소프트웨어공학 기말고사 정리

chap 6

1. System Modeling

: Process of developing abstract models of a system, with each model presenting a differnet view or perspective of that system.

= representing a system using some kind of graphical notation(ex. Unified Modeling Language)

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

2. Existing and planned system models

1) Models of the existing system

- They used during requriements engieering.
- They help clarify what the existing system does and can be used as a basis for discussing its strenghts and weaknesses. These then lead to requirements for the new system.

2) Models of the new system

- They are used during requirements engineering to help explain the proposed requirements to other system stakeholders.
- Engineers use these models to discuss design proposals and to document the system for implementation.

3) In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

3. System perspectives

- External perspective : the context or environment of the system context(정적), activity(동적) diagram
- Interaction perspective : the interactions between a system and its environment, or between the components of a system. use case, sequence diagram
- Structural perspective : the organization of a system or the structure of the data that is processed by the system. class diagram
- Behavioral perspective : the dynamic behavior of the system and how it responds to events.
  data – activity, sequence diagram, event – state diagram

4. UML (Unified Modeling Language) diagram types 총 13개임

- Activity diagram : the activities involved in a process or in data processing
  * DFD(Data Flow Diagram) : UML에서는 객체 지향적 개념(system은 상호독립적인 object 셋)과 맞지 않아 사용하지 않음. DFD의 관점은 system은 기능의 셋으로 기능은 input data를 output data로 transform하게 해주는 function을 의미함.
- Use case diagram : the interactions between a system and its environment
- Sequence diagram : interactions between actors and the system and between system components

- Class diagram : the object classes in the system and the associations between these classes.
- State diagram : the system reacts to internal and external events.

5. Use of graphical models

- As a means of facilitating discussion about an existing or proposed system.
  : Incomplete and incorrect models are OK as their role is to support discussion.
- As a way of documenting an existing system.
  : Models should be an accurate representation of the system but need not be complete.
- As a detailed system description that can be used to generate a system implementation.
  : Models have to be both correct and complete.

6. Context Model

: show how a system that is being modeled is positioned in an evironment with other systems and processes.

(1) System Boundaries

- illustrate the operational context of a system. System boundaries are established to define what is inside and what is outside the system.
  : They show other systems that are used or depend on the system being developed.
- Social and organizational concerns may affect the decision on where to position system boundaries.
  : Defining a system boundary is a political judgment. There may be pressures to develop system boundaries that increase/decrease the influence or workload of different parts of an organization.
- Architectural models show the system and its relationship with other systems.
- The position of the system boundary has a profound effect on the system requirements.

(2) Process Perspective

- Context models simply show the other systems in the environment, not how the system being developed is used in that environment(process model).
- Process models reveal how the system being developed is used in broader business processes. UML activity diagrams may be used to define business process models.
- 즉, **context diagram**과 **activity diagram** 모두 externel한 것들을 표현하지만, context diagram은 우리의 system이 외부의 system들과 어떤 relationship을 가지는지 정적인 표현하고, activity diagram은 동적인 process를 표현함.

7. Interaction Models

(1) Use case modeling

- Modeling user interaction is important as it helps to identify user requirements
- Use cases were developed originally to support requirements elicitation and now incorporated into the UML.
- Each use case represents a discrete task that involves external interaction with a system.

- Actors in a use case may be people or other systems.(actor는 external)
- Represented diagrammatically to provide an overview of the use case and in a more detailed textual form(ex. Tabular description).

(2) Sequence Diagrams
- Modeling system-to-system interaction highlights the communication problems that may arise.
- Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- A sequence diagram shows the sequence of interactions between the actors and the objects within a system 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.

8. Structural models (static structure)
- Structural models of software display the organization(architecture) of a system in terms of the components that make up that system and their relationships.
- 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.
- You create structural models of a system when you are discussing and designing the system architecture.

(1) 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.
- Object class : a general definition of one kind of system object.
  하나의 class는 name, attribure, operation(method)로 구성되어 있음
- Association : a link between classes that indicates that there is some relationship between these classes
- When you are developing models during the early stages of the software engineering process, objects represent something in the real world.

(2) Generalization hierarchy
- If changes are proposed because of scope for generalization, then you do not have to look at all classes in the system to see if they are affected by the change.
- In object-oriented languages(ex. Java) generalization is implemented using the class inheritance mechanisms built into the language.
- The lower-level classes(subclasses) inherit the attributes and operations from their superclasses(higher-level classes). These lower-level classes then add more specific attributes and operations(is – a 관계, kind – of 관계).
- An aggregation model shows how classes that are collections are composed of other classes(consist – of 관계).

9. Behavioral Models (dynamic behavior)

- They show what happens or what is supposed to happen when a system responds to a stimulus(event) from its environment.
- these stimuli as being of Data(Some data arrives that has to be processed by the system, system에 의해 처리되는 것) and Event(Some event happens that triggers system processing, system의 trigger가 되거나 사건이 되는 것).
- Events may have associated data, although this is not always the case.

(1) **Data**-driven(processing) Modeling

- business system에서 주로 사용
- Data-driven models show the sequence of actions involved in processing(=bubble = system) input data and generating an associated output. 동그라미는 process이며, 사각형은 data store(file, system, data base)를 의미한다.
- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.
- UML에선 **activity model**을 대안으로 사용. 동그라미는 activity(d1->d2)이며, 사각형은 device와 source를 의미한다. + Sequence Diagram을 이용해서 Interaction perspective(어떤 entity=object class가 어떤 data를 주고 받는지)를 가짐

(2) **Event**-driven Modeling

- control system, embedded system(주로 real-time system)에서 주로 사용
- Event-driven modeling 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. (state1 $\overrightarrow{event}$ state2)

1) State Machine Models

- These model the behaviour of the system in response to external and internal events.
- They show the system's responses to stimuli so are often used for modelling real-time systems.
- State machine models show system states as nodes(state name과 do:activity로 구성) and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- UML의 statechart가 state machine model을 표현하기 위해서 사용됨
- state diagram은 직관적이며, system이 방대해지면, stepwise requirment에 따라 위에서부터 나눠서 그린다.

10. Model-driven Engineering(MDE)

- Approach to software development where models rather than programs are the principal outputs of the development process.
- Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

(1) 장점

- Allows systems to be considered at higher levels of abstraction
- Generating code automatically means that it is cheaper to adapt systems to new platforms.

(2) 단점
- Models for abstraction and not necessarily right for implementation.
- Savings from generating code may be outweighed by the costs of developing translators for new platforms.

11. Model Driven Architecture(MDA) MDE->MDA like Agile ->XP
- General model-driven engineering
- Model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

(1) Types of model
- A computation independent model (CIM) = domain models
 : the important domain abstractions used in a system.
- A platform independent model (PIM)
 : These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
- Platform specific models (PSM)
 : These are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

(2) Agile methods and **MDA**
- The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods.
- The notion of **extensive up-front modeling** contradicts the fundamental ideas in the agile manifesto and so suspect that few agile developers feel comfortable with model-driven engineering.
- If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required.

(3) Executable UML
- MDE는 모델에서부터 코드가 완벽히 자동적으로 변환된다는 가정이 있어야 한다. 이를 UML을 통해서 하며 가능하게 하는 UML을 Excutable UML 또는 xUML이라 부른다.

1) Executable subset of UML
- Domain models : identify the principal concerns in a system. They are defined using UML class diagrams and include objects, attributes and associations.
- Class models in which classes are defined, along with their attributes and operations.
- State models : a state diagram is associated with each class and is used to describe the life cycle of the class.

