planning, the **product owner and development team** agree on a sprint goal that defines what the upcoming sprint is supposed to achieve
  - ② **Tasks(From targets in PBI), along with their associated product backlog items, forms a second backlog called the sprint backlog.**
- (6) **Sprint Execution:** Once the Scrum team finishes sprint planning, the

**development team performs all of the task-level work necessary to get the features done.**

**(7) Daily Scrum:** 由 **Scrum master** 主持，回答三个问题：

- ① What did I accomplish since the last daily scrum?
- ② What do I plan to work on by the next daily scrum?
- ③ What are the obstacles or impediments that are preventing me from making progress?

## **7. Product backlog:**

- (1) PO** determining and managing the sequence of works (**product backlog items**) and communicating it in the form of a **prioritized (or ordered) list** known as the product backlog.
- (2)** In practice, many teams use a relative size measure such as **story points** or **ideal days** to **express the item size**.
- (3) Example:**

### PBI Example

---

- User Story: Online user registration
- Description: As a user, I want to be able to register online, so that I can perform online shopping
- Acceptance Criteria:
  - User can register only if the user fills in all required fields
  - The email used in the registration must not be a free email
  - User will receive a notification email after successful registration

- (4)** The activity of **creating and refining product backlog items, estimating them, and prioritizing them** is known as **Product Backlog Grooming**.

## **W5 - Lecture 4: Requirement Engineering**

### **1. System requirements and User requirements**

- (1)** User requirements - Statements in a natural language plus diagrams to describe the services and constraint of a system
- (2)** System requirements - more detailed descriptions of the software system's functions, services, and operational constraints



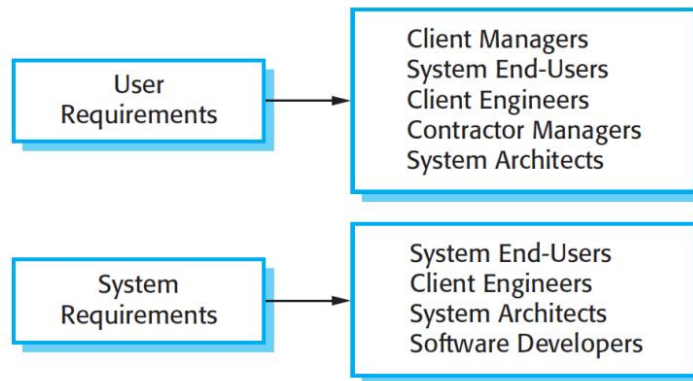
#### User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

#### System Requirements Specification

- 1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
- 1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
- 1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.
- 1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.
- 1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

(3)



(4)

## 2. Functional and Non-functional requirement

### (1) Functional requirement:

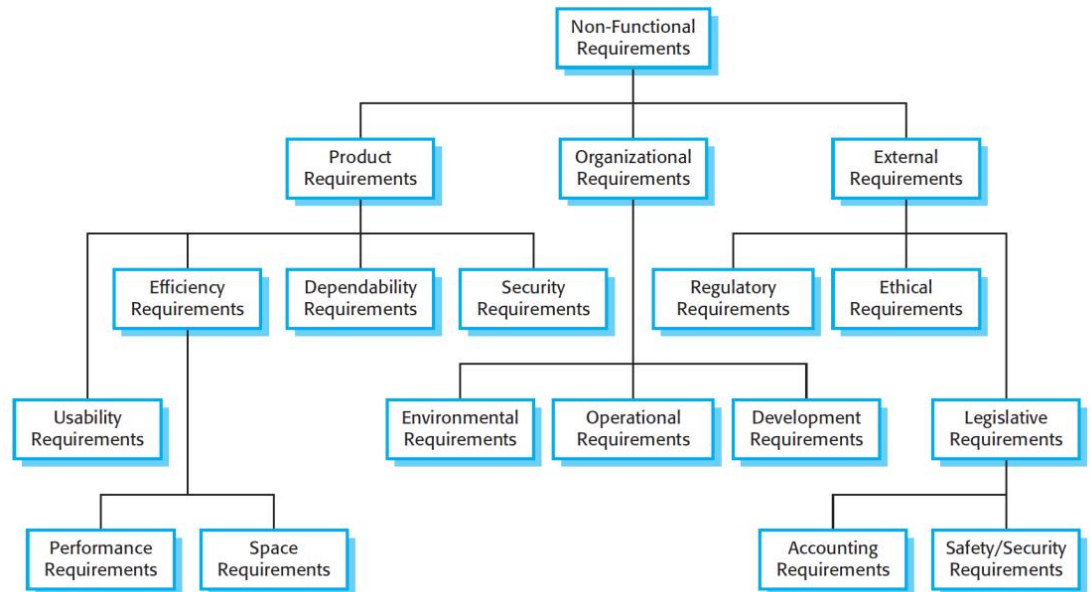
- ① When expressed as **user requirements**, functional requirements are usually described in an abstract way that can be understood by system users.
- ② More specific functional system requirements describe the system functions, its **inputs and outputs, exceptions**, etc., in detail.

### (2) Non-functional requirement:

- ① Requirements that are **not directly concerned with the specific** services delivered by the system to its users.
- ② System properties such as **reliability, response time, and store occupancy**.
- ③ Constraints on the system implementation such as **performance, security, or availability**.
- ④ Usually specify **constraint characteristics** of the system as a

whole

- ⑤ Often more **critical** than individual functional requirements, failing to meet a non-functional requirement can mean that the **whole system is unusable**



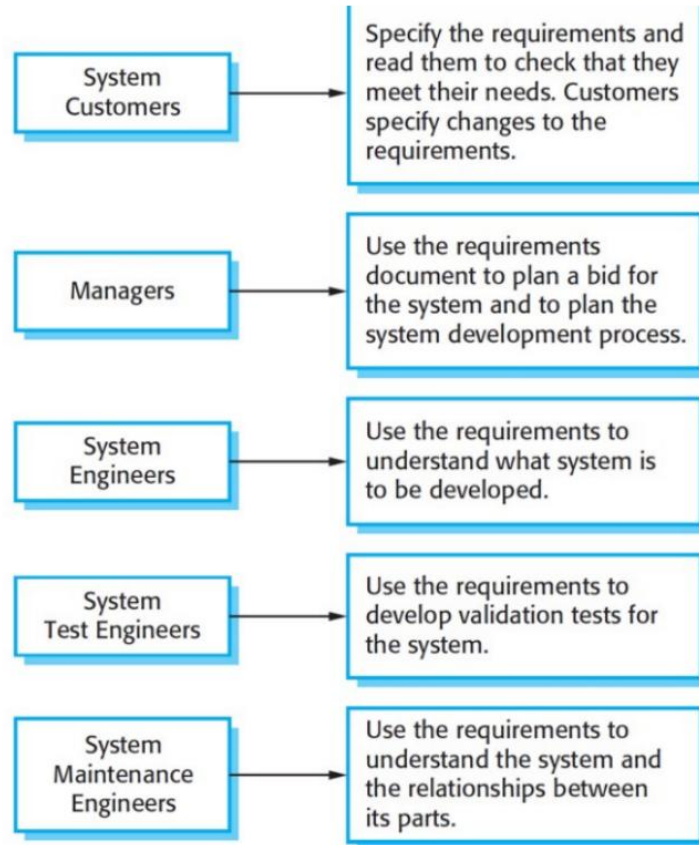
- ⑥
- 1) Product requirements - These requirements specify or **constrain the behavior of the software**. (软件运行的要求)
  - 2) Organizational requirements - These requirements are broad system **requirements derived from policies** and procedures in the customer's and developer's organization. (领域的政策要求)
  - 3) External requirements - This broad heading covers all requirements that are **derived from factors external to the system** and its development process. (隐私协议等)