2) The dynamic behavior of the system may be specified declaratively using the object constraint language (OCL), or may be expressed using UML's action language.

chap 7

1. Software Architecture

- Architectural Design : the design process for identifying the sub-systems making up a system and the framework for sub-system control and communication
- Architectural design decisions include decisions on the application architecture, the distribution and the architectural styles to be used.
- Different architectural models such as a structural model, a control model and a decomposition model may be developed.
- Output for Architecture Design = description of the software architecture

2. Architectural Design

: An early stage of the system design process

- Represents the link between requirement engineering and design processes.
- Requirement engineering : requirment gathering → partioning → grouping → clustering
- Design processes : subsystem 식별 → 식별된 subsystem에 requirement 배정 → subsystem 정의 → subsystem 간의 인터페이스 정의
- Often carried out in parallel with some requirements specification activities.
- It involves **identifying** major system components and their **communications & control.**

(1) Advantage of explicit architecture

- Stakeholder communication : Architecture may be used as a focus of discussion by system stakeholders.
- System analysis : Means that analysis of whether the system can meet its nonfunctional requirements is possible. Performance, reliability, etc.
  A > B> C처럼 A architecture가 B, C보다 non-functional requirement를 잘 충족시키는지 판단할 수 있는 베이스라인 제공
- Large-scale reuse : The architecture may be reusable across a range of related systems with similar requirements.
  design level(= abstract level) ~ code까지 reuse 가능
- 점진적인 prototype 개발을 가능하게 한다 : Architecture가 결정되면, 분석 가능한 prototype 형태의 골격 시스템으로 개발이 가능해짐.

(2) Architecture confilcts

- Performance : Localize critical operations within a small number of subsystems and minimize communications. Use large rather than fine-grain components.
  즉, components간의 통신을 줄이기 위해 그 크기를 크게 해야 함.
- Security : Use a layered architecture, with critical assets protected in the inner layers and with high level of security validation to these.
  즉, layered architecture를 사용하여 중요한 것을 내부 layer에 보관하고 보안 인증을 높은 level에서 처리함
- Safety : Localize safety-critical features in a small number of sub-systems
  즉, 안전에 관련된 operations를 하나의 서브시스템 내부로 국한시킴

- Availability : Include redundant components and mechanisms for fault tolerance.
  즉, 여분의 components를 두고 fault tolerance mechanism을 가짐
- Maintainability : Use fine-grain, replaceable components.
  즉, 작고 독립적인 components로 설계하여 변경하기 쉽게 함

1) Trade-off

- performance & maintainability
  : using large-grain component improves performance but reduces maintainability
- availability & security
  : introducing redundant data improves availability but makes security more difficult
- safety & performance
  : localizaing safety-related features usually means more communication so degreded performance

(3) System structuring

- Concerned with decomposing the system into interacting sub-systems.
- The architectural design is expressed as a **block diagram** presenting an overview of the system structure. Block diagram is very abstract. It does not show the nature of component relationships nor the externally visible properties of the subsystems. However, useful for communication with stakeholders and for project planning.
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

3. Architectural design decisions

: A creative process so the process differs depending on the type of system being developed. However, a number of common decisions span all design processes

(1) 고려사항

- Is there a generic application architecture that can be used? generic models, reference model
- How will the system be distributed?
- What architectural styles(pattern) are appropriate? repository, client-server, layered architecture
- What approach will be used to structure the system?
- How will the system be decomposed into modules? object model or functional-oriented pipelining
- What control strategy should be used? centralized control or event-driven control
- How will the architectural design be evaluated?
- How should the architecture be documented?

(2) Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- When designing a system architecture, you have to decide what your system and broader application classes have in common, and decide how much knowledge from these application architectures you can reuse.

(3) Architecture styles = Pattern of system organization
- The architectural model of a system may conform to a generic architectural model or style(ex. client-server, layered architecture). An awareness of these styles can simplify the problem of defining system architectures.
- However, most large systems are heterogeneous and do not follow a single architectural style.

(4) Architectural models
: used to document an aritectural design
- Static structural model : the major system components
- Dynamic process model : the process structure of the system
- Interface model : defines sub-system interfaces.
- Relationships model (ex. data-flow model) : sub-system relationships.
- Distribution model : how sub-systems are distributed across computers.

(5) Architectural Description Language(ADL)
: Block diagram을 보충해줌

1) C2(connector와 compoenet)
- 장점 : 공통된 특성을 가진 component군과 또 다른 특성을 가진 component군의 상호작용을 알기 쉽게 표현함
- 단점 : 같은 layer에 존재하는 component들의 상호 관계를 파악하기 어려움
- layered architecture는 아님

2) Weave
- 장점 : port를 통해서 연결하여 어떤 component의 관계도 표현 가능함
- 단점 : component들의 관계도 함께 표현하다보니 복잡해짐

3. System organization (Structural level에 따라 system → sub-system → sub-system으로 나누는 과정)
: Reflects the basic strategy that is used to structure a system.

(1) A shared data repository style
: Sub-systems must exchange data.
① indirected communication : Shared data is held in a central database or repository and may be accessed by all sub-systems
② directed communication : Each sub-system maintains its own database and passes data explicitly to other sub-systems.

1) 장점
- Efficient way to share large amounts of data
- Sub-systems that produce data need not be concerned with how that data is used other sub-systems
- Activities such as backup, security, access control and recovery from error are centralized; Repository Manager subsystem들은 각각의 repository를 가질 필요 없이 중앙에서만 관리

- The sharing model is visible through the repository schema. Integrate new tools with agreed data model.

2) 단점
- Sub-systems must agree on the repository data model. Inevitably a compromise
- Data evolution is difficult and expensive
- Different sub-systems may have different requirements for security, recovery and backup policies
- Difficult to distribute the repository over a number of machines.
: system의 규모가 커지면 repository의 개수가 증가하고 이들 간의 동기화(synchronize) 문제인 data redundancy, inconsistency가 발생할 수 있음

(2) Client-server Model : A shared services and servers style
- Distributed system model which shows how data and processing is distributed across a range of components. (ex. web service & cloud computing)
- Set of stand-alone Servers : provide specific services(ex. printing, data management, etc)
- Set of Clients : call on these services
- Network which allows clients to access servers.

1) 장점
- Distribution of data and processing is straightforward
- Makes effective use of networked systems.
- May require cheaper hardware(하나당 데이터가 작으므로).
- Easy to add new servers or upgrade existing servers.

2) 단점
- No shared data model so sub-systems use different data organization. Data interchange may be inefficient. → standard or transformer 사용
- Redundant management in each server.
- No central register of names and services. it may be hard to find out what servers and services are available. → directory service(white/yellow page : server의 이름, 제공하는 서비스를 이용하여 검색) 사용

(3) Layered model(Abstract machine model)
: Used to model the interfacing of sub-systems.
- Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.

1) 장점
- Supports the incremental development of sub-systems in different layers.
- When a layer interface changes, only the adjacent layer is affected.

2) 단점
- Often artificial and difficult to structure systems in this way.
- Performance can be a problem because of the multiple levels of command interpretation.

4. Modular decomposition styles (sub-system → module로 나누는 과정)

- Styles of decomposing sub-systems into modules after an overall system organization has been chosen.
- No rigid distinction between system organization and modular decomposition.
- If possible, decisions about concurrency should be delayed until modules are implemented.
- Sub-systems : a system in its own right whose operation is independent of the services provided by other sub-systems.
- Modules : a system component that provides services to other components but would not normally be considered as a separate system.

(1) Object Models

- Structure the system into a set of loosely coupled objects with well-defined interfaces.
  = Strong cohesion & weak coupling을 충족하는 단위 = object
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations.
- When implemented, objects are created from these classes and some control model used to coordinate object operations.

1) 장점

- Objects are loosely coupled so their implementation can be modified without affecting other objects.
- The objects may reflect real-world entities.
- OO implementation languages are widely used.
- maintainence가 좋아서 long life time system에서 좋음

2) 단점

- Object interface changes may cause problems and complex entities may be hard to represent as objects.