⑦ **Non-functional requirement** 的类型

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

### 3. Software Requirement Specification (SRS)





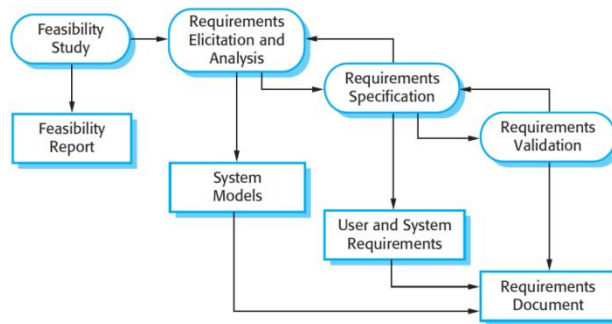
(1) 作用:

(2) **Characteristic:**

- ① Correct
- ② Complete
- ③ Unambiguous
- ④ Verifiable
- ⑤ Consistent
- ⑥ Ranked for importance and/or stability
- ⑦ Modifiable
- ⑧ Traceable

(3) **Level of details:**

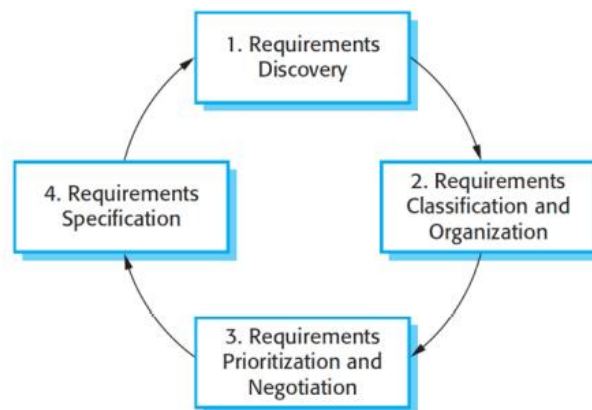
- ① Detailed requirements
  - 1) **Critical systems**
  - 2) System is to be developed by a separate company
- ② Less detailed requirements
  - 1) Inhouse(开发公司内部的需求)
  - 2) Iterative development process



(4)

Four high-level(Main) activities:

- ① Assessing if the system is useful to the business (**feasibility study**)
- ② Discovering requirements (**elicitation and analysis**)



1)

- 2) 软件开发人员与客户一起商讨，了解相关领域的各种知识与政策，并对要求进行超多次沟通与协商。

3) **Challenges:**

- a. Stakeholders often don't know what they want from a computer system except in the most **general terms**.
- b. Stakeholders in a system express requirements in **their own terms** and with **implicit knowledge** of their own work.
- c. Different stakeholders have different requirements and priorities.
- d. Political factors may influence the requirements of a system.
- e. The economic and business environment is dynamic.

4) **Requirement Discovery:**

- a. Interviews 访谈
- b. Observation 观察
- c. **Scenarios** 情景

**INITIAL ASSUMPTION:**

The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

**NORMAL:**

The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).

**WHAT CAN GO WRONG:**

The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

**OTHER ACTIVITIES:**

Record may be consulted but not edited by other staff while information is being entered.

**SYSTEM STATE ON COMPLETION:**

User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.

**d. Prototypes 原型****e. Use case**

a) A use case can then be linked to other models in the UML that will develop the scenario in more detail.

③ Converting these requirements into some standard form

**(specification)**

1) Specify only the external behavior of the system.

2) Should not include details of the system architecture or design

④ Checking that the requirements actually define the system that the customer wants **(validation)**

1) Validity checks (客户提出要求的功能是否完成, 是否冗余)

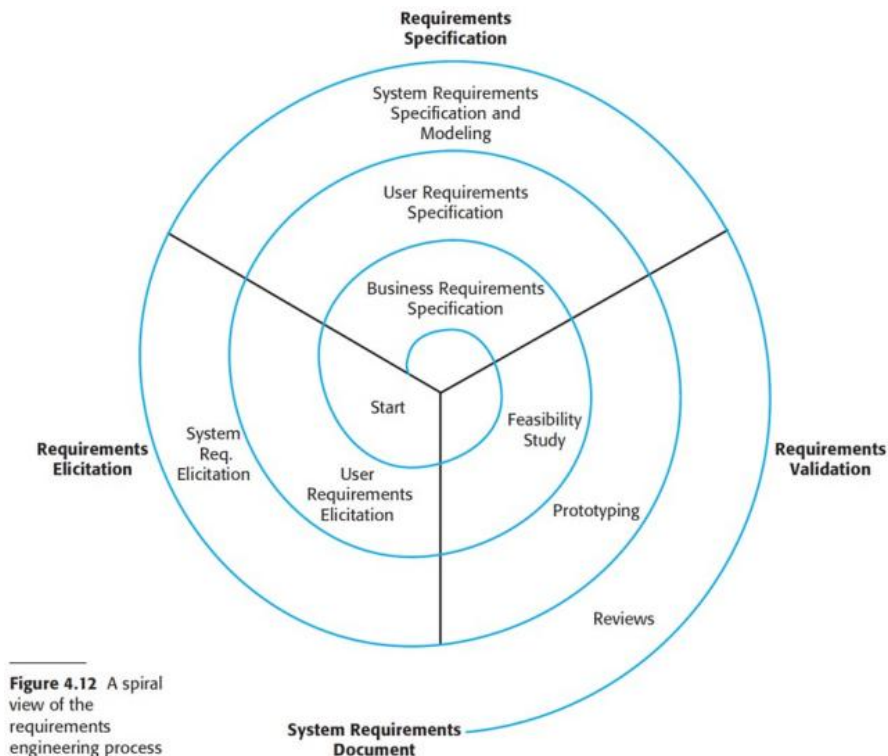
2) Consistency checks (不能有不统一的或冲突的描述)

3) Completeness checks (所有功能的需求与约束)

4) Realism checks (是否可实现)

5) Verifiability (可验证, 减少争议)

**(5) 迭代过程:**



## W6/7 - Lecture 5: Modeling

### 1. Unified Modeling Language (UML):

System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

### 2. System Perspective :

- (1) An external perspective, where you model the context or environment of the system.
- (2) An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
- (3) A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- (4) A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

### 3. Use of diagram models:

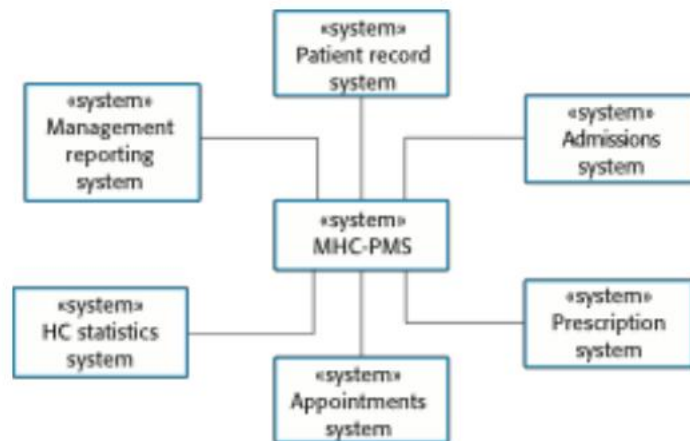
- (1) As a means of facilitating discussion about an existing or proposed system
- (2) As a way of documenting an existing system
- (3) As a detailed system description that can be used to generate a system

implementation

## 4. Context model and Interactive model(模型):

### (1) Context model:

- ① The operational context of a system
- ② System boundaries are established to define what is inside and what is outside the system.