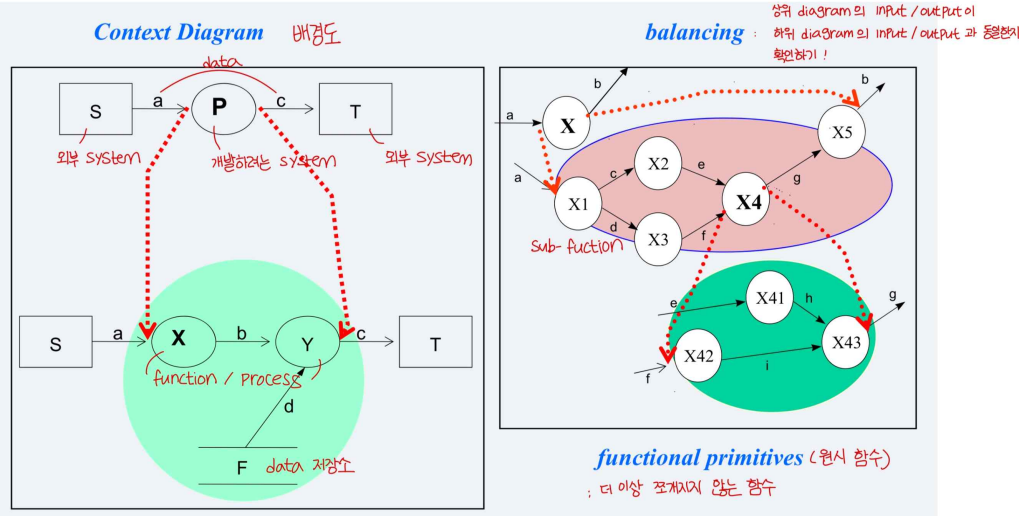
→ interface에 대한 정보 공유 필요

(2) Function-oriented Pipelining

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems(OOM이 적합). Business system엔 적합함

1) 장점

- Supports transformation reuse.
- Intuitive organization for stakeholder communication.
- Easy to add new transformations.
- Relatively simple to implement as either a concurrent or sequential system.



2) 단점

- Requires a common format for data transfer along the pipeline
- Difficult to support event-based interaction.

→ DFD(Data Flow Diagram)을 지원하는 editor로 event 내용을 보충하려 하지만 어려움

5. Control Styles

: Concerned with the control flow between sub-systems. Distinct from the system decomposition model.

(1) Centralized control

: One sub-system has overall responsibility for control and starts and stops other sub-systems.

- A control sub-system takes responsibility for managing the execution of other sub-systems.

1) Call-return model

- Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards.
- Applicable to sequential systems.

2) Manager model(= Soft Real-time system control)

- One system component controls the stopping, starting and coordination of other system processes.
- Applicable to concurrent systems and also implement in sequential systems as a case statement.

(2) Event-based control

: Each sub-system can respond to externally generated events from other sub-systems or the system's environment. (하나 아님)

- Driven by externally generated events where the timing of the event is outside the control of the sub-systems which process the event.
- Broadcast model, Interrupt-driven model, spreadsheets, production system 등이 있다

1) Selective Broadcast Model

- Broadcast Model : An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so.
  BUT, sub-system들이 event를 계속 감시해야 하므로 process가 낭비되어 overhead가 발생함
- Selective Broadcast Model : Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event. Control policy is not embedded in the event and message handler.

→ 단점

- sub-systems don't know if or when an event will be handled. + 할지 안 할지도 모름 따라서, real time system이나 time-contraint가 강한 system은 구현하지 못함
- 각각의 sub-system의 event에 응답이 다를 때, 어떻게 처리해야 하는지 고민을 해야함.
- 따라서, 사용되는 곳이 제한적임

2) Interrupt-driven model

: Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing.

- There are known interrupt types with a handler defined for each type. (사전에 처리 가능한 Interrupt type과 handler가 정의되어 있음)
- Each interrupt type is associated with a memory location
- A hardware switch(= interrupt vector : handler의 address가 담김) causes transfer to its handler.
- Allows fast response but complex to program and difficult to validate.

6. Reference Architecture

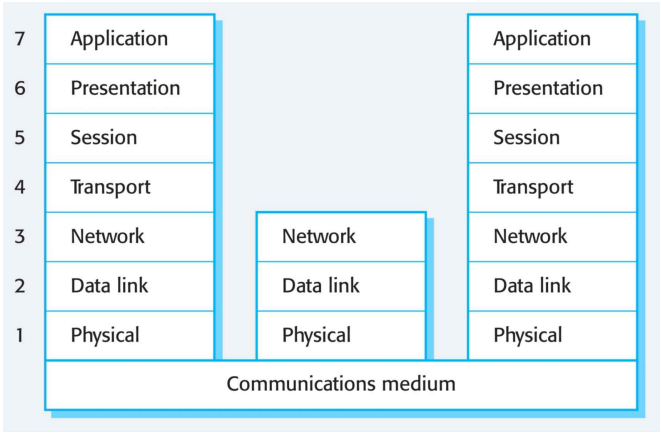
- Architectural models may be specific to some application domain.
- Reference architectures may be used to communicate domain specific architectures and to assess and compare architectural designs.
- Two types of domain-specific model

① Generic models : abstractions from a number of real systems and which encapsulate the principal characteristics of these systems. (bottom-up models : domin들의 공통적인 속성을 모아서 architecture 만듦)

② Reference models : more abstract, idealized model. Provide a means of information about that class of system and of comparing different architectures. (top-down models : 반드시 따라야함)

(1) Reference model

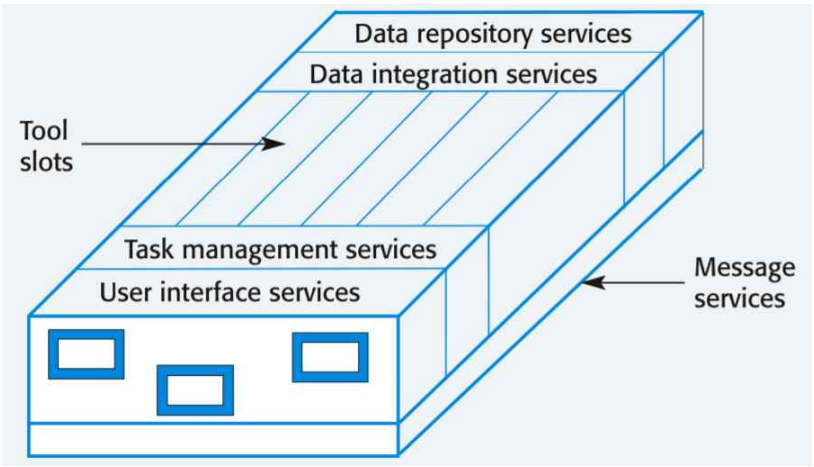
- Reference models are derived from a study of the application domain rather than from existing systems.
- May be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated.
- OSI model is a layered model for communication systems.



OSI reference model

(2) Case reference model

- Data repository services : Storage and management of data items.
- Data integration services : Managing groups of entities.
- Task management services : Definition and enaction of process models.
- Messaging services : Tool-tool and tool-environment communication.
- User interface services : User interface development.



ECMA reference model

Chap 9 Design and Implementation

1. Design and Implementation

: the stage in the software engineering process at which an executable software system is developed.

- Software design and implementation activities are invariably inter-leaved.
- The level of detail in the design depends on the type of system and wheter you are using plan-driven or agile approach.
- Software design : A creative activity in which you identify software components and their relationships, based on a customer's requirements.
- Implementation : The process of realizing the design as a program.

2. Build or Buy

- In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
- When you develop an application, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

3. An object-oriented design process

- Structured object-oriented design processes involve developing a number of different system models.
- They require a lot of effort for development and maintenance of these models.
- For large systems developed by different groups design models are an important communication mechanism. However, for small systems(+short lifetime +computational system), this may not be cost-effective.

(1) Process Stages

: There are a variety of object-oriented design processes that depend on the organization using the process

① Define the context and modes of use of the system
② Design the system architecture
③ Identify the principal system objects
④ Develop design models
⑤ Specify object interfaces

1) System context and interactions context diagram + use case diagram

- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- Understanding of the context lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

- System context model(structural model) : demonstrates the other systems in the environment of the system being developed.
- Interaction model(dynamic model) : show how the system interacts with its environment as it is used.