- ③
- ④ Context models simply show the other systems in the environment, not how the system being developed is used. Used along with other models, such as business process models. **UML activity diagrams may be used to define business process models.**

### (2) Interactive model:

- ① Modeling user interaction is important as it **helps to identify user requirements.**
- ② Modeling system-to-system interaction highlights the communication **problems that may arise.**
- ③ Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required **system performance and dependability.**
- ④ **Use case diagrams and sequence diagrams** may be used for interaction modeling.

### (3) Structural models:

- ① Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.

### (4) Behavioral models:

- ① Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment
- ② **Data:** Some data arrives that has to be processed by the system.
- ③ **Events:** Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

**(5) Data-driven modeling:**

- ① Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively **little external event** processing.
- ② **Activity diagram** and **sequence diagram** are used in data-driven modeling.

**(6) Event-driven modeling**

- ① Real-time systems are often event-driven, with **minimal data processing**. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone.
- ② Event-driven modeling shows how a system responds to **external and internal events. (State machine)**

## 5. Use case diagram:

- (1)** Each use case represents a discrete task that involves external interaction with a system.
- (2)** Actors in a use case may be people or other systems.
- (3)** Example:

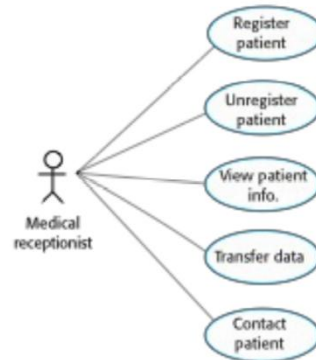
**Tabular description of the 'Transfer data' use-case**



MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the MHC-PMS to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

- (4)** Example:

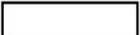
### Use cases in the MHC-PMS involving the role 'Medical Receptionist'

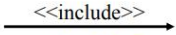


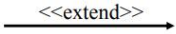
### (5) Detailed points:

- ① A picture, diagrams are not essential, text is essential.
- ② Actor:
  - 1) All user roles that interact with the system and system components only if they responsible for initiating/triggering a use case.
  - 2) **primary** - a user whose goals are fulfilled by the system. [define user goals]
  - 3) **supporting** - provides a service (e.g., info) to the system. [clarify external interfaces and protocols]
  - 4) **offstage** - has an interest in the behavior but is not primary or supporting, e.g., government. [ensure all interests (even subtle) are identified and satisfied ]
- ③ Each Actor must be linked to a use case, while some use cases may not be linked to actors.

———— Connection between Actor and Use Case

 Boundary of system

 **Include** relationship between Use Cases (one UC must call another; e.g., Login UC includes User Authentication UC)

 **Extend** relationship between Use Cases (one UC calls Another under certain condition; think of if-then decision points)

④

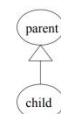


- **Association** relationships
- **Generalization** relationships
  - One element (child) "is based on" another element (parent)
- **Include** relationships
  - One use case (base) includes the functionality of another (inclusion case)
  - Supports re-use of functionality
- **Extend** relationships
  - One use case (extension) extends the behavior of another (base)

⑤

## 1. Generalization

- The child use case inherits the behavior and meaning of the parent use case.
- The child may add to or override the behavior of its parent.



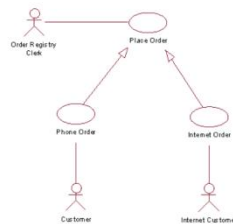
(继承关系)

⑥

## Generalization Example

The actor Order Registry Clerk can instantiate the general use case Place Order.

Place Order can also be specialized by the use cases Phone Order or Internet Order.



## 2. Include



- The base use case explicitly incorporates the behavior of another use case at a location specified in the base.
- The included use case never stands alone. It only occurs as a part of some larger base that includes it.

⑦

## Include relationship

- **Include relationship** – a standard case linked to a **mandatory** use case.
- Example: to *Authorize Car Loan* (standard use case), a clerk must run *Check Client's Credit History* (include use case).
- The standard UC includes the mandatory UC (use the verb to figure direction arrow).
- Standard use case can NOT execute without the include case → **tight coupling**.



### 3. Extend



- The base use case implicitly incorporates the behavior of another use case at certain points called extension points.
- The base use case may stand alone, but under certain conditions its behavior may be extended by the behavior of another use case.

⑧

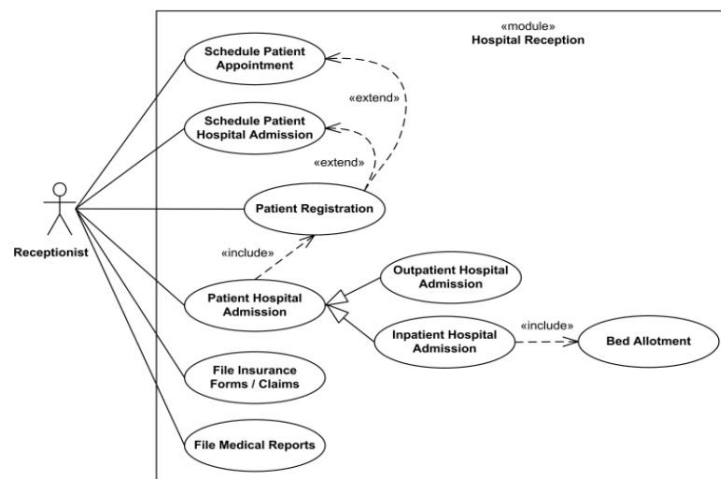
**Extend** 的箭头指向与 **Include** 相反

### Extend relationship

- **Extend** relationship – linking an **optional** use case to a standard use case.
- Example: *Register Course* (standard use case) may have *Register for Special Class* (extend use case) – class for non-standard students, in unusual time, with special topics, requiring extra fees...).
- The optional UC extends the standard UC
- Standard use case can execute without the extend case  
→ loose coupling.

⑨ LEC 课件中的 **example** 非常有价值。

### Extend Example #1



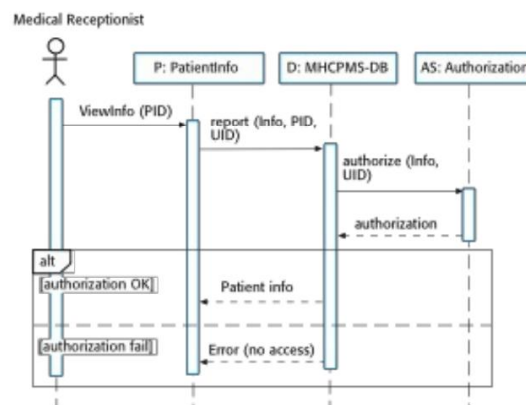
## 6. Sequence diagram:

- (1) A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- (2) The objects and actors involved are listed along the top of the diagram,

with a dotted line drawn vertically from these.

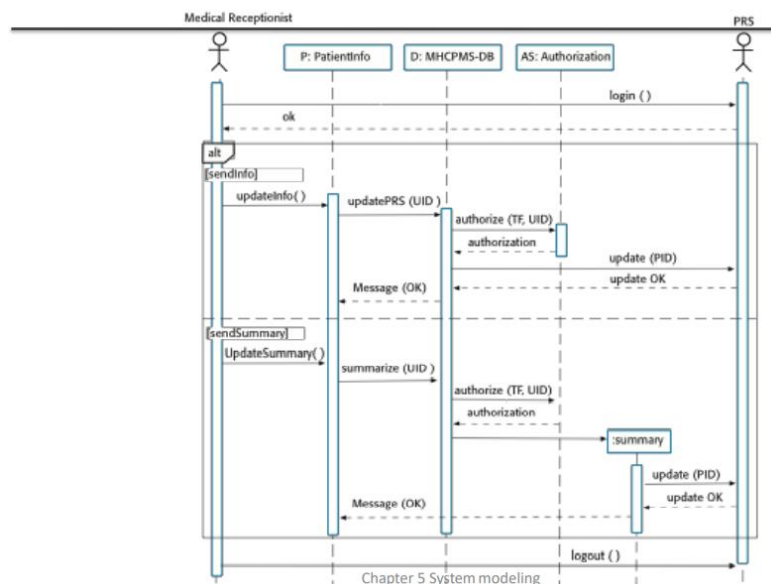
### (3) Example:

#### Sequence diagram for View patient information

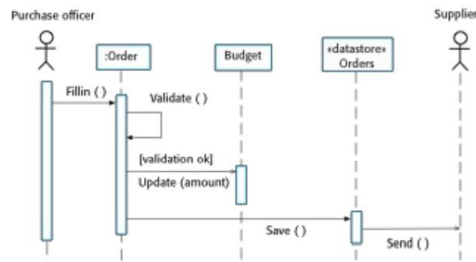


### (4) Example:

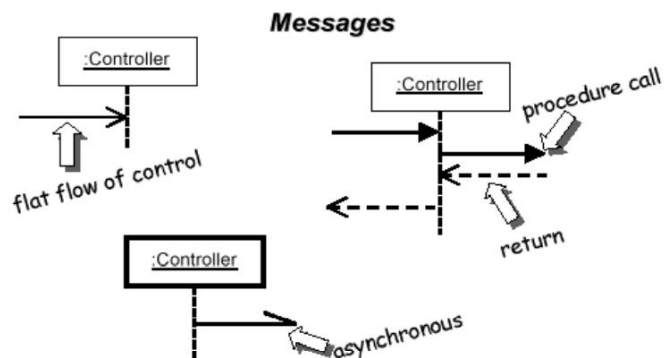
#### Sequence diagram for Transfer Data



## Sequence diagram of an order processing



(5) Example:



(6)

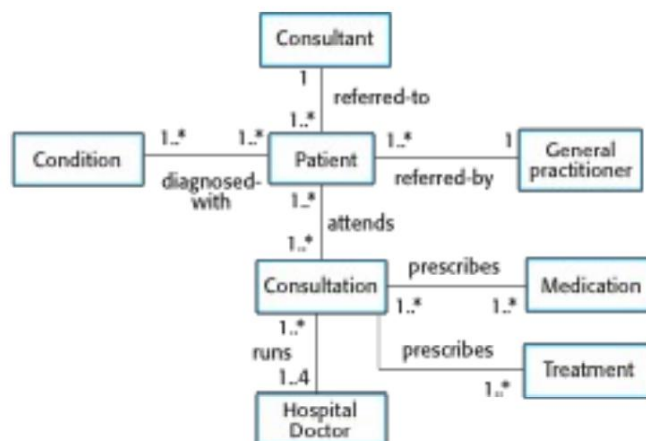
[流控制：发后等待

return，过程控制：发后不理]

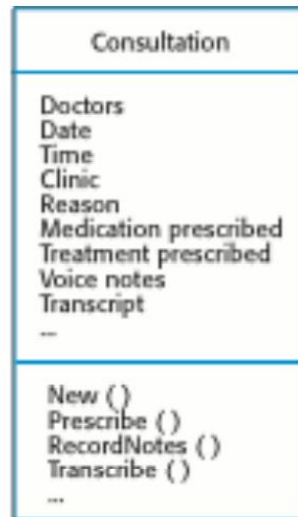
## 7. Class diagram:

(1) Class diagrams are used when developing an objectoriented system model to show the classes in a system and the associations between these classes.

Classes and associations in the MHC-PMS

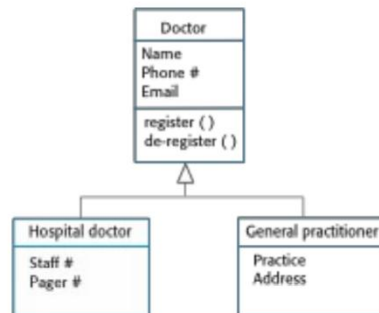


(2)



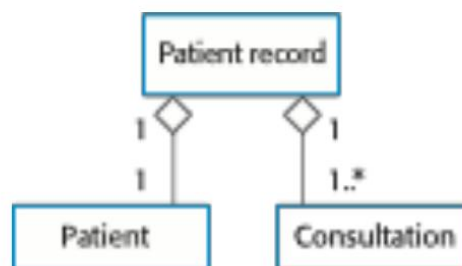
### (3) A kind of Class:

A generalization hierarchy with added detail



### (4)

### The aggregation association



[菱形符号，代表包含或拥有('have')]

有('have')]

[The lifecycle of both objects are independent of each other(可以一对多，不依赖)]

## Composition

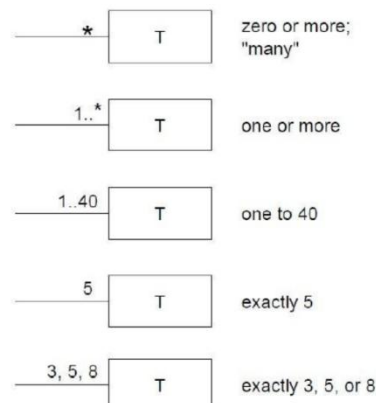


- **Composition**

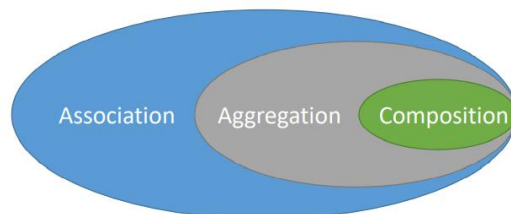
- A special type of Association, “Strong Association”
- “part of” relationship
- The containing object, the Company, exclusively “own” the Dept object
- When containing object ended it’s lifecycle, the Dept object end with in