2) Architectural design package diagram
: identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.

3) Object class identification 여기서부터 객체 지향만의 방법 Class diagram
- Identifying object classes : difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

• Approaches to Object class identification
- Grammatical analysis : based on a natural language description of the system. Objects and attributes are nouns; operations or services are verbs.
- Base the identification on tangible things in the application domain, roles, events, interactions, locations and organizational units.
- Behavioural approach : what participates in what behaviour.
- scenario-based analysis : The objects, attributes and methods in each scenario are identified.
: OMT 때 사용

4) Design models
: show the objects and object classes and relationships between these entities.
- **Static** models : the static structure of the system in terms of object classes and relationships.
ex. Subsystem models, use-case models, class models, generalization models, aggregation models
- **Dynamic** models : the dynamic interactions between objects. 관계 속에서 어떤 data를 주고 받는지
ex. sequence models, state machine models

• Subsystem models package diagram
: logical groupings of objects into coherent subsystems
- In the UML, these are shown using packages(encapsulation construct).
- This is a logical model. The actual organization of objects in the system may be different.

• Sequence models sequence diagram
: the sequence of object interactions
- Objects are arranged horizontally across the top
- Time is represented vertically so models are read top to bottom
- Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction
- A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

• State machine models state diagram 모든 loop마다 testing을 해야함
: show how objects respond to different service requests and the state transitions triggered by these requests. method call에 대해서 어떤 state에 transit을 일으키는지
- You don't usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

• use-case models : 단계마다 use case 계속 작성
• aggregation models : part-of, consist of 관계 기술
• generalization models : is-a, kind of 관계 기술

5) Interface specification
- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Designers should avoid details of the data representation in an interface design, as attributes are not defined in an interface specification. But include operations to access and update data.
- Objects may have several interfaces which are viewpoints on the methods that they provide.
- The UML uses class diagrams for interface specification.

4. Design patterns 23개의 design pattern이 있음(adapter, factory, singleton, strategy, composite, facade...)
: a way of reusing abstract knowledge about a problem and its solution
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism

subsystem들의 공통의 문으로 subsystem들의 정보를 가지고 있음. 외부에서 정보가 오면 어떤 subsystem에게 이를 전달하지 알고 있음. 이때 subsystem들은 black box임

(1) Pattern elements
- name : a meaningful pattern identifier
- problem description : a description of the problem area that explains when the pattern may be applied
- solution description : not a concrete design but a template for a desgin solution that can be instantiated in different ways
- consequences : the results and trade-offs of applying the pattern.

(2) Observer pattern
- name : observer = publish – subscrible
- Description : Separates the display of object state from the object itself.
- Problem description : Used when multiple displays of state are needed.
- Solution description : See slide with UML description.
  - instantation : 클래스 내의 객체에 대해 특정한 변형을 정의하고 이름을 붙인 다음, 그것을 물리적인 어떤 장소에 위치시키는 등의 작업을 통해 instance를 만드는 것
  - abstract class : 동작의 기본이 되는 object를 instantation을 할 수 없음
  - concrete class : instantation 가능
- Consequences : Optimizations to enhance display performance are impractical.

(3) Design problems

: To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.

(4) Implementation issues

: Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts

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.

• Reuse levels

① The abstraction level : At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
② The object level : At this level, you directly reuse objects from a library rather than writing the code yourself.
③ The component level : Components are collections of objects and object classes that you reuse in application systems.
④ The system level : At this level, you reuse entire application systems.

• Reuse costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

2) Configuration management

: the name given to the general process of managing a changing software system.
ex. github : open-source configuration management

- Aim : to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

• Configuration management activities

① Version management : to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
② System integration : to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.

③ Problem tracking : to allow users to report bugs and other problems, and to allow all developers to see who(bug report) is working on these problems and when they are fixed.

3) Host-target development

: Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on the host system and execute it on the target system.

- Platform : More than just hardware. It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform. these platforms may have different architectures.
- You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.

• Development platform tools

- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- Project support tools that help you organize the code for different development projects.

• IDE (Integrated Development Environments)

- Software development tools are often grouped to create an IDE.
- An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

5. Component / System deployment factors

- If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.

6. Open Source development

: RE → OSS(open source search) → Design → Implement → Testing

- Open source development is 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
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

1) Open Source systems

- The best-known open source product is the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- Other important open source products are Java, the Apache web server and the mySQL database management system.

2) Open source issues

- Should the product that is being developed make use of open source components? YES
- Should an open source approach be used for the software development? YES

3) Open Source business

- More and more product companies are using an open source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

4) Open Source licensing

- A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
- Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
- Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
- Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

5) License models

- General Public License (GPL) : This is a so-called 'reciprocal' license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- Lesser General Public License (LGPL) : a variant of the GPL license where you can write

components that link to open source code without having to publish the source of these components.
- Berkley Standard Distribution (BSD) License : This is a nonreciprocal license, which means you are not obliged to re publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.
- Black Duck : open source에 대한 법적 문제가 없는지 확인해주는 tool

6) License management
- Establish a system for maintaining information about opensource components that are downloaded and used.
- Be aware of the different types of licenses and understand how a component is licensed before it is used.
- Be aware of evolution pathways for components.
- Educate people about open source.
- Have auditing systems in place.
- Participate in the open source community

chap 10. Object-oriented Analysis and Design

1. Why OOAD?

① maintainence 비용 감소

② cohesion 증가와 coupling 감소로 인한 quality 상승

- cohesion : 한 component를 설명하는 문서들이 얼마나 관련성이 높은가
- coupling : 한 문서를 수정했을 때 다른 문서에 영향을 얼마나 미치는가

③ Time-to-market(TTM) 감소로 reuse 상승

2. UML vs OMT

- UML : 주로 객체 지향 소프트웨어에 사용되는 대중화되고 표준화 된 모델링 언어 (computation)
- OMT : 객체 모델링 기법의 약자 (interaction, 초기시간 필요)

structure diagram	class diagrm		시스템을 구성하는 클래스들의 관계 표현
	object diagram		객체 정보
	composite structure diagram		복합 구조의 클래스와 컴포넌트 내부 구조 표현
	deployment diagram		HW, SW, 네트워크를 포함한 실행 시스템의 물리 구조 표현
	component diagram		컴포넌트 구조 사이의 관계 표현
	package diagram		클래스나 유스케이스 등을 포함한 여러 모델 요소들을 그룹화해서 패키를 구성 & 관계 표현
behavior diagram	activity diagram		업무 처리 과정이나 연산이 수행되는 과정
	state machine diagram		객체의 생명주기 표현
	use case diagram		사용자 관점에서 시스템 행위 표현
	interaction diagram	sequence diagram	시간 흐름에 따른 객체 사이의 상호작용
		interaction overview diagram	여러 상호작용 다이어그램 사이의 제어 흐름
		communication diagram	객체 사이의 관계를 중심으로 상호작용 표현
		timing diagram	객체 상태 변화와 시간 제약을 명시적으로 표현

UML

3. OMT 과정

① Object Modeling 정적 구조

- system 내에서 필요한 object 추출 & object calss 간의 relationship 분석
- objects(classes) identification : problem description → extraction of classes from problem description → Relationship among classes → attributes of classes

② Dynamic Modeling 동적 구조

: problem description → Senario → Sequence diagram & Event trace diagram
→ STD(State Transition Diagram) for each object identified in ①

- 시간과 변화하는 시스템의 동적인 측면에 초점
- 각각의 객체가 event에 따라서 어떤 처리로 넘어가고 서로 다른 state로 옮겨가느냐에 대한 기술

- event trace diagram : event의 개념 및 event을 바꿔주는 object 표현
- state transition diagram : event와 state를 관련지어 표현

③ Functional Modeling
: DFD(Data Flow Diagram) for each Activity appeared in ②
만약, UML로 그리면 activity diagram으로 대체 가능
- 각각의 object 사이에 어떤 data가 흐르고, object가 어떤 process를 수행할까를 기술

Chap 11 Project Management

1. Software project management

- Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organizations developing and procuring the software.
- Project management is needed because software development is always subject to budget and schedule constraints that are set by the organization developing the software.