(5)

Examples of  
Multiplicity



(6)

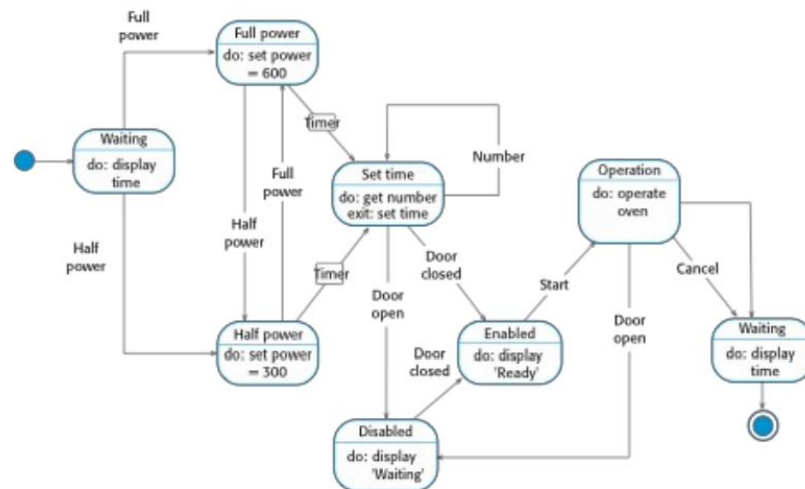


(7)

## 8. State machine diagram:

- (1) They show the system’s **responses to stimuli** so are often used for modelling real-time systems.
- (2) **When an event occurs, the system moves from one state to another.**
- (3) **Statecharts(状态图)** are an integral part of the UML and are used to represent state machine models.
- (4) Example:

## State diagram of a microwave oven



• A *passive state* is simply the current status of all of an object's attributes

- For example: Attributes of Student class
  - Student ID, name, enrolled date,.....

• The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing.

- For example: Status of Student class
  - New, Enrolled, Suspended, Graduated

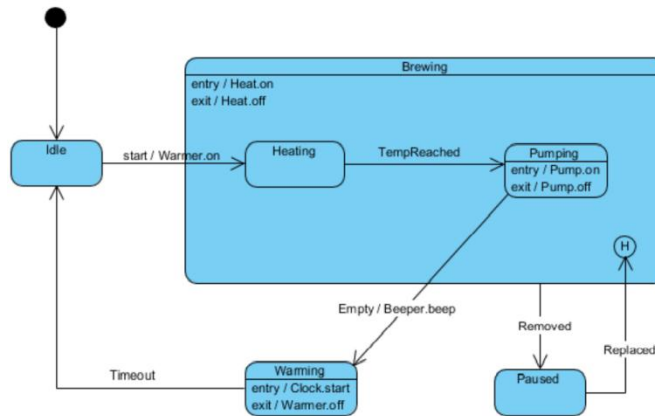
(5)

(6) An event (sometimes called a trigger ) must occur to force an object to make a transition from one active state to another.

### Example: Modeling an automatic coffee maker



(7) Example:

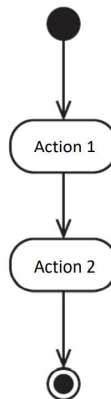


## 9. Activity diagram:

### Basic

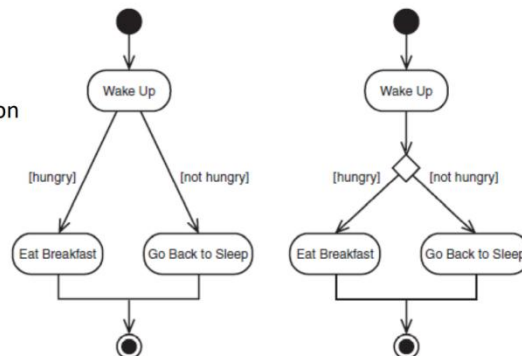
- An **activity** represents an operation, a step in a business process, or an entire business process.
- Action is a named element which represents a **single atomic step within activity** i.e. that is **not further decomposed within the activity**.
- Arrow represents the transition from one activity to the next
- Filled circle represent start, bull's eye represent an endpoint

(1)



### Decision

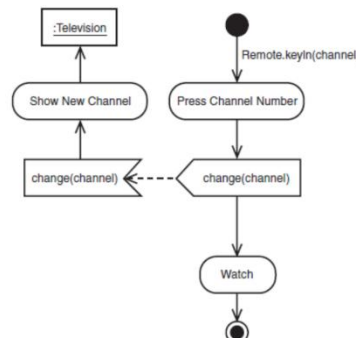
- Both are valid
- Label the condition with a squared bracket



(2)

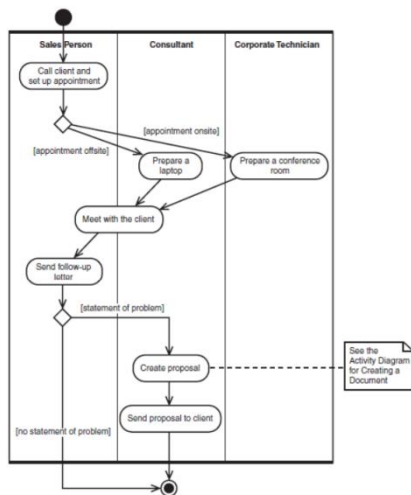
### Sending and Receiving Signal

- There will be situation when your process need to send or receive signal
- Convex polygon – sending signal / output event
- Concave polygon – receiving signal / input event



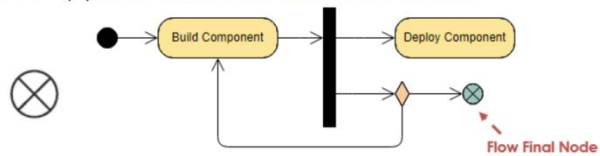
(3)

#### (4) 泳道(显示 activity 的 actor):



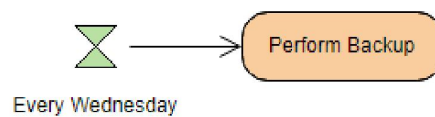
#### Flow Final Node

- Not Activity Final Node
- UML 2.0 has an additional control node type called Flow Final that is used as an alternative to the Activity Final node to terminate a flow.
- It is needed because in UML 2.0, when control reaches any instance of Activity Final node, the entire activity (including all flows) is terminated. The Flow Final simply terminates the flow to which it is attached.