2. Software management distinctions

- The product is intangible 문서만이 tangible
- The product is uniquely flexible.
- Software engineering is not recognized as an engineering discipline with the sane status as mechanical, electrical engineering, etc.
- The software development process is not standardized.
- Many software projects are 'one-off' projects.

3. Management activites

- Proposal writing.
- Project planning and scheduling.
- Project costing.
- Project monitoring and reviews.
- Personnel selection and evaluation.
- Report writing and presentations.

: These activities are not peculiar to software management. (Management Commonalities)

(1) Project Staffing

1) May not be possible to appoint the ideal people to work on a project

- Project budget may not allow for the use of highly-paid staff
- Staff with the appropriate experience may not be available
- An organization may wish to develop employee skills on a software project.

2) Managers have to work within these constraints when there are shortages of trained staff.

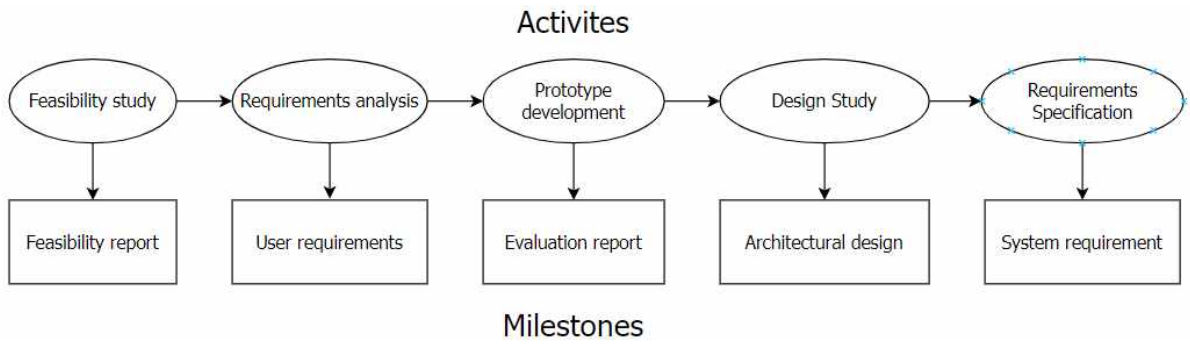
3) Managers have to diverse roles but their most significant acitivities are planning, estimating and scheduling. planing and estimating are iterative processes which continue throughout the course of a project

4. Project Planning

- The most time-consuming project management activity.
- Continuous activity from initial concept through to system delivery. plans must be regularly revised as new information becomes available.
- Various different types of plan may be developed to support the main software project plan that is concerned with schedule and budget.

(1) Activity organization

- Activity in a project : to produce tangible outputs for management to judge progress
- Milestones : the end-point or check point(predicitable state) of a process activity
- Deliverables : project results delivered to customers
- The waterfall process allows for the straightforward definition of progress milestones.



(2) Type of project plan

Quality plan	describes the quality procedures and standards that will be used in a project.
Validation plan	describes the approach, resources(human, hardware, software) and schedule used for system validation.
Configuration management plan	describes the configuration management procedures and structures to be used
Maintenance plan	predicts the maintenance requirements of the system, mainenance costs and effort(man month) required
Staff development plan	describes how the skills and experience of the project team members will be developed

(3) Project planning process

Establish the project constraints(delivery data, staff avaliable, budget, ...)
Make initial assessments of the project parameters
(project structure, size, distribution of functions, ...)
Define project milestones(이정표) and deliverables(고객에 전할 것)
while project has not been completed or cancelled loop
Draw up project schedule
Initiate activities according to schedule

```
        Wait ( for a while )
        Review project progress
        Revise estimates of project parameters
        Update the project schedule
        Re-negotiate project constraints and deliverables
        if ( problems arise ) then
                Initiate technical review and possible revision
        end if
end loop
```

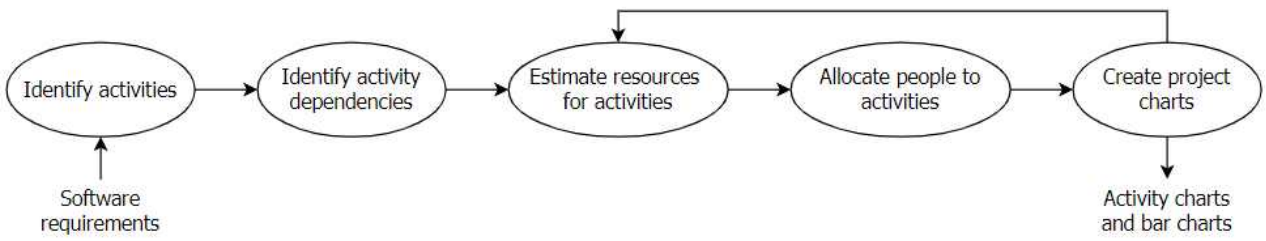
(4) Project plan structure

Introduction	objectives of the project, constraints, ...
Project organization	development team(people, roles)
Risk analysis	possible risks, likelihood, reduction strategies
HW&SW resource requirements	H/W, S/W for development, if but, its price and delivery time
Work breakdown	breakdown of the project into activities and milestones, deliverables
Project schedule	dependencies between activites, estimate time for each milestone and the allocation of people to activities
Monitoring and reporting mechanism	management reports which should be produced, when these should be produced and the project monitoring mechanism used

5. Project Scheduling

- Split project into tasks and estimate time and resources required to complete each task.
- Organize tasks concurrently to make optimal use of workforce.
- Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- Dependent on project managers intuition and experience.

(1) Project scheduling process



(2) Scheduling problems
- Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- Productivity is not proportional to the number of people working on a task.
- Brook's law : Adding people to a late project makes it later
- The unexpected always happens. Always allow contingency in planning.
- project scheduling involves preparing various graphical representations showing project activities, their durations and staffing.

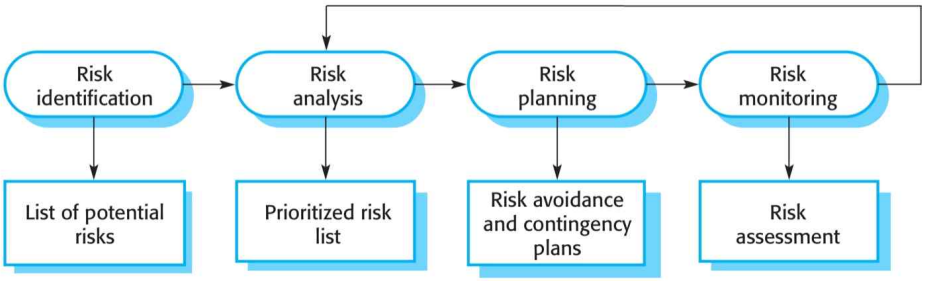
(3) Bar charts and activity networks
- Graphical notations used to illustrate the project schedule.
- Show project breakdown into tasks. Tasks should not be too small. They should take about a week or two.
- Activity charts : task dependencies and the critical path(= longest path : 프로젝트를 위한 최소 시간).
- Bar charts : schedule against calendar time.

6. Risk Management
- Risk management is concerned with identifying risks and drawing up plans to minimize their effect on a project.
- A risk is a probability that some adverse circumstance will occur
- **Project risks** : schedule or resources
- **Product risks** : the quality or performance of the software being developed
- **Business risks** : the organization developing or procuring the software.

(1) Software risks

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of company management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
Software tool underperformance	Product	Software tools that support the project do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

(2) The risk management process



1) Risk identification

: identify project, product and business risks

Risk type	Possible risks
Estimation	1. The time required to develop the software is underestimated. 2. The rate of defect repair is underestimated. 3. The size of the software is underestimated.
Organizational	4. The organization is restructured so that different management are responsible for the project. 5. Organizational financial problems force reductions in the project budget.
People	6. It is impossible to recruit staff with the skills required. 7. Key staff are ill and unavailable at critical times. 8. Required training for staff is not available.
Requirements	9. Changes to requirements that require major design rework are proposed. 10. Customers fail to understand the impact of requirements changes.
Technology	11. The database used in the system cannot process as many transactions per second as expected. 12. Faults in reusable software components have to be repaired before these components are reused.
Tools	13. The code generated by software code generation tools is inefficient. 14. Software tools cannot work together in an integrated way.

: Derive from the management estimates of the system characteristics and the resources required to build the system

: Derive from the organizational environment where the system is being developed

: Associated with the people in the development team

: Derive from changes to the customer requirements and the process of managing the requirements change

: Derive from the S/W and H/W technology which are being used as part of the system being developed

: Derive from the CASE tools and other support SW used to develop the system

2) Risk analysis

: assess the likelihood(probability) and consequences(seriousness) of these risks

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (5).	Low	Catastrophic
It is impossible to recruit staff with the skills required (6).	High	Catastrophic
Key staff are ill at critical times in the project (7).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused (12).	Moderate	Serious
Changes to requirements that require major design rework are proposed (9).	Moderate	Serious
The organization is restructured so that different managements are responsible for the project (4).	High	Serious
The database used in the system cannot process as many transactions per second as expected (11).	Moderate	Serious

The time required to develop the software is underestimated (1).	High	Serious
Software tools cannot be integrated (14).	High	Tolerable
Customers fail to understand the impact of requirements changes (10).	Moderate	Tolerable
Required training for staff is not available (8).	Moderate	Tolerable
The rate of defect repair is underestimated (2).	Moderate	Tolerable
The size of the software is underestimated (3).	High	Tolerable
Code generated by code generation tools is inefficient (13).	Moderate	Insignificant

3) Risk planning

: Draw up plans to avoid or minimise the effects of the risk

- Avoidance strategies : the probability that the risk will arise is reduced
- Minimization strategies : the impact of the risk on the project or product will be reduced
- Contingency plans : if the risk arises, contingency plans are plans to deal with that risk

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of automated code generation.

4) Risk monitoring

- Assess each identified risks regularly to decide whether or not it is becoming less or more probable.
- Also assess whether the effects of the risk have changed.
- Each key risk should be discussed at management progress meetings.

Risk type	Potential indicators
Estimation	Failure to meet agreed schedule; failure to clear reported defects.
Organizational	Organizational gossip; lack of action by senior management.
People	Poor staff morale; poor relationships among team members; high staff turnover.
Requirements	Many requirements change requests; customer complaints.
Technology	Late delivery of hardware or support software; many reported technology problems.
Tools	Reluctance by team members to use tools; complaints about software tools; requests for faster computers/more memory, and so on.

Chap 13. Software Testing

1. Program testing

- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- When you test software, you execute a program using artificial data.
- You check the results of the test run for errors, anomalies or information about the program's **non-functional attributes.**
- Can reveal the presence of errors NOT their absence.
- Testing is part of a more general verification and validation process.

(1) Validation Testing

: To demonstrate to the developer and the customer that the software **meets its requirements.**

- For custom software, this means that there should be at least one test for every requirement in the requirements document.
- For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- A successful test shows that the system operates as intended.

(2) Defect Testing

: To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.

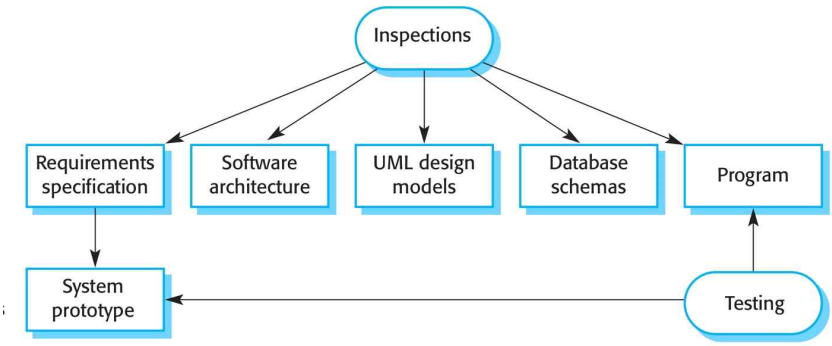
- rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.
- The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.
- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Testing은 defeat의 존재여부확인, Debugging은 defeat가 어디에 있고 그걸 찾아서 고치는 것

2. Inspections and testing 상호보완적

: Inspections and testing are complementary and not opposing verification techniques.

- Both should be used during the V & V process.



(1) V & V confidence

- **Verification** : building the product right → The software should conform to its **specification**
- **Validation** : building the right product → The software should do **what the user really requires.**
- Aim of V & V : establish confidence that the system is 'fit for purpose'.
- Depends on system's purpose, user expectations and marketing environment

① Software purpose : The level of confidence depends on how critical the software is to an organization.

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.

(2) Software inspections (static verification)

: Concerned with analysis of the static system representation to discover problems.

- May be supplement by tool-based document and code analysis.
- These involve people examining the source representation with the aim of discovering anomalies and defects.
- planning -> overview -> individual preparation -> inspection meeting -> rework

1) 장점

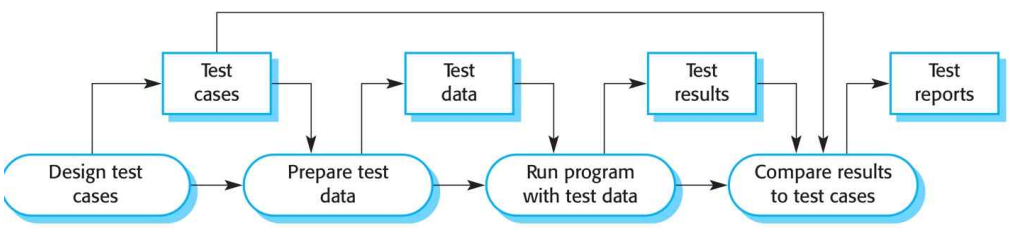
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.
- During testing, errors can mask (hide) other errors. Because inspection is a **static** process, you don't have to be concerned with interactions between errors.
- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader **quality** attributes of a program, such as **compliance** with standards, **portability** and **maintainability.**
- Inspections can check conformance with a **specification**

2) 단점

- Inspections can not check conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

(3) Software testing(dynamic V & V)

: Concerned with exercising and observing product behaviour. The system is executed with test data and its operational behaviour is observed.



3. Stage of Testing

: Development testing → Release testing → User testing

- Testing can only show the presence of errors in a program. it cannot demonstrate that there are no remaining faults.
- When testing software, you should try to break the software by using experience and guidelines to choose types of test case that have been effective in discovering defeats in other systems
- Whenever possible, you should write **automated tests**. the tests are embedded in a program that can be run **every** time a change is made to a system

4. Development testing

: where the system is tested during development to discover bugs and defects

- all testing activities that are carried out by the team developing the system

4.1. Unit testing defeat testing

- where individual program(component) units or object classes are tested in isolation.
- focus on **testing the functionality of objects or methods.**
- The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
- If there are defects in the component, these should be revealed by test cases. Test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

(1) Unit

- Individual functions or methods within an object
- Object classes with several attributes and methods
- Composite components with defined interfaces used to access their functionality.

(2) Object class testing

- Complete test coverage of a class involves

① Testing all operations associated with an object → 모든 loop를 한번씩은 test해야함
② Setting and interrogating(값 호출 및 반환) **all** object attributes
③ Exercising the object in **all** possible states.