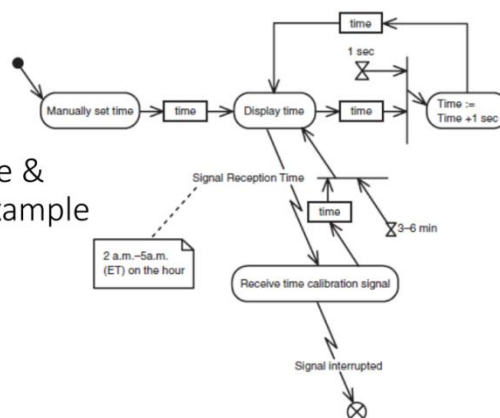
#### (5)

- Time event flows when the time expression is true



#### (6)

#### Flow Final Node & Time Event - Example

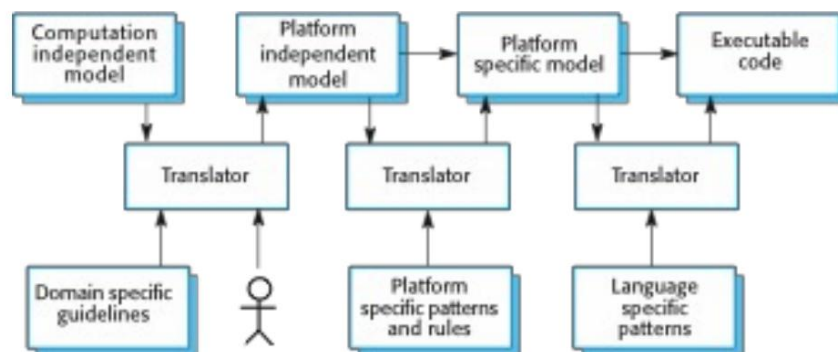




## 10. Model-driven engineering

- (1) Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
- (2) The programs that execute on a hardware/software platform are then **generated automatically from the models**.
- (3) 优点:
  - ① Allows systems to be considered at higher levels of abstraction
  - ② Generating code automatically means that it is cheaper to adapt systems to new platforms.
- (4) 缺点:
  - ① Models for abstraction and not necessarily right for implementation.
  - ② Savings from generating code may be outweighed by the costs of developing translators for new platforms。

### MDA transformations



(5)

## Key points



- ✧ A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behavior.
- ✧ Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
- ✧ Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- ✧ Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.

## Key points



- ✧ Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system, or by the events that stimulate responses from a system.
- ✧ Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- ✧ State diagrams are used to model a system's behavior in response to internal or external events.
- ✧ Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

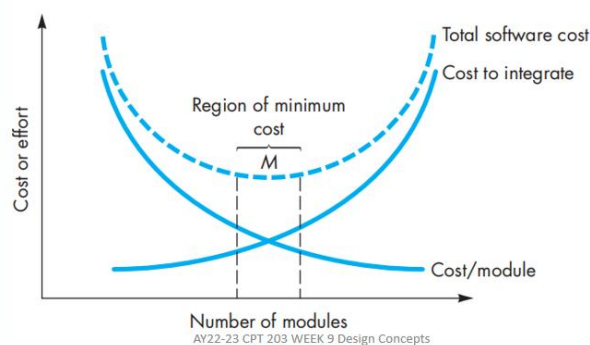
## Week 9 Design Concepts

1. The goal of design is to produce a model or representation that exhibits:
  - (1) **Firmness (no bugs)**
  - (2) **Commodity (useful or valuable)**
  - (3) **Delight (pleasurable experience)**
2. Software design is:
  - (1) **The last software engineering action within the modelling activity and sets the stage for construction (code generation and testing).**
  - (2) **A process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.**
3. Design architecture:
  - (1) **Component-level design:** transforms **structural elements** of the software architecture into a **procedural description of software components**.
  - (2) **Interface design:** describes how the software communicates with **systems** that interoperate with it, and with **humans** who use it
  - (3) **Architectural design :** defines the relationship between **major structural elements** of the software, the **architectural styles and patterns** that can be used to achieve the requirements defined for the system, and the **architectural constraints** that affect the way in which architecture can be implemented.
  - (4) **Data design/class design:** transforms class models into **design class** realizations and the requisite **data structure** required to implement it.
4. Design goals of good quality:
  - (1) The design should implement **all of the explicit requirements contained in the requirements model**, and it must accommodate all of the implicit requirements desired by stakeholders.
  - (2) The design should be a **readable, understandable** guide for those who generate code and for those who test and subsequently support the software.
  - (3) The design should **provide a complete picture of the software**, addressing the data, functional, and behavioral domains from an implementation perspective
5. Design criteria:
  - (1) A design should exhibit an architecture that (1) has been created using **recognizable architectural styles or patterns**, (2) is composed of **components that exhibit good design characteristics**, and (3) can be implemented in an **evolutionary fashion**, thereby facilitating implementation and testing.
  - (2) A design should be **modular**; that is, the software should be logically partitioned into elements or subsystems.
  - (3) A design should **contain distinct representations** of data, architecture, interfaces, and components.

- (4) A design should lead to **data structures** that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- (5) A design should lead to **components that exhibit independent functional characteristics**.
- (6) A design should lead to **interfaces that reduce the complexity of connections** between components and with the external environment.
- (7) A design should be derived using a **repeatable method that is driven by information obtained during software requirements analysis**.
- (8) A design should be represented using a **notation** that effectively communicates its meaning.
- (9) FURPS: Functionality/Usability/Reliability/Performance/Supportability

## 6. Design concepts:

- (1) **Abstraction:** Simply means to hide the details to reduce complexity and increases efficiency or quality. As different levels of abstraction are developed, you work to create both **procedural and data abstractions**.
- (2) **Modularity:** It basically clusters similar or relative functions together, sets up boundaries and provides interfaces for communication, which increases manufacture efficiency and save time.



- (3) **Information hiding:** Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information, which provides the greatest benefits when modifications are required during testing and later during software maintenance.
- (4) **Functional independence:** is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Criteria: coupling and cohesion.
- (5) **Coupling:** Two modules are considered independent if one can function completely without the presence of the other. To keep coupling low we would like to minimize the number of interfaces per module and the complexity of each interface.
- (6) **Cohesion:** Cohesion is the degree of how closely the elements of a module are related to each other
- (7) **Object-Orient Design:** system is made up of interacting objects that maintain their own local state and provide operations on that state, involve designing object classes and the relationships between these classes.
  - ① Understand and define the context and the external interactions with the system.

- ② Design the system architecture.
  - ③ Identify the principal objects in the system.
  - ④ Develop design models
  - ⑤ Specify interfaces
- (8) **Design classes:** define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
- ① **User interface classes** define all abstractions that are necessary for human-computer interaction (HCI) and often implement the HCI in the context of a metaphor.
  - ② **Business domain classes** identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or more analysis classes.
  - ③ **Process classes** implement lower-level business abstractions required to fully manage the business domain classes.
  - ④ **Persistent classes** represent data stores (e.g., a database) that will persist beyond the execution of the software.
  - ⑤ **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.
  - ⑥ **A good design class:**
    - 1) **Complete and sufficient.** A design class should be the **complete** encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. **Sufficiency** ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.
    - 2) **Primitiveness.** Methods associated with a design class should be focused on accomplishing one service for the class.
    - 3) **High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
    - 4) **Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum.
- (9) **Elements:**
- ① **data design** (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).
    - 1) **Program-component level:** essential to the creation of high-quality applications
    - 2) **Application level:** pivotal to achieving the business objectives of a system
    - 3) **Business level:** enables data mining or knowledge discovery that can have

an impact on the success of the business itself

- ② **Architectural design:** is the equivalent to the floor plan of a house. Architectural design elements give us an overall view of the software.
- ③ **Interface design elements** for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.
- ④ **The component-level design** for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. It fully describes the internal detail of each software component.
- ⑤ **Deployment-level design elements** indicate how software **functionality and sub-systems** will be allocated within the physical computing environment that will support the software