- **Inheritance** makes it more difficult to design object class tests as the information to be tested is not localized.
- Using a **state model**, identify sequences of state transitions to be tested and the event sequences to cause these transitions

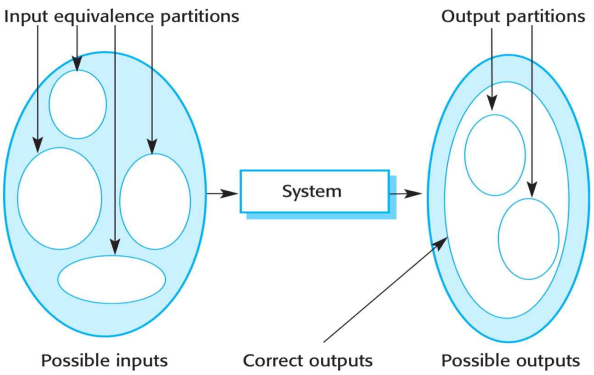
(3) Automated testing

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.

- Unit testing frameworks provide **generic** test classes that you **extend** to create specific test cases. They can then run all of the tests that you have implemented and report, often through some **GUI,** on the success or otherwise of the tests.

1) Automated test components
- Setup part : initialize the system with the test case, namely the inputs and expected outputs.
- Call part : call the object or method to be tested.
- Assertion part : compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

(4) Testing strategies
1) Partition testing
: identify groups of inputs(=datas) that have common characteristics and should be processed in the same way.
- Input data and output results of a program fall into **different classes** that **have common characteristics**, such as positive, negative numbers and menu selections.
- Each of these **classes** is an **equivalence partition** or domain where the **program behaves in an equivalent way** for each class member.
- Test cases should be chosen **from each partition.**



• Test case design in parition testing
① identifying all paritions for a system or components
- identify partition **by using the specification or user documentation**
- input equivalence partitions : all of the set members should be processed in an equivalent way
- output equivalence partitions : program outputs that have common characteristics

② choose test cases from each of theses partitions
- choose test cases on the boundaries of the partitions plus cases close to the mid-point of the partition

2) Guideline-based testing
: use testing guidelines to choose test cases. These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

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

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

4.2 Component testing = Interface testing defeat testing

: Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Software components are composite components that are made up of several interacting objects.
- You access the **functionality** of these objects through the defined component interface.
- focus on showing that the **component interface behaves according to its specification.**
- You can assume that unit tests on the individual objects within the component have been completed.

(1) Interface Type

1) Direct
- Parameter interfaces : Data passed from one component to another.
  ex) Methods in an object.
- Procedural interfaces : Subsystem encapsulates a set of procedures to be called by other subsystems.
  ex) Object and reusable components have this form of interface.
- Message passing interfaces : Sub-systems request services from other sub-systems.
  ex) client-server system

2) Indriect
- Shared memory interfaces : Block of memory is shared between procedures or functions.
  ex) Embedded systems with sensors-processors

(2) Interface Errors
- Interface misuse : A calling component calls another component and makes an error in its use of its interface (ex. parameters in the wrong order)
- Interface misunderstanding : A calling component embeds assumptions about the behaviour of the called component which are incorrect.
- Timing errors : The called and the calling component operate at different speeds and out-of-date information is accessed.

(3) Interface testing guidelines
- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.

- Design tests which cause the component to fail.
- Use **stress test**ing in message passing systems.
- In shared memory systems, vary the order in which components are activated

4.3 System testing

: Integrating components to create a version of the system and then testing the integrated system.

- The focus in system testing is testing the **interactions between components.**
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing tests the **emergent behavior** of a system (nonfunctional).

(1) system and component testing

- During system testing, reusable components that have been separately developed and **off-the-shelf systems may be integrated with newly developed components**. The complete system is then tested.
- Components developed by different team members or sub-teams may be integrated at this stage. System testing is a **collective** rather than an individual process.
- In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

(2) Use-case testing

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- **Each use case usually involves several system components** so testing the use case forces these interactions to occur.
- The **sequence diagrams associated with the use case documents** the components and interactions that are being tested.

(3) Testing policies

: Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.

1) Examples of testing policies

- All system functions that are accessed through menus should be tested.
- Combinations of functions that are accessed through the same menu must be tested.
- Where user input is provided, all functions must be tested with both correct and incorrect input.

5. Test-driven development (TDD)

: Approach to program development in which you **inter-leave** testing and code development

- **Tests are written before code** and 'passing' the tests is the critical driver of development.
- 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.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also

be used in plan-driven development processes.

(1) TDD process activites

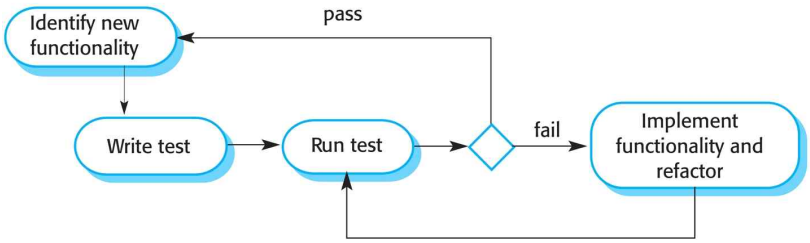
① Start by identifying the increment of **functionality** that is required. This should normally be small and implementable in a few lines of code.

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.



(2) Benefits of TDD

- Code coverage : Every code segment that you write has at least one associated test so all code written has at least one test.
- Regression testing : A regression test suite is developed incrementally as a program is developed.
- Simplified debugging : When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- System documentation : The tests themselves are a form of documentation that describe what the code should be doing.

1) Regression testing

- testing the system to check that changes have not 'broken' previously working code.
- In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program.
- Tests must run 'successfully' before the change is committed.

6. Release testing validation testing

: where a separate testing team(= outside of the development team) test a complete version of the system before it is released to user.

1) Goal of the release testing process

: to convince the supplier of the system that it is good enough for use.

- Show that the system delivers its specified **functionality, performance and dependability,** and that it does not fail during normal use.
- **Black-box testing** process where tests are only derived from the system **specification**(고객의 요구).
  ↔ White-bos testing : defeat 검사 ex. unit testing, component testing

2) Release testing and system testing

- Release testing is a form of system testing. But They have some Important differences:

① A **separate** team that has not been involved in the system development, should be responsible for release testing.

② System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system **meets its requirements and is good enough for external use(validation testing).**

(1) Requirements based testing

: examining each requirment and developing a test or tests for it.

(2) Features tested by scenario

: inventing a typical usage scenario and using this to derive test cases

(3) Performance testing

- Part of release testing may involve testing the **emergent properties** of a system, such as **performance and reliability.**
- Tests should reflect the profile of use of the system.
- Performance tests : planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing : performance testing where the system is deliberately overloaded to test its failure behavior

7. User(=Cutomer) testing

: Users or potential users of a system test the system in their own environment.

- User testing is essential, even when comprehensive system and release testing have been carried out.
- The reason for this is that influences from the user's working environment have a major effect on the **reliability, performance, usability and robustness of a system.** These cannot be replicated in a testing environment.

(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

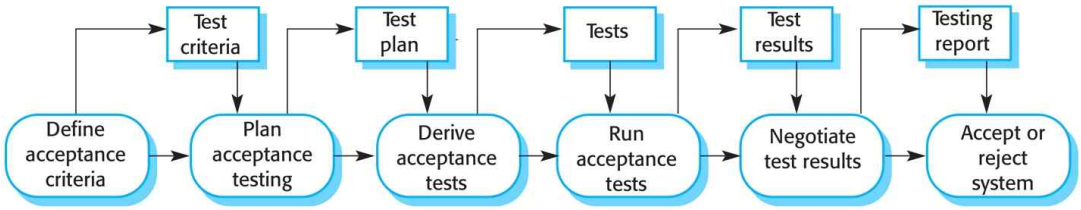
: Customers test a system to **decide** whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

1) Agile methods and acceptance testing

- In agile methods, the user/customer is part of the development team and is responsible for

making decisions on the acceptability of the system.

- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- **There is no separate acceptance testing process.**
- Main problem here is whether or not the embedded user is 'typical' and can **represent** the interests of all system stakeholders.