## Week 10 Architecture/ Component level and Interfaces

### 1. Architecture design:

- (1) Is concerned with understanding how a system should be organized and designing the overall structure of that system.
- (2) It is the first stage in the software design process, and the critical link between design and requirements engineering.
- (3) It identifies the main structural components in a system and the relationships between them.
- (4) The output of this design process is a description of the software architecture.
- (5) It affects performance, robustness, distributability, and maintainability.
- (6) Views:
  - ① A **logical view**, which shows the key abstractions in the system as objects or object classes.
  - ② A **process view**, which shows how, at run-time, the system is composed of interacting processes.
  - ③ A **development view**, which shows how the software is decomposed for development.
  - ④ A **physical view**, which shows the system hardware and how software components are distributed across the processors in the system.
- (7) **MCV pattern:** It includes three major interconnected components:
  - ① Model: central component of the pattern that directly manages the data, logic and rules of the application.
  - ② View: can be any output representation of information, such as a chart or a diagram.
  - ③ Controller: accepts input and converts it to commands for the model or view, enables the interconnection between the views and the model
- (8) **Layered pattern:**
  - ① The system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it.
  - ② This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made

available to users

- ③ Performs poorly in the high-performance applications, because it is not efficient to go through multiple layers to fulfil a business request. It is a good choice for situations with a very tight budget and time constraints.

**(9) Responsibility patterns:** All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.

**(10) Client-Server patterns:** In a client-server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.

**(11) Pip and Filter patterns:** The processing of the data in a system is organized so that each processing component(filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.

## **2. Component level design:**

**(1)** Component-level design occurs after the first iteration of architectural design has been completed.

**(2)** A complete set of software components is defined during architectural design. But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code (in architecture design).

**(3)** Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.

### **(4) Component:**

- ① A software component is a modular building block for computer software.
- ② It can be used to review for correctness and consistency with other components.
- ③ It can be used to access whether data structure, interfaces and algorithms will work
- ④ It should provide sufficient information to guide implementation.

### **(5) Design process steps:**

- ① Identify all design classes that correspond to the problem domain.
- ② Identify all design classes that correspond to the infrastructure domain.
- ③ Elaborate all design classes that are not acquired as reusable components.
  - 1) Specify message details when classes or components collaborate.
  - 2) Identify appropriate interfaces for each component.
  - 3) Elaborate attributes and define data types and data structures required to implement them.
  - 4) Describe processing flow within each operation in detail.
- ④ Describe persistent data sources (databases and files) and identify the classes required to manage them.
- ⑤ Develop and elaborate behavioral representations for a class or component.
- ⑥ Elaborate deployment diagrams to provide additional implementation detail.
- ⑦ Refactor every component-level design representation and always consider alternatives.

### 3. Interface design:

(1) User interface design creates an **effective communication medium** between a human and a computer.

(2) **Golden rules:**

- ① Place the user in control
- ② Reduce the user's memory
- ③ Make the interface consistent

(3) **Interface analysis**

- ① User analysis: studying what type of person might use the application or product
- ② Task analysis: studying how users accomplish the tasks they set out to perform using a software system

(4) **Design steps:**

- ① Using information developed during interface analysis, define interface objects and actions (operations).
- ② Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
- ③ Depict each interface state as it will actually look to the end-user.
- ④ Indicate how the user interprets the state of the system from information provided through the interface.

(5) **Evaluation:**

- ① User testing: time-consuming, expensive, essential. Prepare -> Conduct session -> Analyse result

(6) **Design and Implementation:** is the stage in the software engineering process at which an executable software system is developed.

- ① Design is to identify software components and their relationships, based on a customer's requirements. Implementation is the process of realizing the design as a program.
- ② These two activities are invariably interleaved.
- ③ Implementation issues:
  - 1) **Reuse:** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - 2) **Configuration management:** During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - 3) **Host-target development:** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).
  - 4) **Open source development:** An approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process.



## Week 11 Testing

**1.** 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. You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.

**2.** Goals:

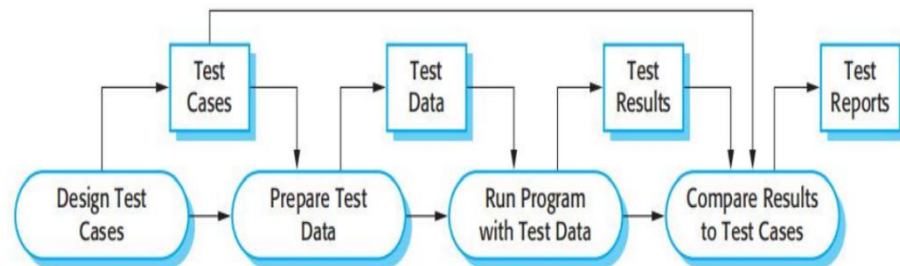
- (1)** To demonstrate to the developer and the customer that the software meets its requirements. -- leads to **validation testing**:
  - ① You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
- (2)** To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification. -- leads to **defect testing** :
  - ① The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

**3.** Verification and Validation(V & V):

- (1)** These two processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software.
- (2)** Verification: Is to check that the software meets its stated functional and non-functional requirements. --"**Are we building the product right**".
- (3)** Validation: Is to ensure that the software meets the customer's expectations. -- "**Are we building the right product**".
- (4)** Depends on:
  - ① Software purpose: The level of confidence depends on how critical the software is to an organization. (safety-critical system v.s. a prototype)
  - ② User expectations: Users may have low expectations of certain kinds of software.
  - ③ Marketing environment: Getting a product to market early may be more important than finding defects in the program.
- (5) Software inspections:** Concerned with analysis of the static system representation to discover problems (static verification). **See documents and code.**
  - ① **Advantages:** 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.

- ② **Disadvantages:** Check conformance with specification but may not user requirements; Can not check non-functional characteristics. [Inspections and testing are complementary]

**Figure 8.3** A model of the software testing process



③

(6) **Software testing:** Concerned with exercising and observing product behaviour (dynamic verification). **Use test data.**

① **Development testing.**

- 1) **Unit test(Object class) [State machine diagram]-> Component test(Interface) -> System test(Integrating component and integrating system, Use case-testing)[sequence diagram]**
- 2) **Testing strategies:** **Partition** testing, where you identify groups of inputs that have common characteristics and should be processed in the same way; **Guideline-based** testing, where you use testing guidelines to choose test cases.
- 3) **Component test** focus on showing that the component interface behaves.
  - a. **Interface testing:**
    - a) **Parameter interfaces:** Data passed from one method or procedure to another.
    - b) **Shared memory interfaces:** Block of memory is shared between procedures or functions.
    - c) **Procedural interfaces:** Sub-system encapsulates a set of procedures to be called by other sub-systems.
    - d) **Message passing:** Sub-systems request services from other sub-systems.
- 4) **Different** between component testing and system testing:
  - a. 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.
  - b. Components developed by different team members or subteams may be integrated at this stage. System testing is a collective rather than an individual(Component testing).

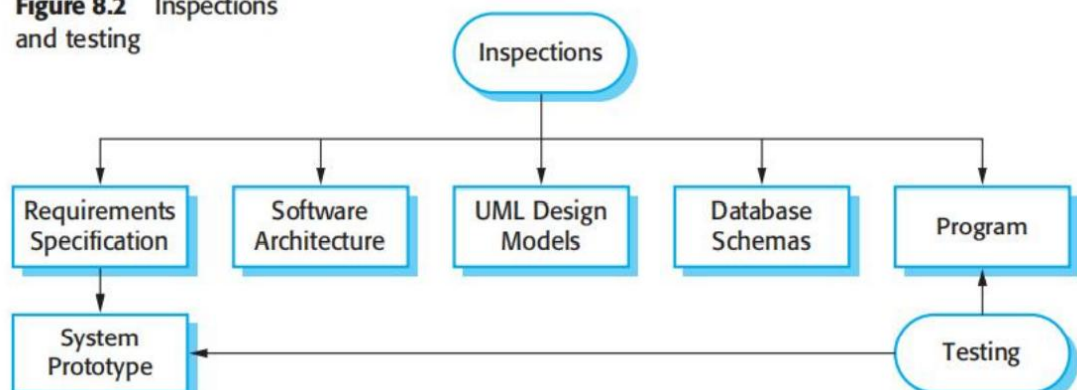
② **Release testing.**

- 1) Convince supplier.
- 2) Black-box test: tests are only derived from the system specification.
- 3) **A form of system test.**
- 4) 由没参与过开发的团队承担。
- 5) **Requirement based testing:** involves examining each requirement and developing a test or tests for it.
- 6) **Scenario testing:** To devise typical scenarios of use and use these to develop test cases for the system. A scenario is a story that describes one way in which the system might be used.
- 7) **Performance testing:** To planning a series of tests where the load is steadily increased until the system performance becomes unacceptable. **Include stress testing.**

③ **User testing.**

- 1) **Alpha testing:** Users of the software work with the development team to test the software at the developer's site.
- 2) **Beta testing:** A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- 3) **Acceptance testing:** Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.
  - a. Define acceptance criteria
  - b. Plan acceptance testing
  - c. Derive acceptance tests
  - d. Run acceptance tests
  - e. Negotiate test results
  - f. Reject/Accept system
- 4) Provide input and advice on system testing.

**Figure 8.2** Inspections and testing



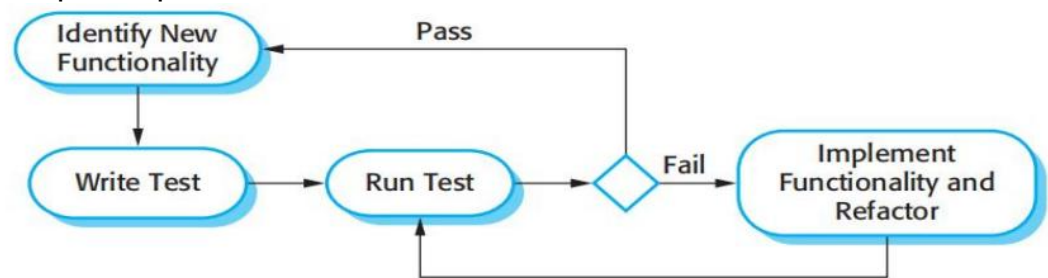
(7)

**(8) Test-driven development:**

- ① Test-driven development (TDD) is an approach to program

development in which you inter-leave testing and code development.

- ② Tests are written before code and 'passing' the tests is the critical driver of development.
- ③ You develop code incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test.
- ④ TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.



1. Start by **identifying the increment of functionality** that is required. This should normally be small and implementable in a few lines of code.
2. Write **a test for this functionality** and implement this as an automated test.
3. **Run the test, along with all other tests** that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
4. **Implement the functionality** and re-run the test.
5. Once all tests run successfully, you **move on to implementing the next** chunk of functionality.

⑤

⑥ Advantages:

- 1) Code coverage
- 2) Regression testing: testing the system to check that changes have not 'broken' previously working code.
- 3) Simplified debugging
- 4) System documentation

(9) General test guidelines:

① Sequence:

- 1) Test software with sequences which have only a single value.
- 2) Use sequences of different sizes in different tests.
- 3) Derive tests so that the first, middle and last elements of the sequence are accessed.
- 4) Test with sequences of zero length

② General:

- 1) Choose inputs that force the system to generate all error messages;
- 2) Design inputs that cause input buffers to overflow;
- 3) Repeat the same input or series of inputs numerous times;
- 4) Force invalid outputs to be generated;

- 5) Force computation results to be too large or too small
- ③ Interface:
  - 1) Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
  - 2) Always test pointer parameters with null pointers.
  - 3) Design tests which cause the component to fail.
  - 4) Use stress testing in message passing systems.
  - 5) In shared memory systems, vary the order in which components are activated.
- ④ System:
  - 1) All system functions that are accessed through menus should be tested.
  - 2) Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - 3) Where user input is provided, all functions must be tested with both correct and incorrect input.

## Week 12 Junit

1. Basic idea of unit test: For a given class ABC, create another class ABCTest to test it, containing various "test case" methods to run. • Each method looks for particular results and passes / fails.

## Week 13 Project management

### 1. Project Management:

- (1) Projects need to be managed because professional software engineering is always subject to organizational budget and schedule constraints
- (2) Ensure that the software project meets and overcomes these constraints as well as delivering high-quality software.
- (3) A bad management may result: late deliver/increase cost/fail to meet the expectations of customers

#### (4) Criteria:

- ① Deliver the software to the customer at the agreed time.
- ② Keep overall costs within budget.
- ③ Deliver software that meets the customer's expectations.
- ④ Maintain a happy and well-functioning development team..

#### (5) Challenges:

- ① The product is intangible
- ② Large software projects are often 'one-off' projects
- ③ Software processes are variable and organization specific

#### (6) Manager's responsibilities:

- ① Project Planning

- ② Reporting to customers and to the managers of the
- ③ **Risk management:** Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks.
  - 1) Project risks: Risks that affect the project schedule or resources.
  - 2) Product risks: Risks that affect the quality or performance of the software being developed.
  - 3) Business risks: Risks that affect the organization developing or procuring the software.
  - 4) Manage Process: Risk identification -> Risk analysis - > Risk planning -> Risk monitoring
- ④ People management
- ⑤ Proposal writing
- ⑥ Manager's characteristic: Motivation/Organization/Ideas or innovation/Problem solving/Managerial identity/Achievement/Influence and team building
- (7) 5 constituencies of stakeholders in software process: Senior manager, Project(technical) manager, Practitioners, Customers, End users.
  - ① Consistency, Respect, Inclusion, Honesty
- (8) **Motivation:** To satisfy Esteem Needs, Social Needs, Safety Needs, Physiological Needs
- (9) **Team organizational paradigms:** Closed paradigm/Random paradigm/Open paradigm/Synchronous paradigm