1. Once interactions between the system and its environment have been understood, you use this information for designing the system architecture. You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client - server model. The layered model is a good architecture to increase security, and for this, a strategy of placing the most important assets at the top of the hierarchy is appropriate.

답: no, top → inner

2. There are different design models, such as subsystem models, sequence models, state machine models, use-case models, aggregation models, generalization models, etc. Of these, sequence model, state machine model, and use-case model are dynamic models.

답: no, use-case model : static model

3. The following are some of the characteristics of design pattern. Please choose all that is true.

1) It should be sufficiently concrete to be reused in different settings

2) A pattern is a description of the problem and the essence of its solution

3) A design pattern is a way of reusing abstract knowledge about a problem and its solution

4) Design pattern elements contain name, problem description, solution description, consequence, and cost and duration for applying

5) To use patterns in your design, you need to be aware of the patterns you can apply to the design problems you are facing

답: 2), 3), 5)

1) concrete → abstract

4) cost and duration for applying은 포함되지 않음

4. The following are related with open source software. Choose all that is true.

1) A fundamental principle is that source code should be freely available, so people can do as they wish with that code

2) Its roots are in the Free Software Foundation

3) Business model is not reliant on selling a software product but on selling support for that product

4) Once a software is published as an open source, the developer of the code cannot own the code anymore.

5) The most reciprocal license is the GPL

답: 2), 3), 5)

1) , this does not mean that anyone can do as they wish with that code

4) the developer of the code still owns the code.

5. 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. The former is called validation testing, the latter is defect testing. Testing can reveal the presence of errors and also their absence.

답: no, error의 absence는 증명할 수 없음

6. With validation testing, you expect the system to perform correctly using a given set of test cases that reflect the system's expected use. A successful test shows that the system operates as intended. For defect testing, on the other hand, 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. A successful test is a test that makes the system perform correctly and so exposes a defect in the system.
답: no. correctly → incorrectly

7. Select all of the descriptions that are appropriate for the repository model.
1) Inefficient way to share large amounts of data
2) Sub-systems that produce data need to concern how that data is used other sub-systems
3) Changes made by one component can be propagated to all components
4) Activities such as backup, security, access control and recovery from error are distributed
5) Data evolution is difficult and expensive
6) The repository is a single point of failure so problems in the repository affect the whole system
7) Subsystems do not interact directly, only through the repository.

답: 3), 5), 6)
1) inefficient → efficient
2) need to concern → need not to concern
4) distributed → centralized
7) directed communication과 indirected communcation 방법 두 가지가 있음

8. Software inspections involve people examining the source representation with the aim of discovering anomalies and defects. The inspections not require execution of a system so may be used before implementation. They may be applied to any representation of the system (requirements, design, configuration data, test data, etc). They have been shown to be an inadequate technique for discovering program errors.
답: no, inadequate → adequate

9. The followings are descriptions of the MDE and MDA. Choose all that are correct.
1) MDE is software development methodology 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) Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer must be concerned with programming language details or the specifics of execution platforms
4) Savings from generating code may not be outweighed by the costs of developing translators for new platforms
5) CIM models the important domain abstractions used in a system
6) PIM models the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events

7) PSM is transformations of the platform-dependent model with a separate PSM for each application platform.
답: 1), 2), 3), 4), 5), 6)
7) platform-dependent → platform-independent

10. Unit testing is the process of testing individual units in isolation. Units may be: individual functions or methods within an object, object classes with several attributes and methods, composite components with defined interfaces used to access their functionality. It is a validation testing process.
답: no, validation → defect

11. For partition testing, input data and output results of a program fall into different classes that have common characteristics. Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member. Test cases should be chosen from the groups of each partition.
답: no, from the goups of each parition → from each parition

12. Software components are often composite components that are made up of several interacting objects. You access the functionality of these objects through the defined component interface. Testing composite components should therefore focus on showing that each object behaves according to its specification.
답: no, each object → component interface

13. The following are general testing guidelines for interface. Design tests so that parameters to a called procedure are at the extreme ends of their ranges. Always test pointer parameters with null pointers. Design tests which cause the component to fail. Use stress testing in message passing systems. In shared memory systems, vary the order in which components are activated.
답: yes

14. System testing during development involves integrating components to create a version of the system and then testing the integrated system. The focus in system testing is testing the interactions between components. System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces. This must be conducted with a separate users.
답: no. separate users → development team

15. Test-driven development (TDD) is an approach to program development in which you interleave testing and code development. Code should be written before test and 'passing' the tests is the critical driver of development. Test-driven development (TDD) has number of benefits: Every code segment that you write has at least one associated test so all code written has at least one test. A regression test suite is developed incrementally as a program is developed. When a test fails, it should be obvious where the problem lies. The tests themselves are a form of documentation that describe what the code should be doing.
답: no. Code should be written before test → Test should be written before code

16. Regression testing is testing the system to check that changes have not 'broken' previously working code. Tests must run 'successfully' before the change is committed. In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. Not all tests are needed to rerun every time a change is made to the program.
답: no. Not all tests → All tests

17. User testing is essential, even when comprehensive system and release testing have been carried out. The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These can often be replicated in a testing environment.
답: no. can often be replicated → can not be replicated

18. List the UML diagrams you will use while working on the OMT (Object Modeling Technique) in order of work process.
답: class diagram, sequence diagram, state diagram, activity diagram

19. Choose all that are advantages of an explicit architecture.
1) maintenance
2) stakeholder communication
3) project management
4) system analysis
5) large scale reuse
6) job allocation
답: 1), 2), 4), 5)

20. Choose all that are correct among the descriptions related with software project management.
1) Software product is intangible and flexible
2) Software development process is standardized
3) Many software projects are 'one-off' projects
4) It may not be possible to appoint the ideal people to work on a project
5) Productivity is proportional to the number of people working on a task
6) Critical path is the shortest path in an activity network and shortest time to complete a project
7) Risk management process consists of risk identification, risk analysis, risk planning, and risk monitoring

답: 1), 3), 4), 6), 7)
2) standardized → not standardized
5) proportional → not proptortional

21. Test cases are a set of input data and expected output data. The software tester runs a series of test cases to check if the execution passed or not to check for any defects in the software. If it fails, it is very likely that the software is defective. However, the premise is that the test case is well made. The test case uses numeric data, not characters.
답: no. Test case uses characters.

22. The fundamental stages of system engineering contain conceptual design, procurement, development and operation. The conceptual design sets out the purpose of the system, why it is needed and the high-level features that users expect to see in the system. This design process involves the following activities: concept formulation, problem understanding, system proposal development, feasibility study, system structure development and system vision document. Of these, what are related with the next procuring process?
답: system vision document.

24. verification vs validation
- verification : 문서를 충족 by inspection → quality 판단 가능 : standard, portability, maintainability
- validation : 요구사항을 충족 by testing → performance, usability, Availability

25. 분산 시스템에서는 stress test가 중요하다. 그 이유를 기술하시오.
: 분산 시스템은 규모가 확정적이지 않은 시스템이기 때문에 언제 어떻게 확장이 이뤄질지 모른다. 그래서 시스템이 과부하를 얼마나 견디고, 과부하 error가 발생했을 때 연계적으로 다른 재앙이 발생하지 않는지 점검해 주는 stress test가 중요하다.
Stressing the system often cause detects to come to light that would not normally be discovered.

26. equivalence partitioning을 사용하는 이유를 기술하시오.
모든 경우에 대하여 전부 testing이 불가능하기 때문에 partition에 대한 대표값을 뽑아 testing 한 다음 그 결과를 partition에 일반화시킨다.

27. client-server model : 분산시스템
분산시스템의 장점
- Resource sharing : HW, SW, resource를 공유할 수 있다.
- Openness(개방성) : standard protocol로 다른 vendor의 장비와 SW를 합칠 수 있다.
- Concurrency : concurrent processing으로 performance를 높일 수 있다.
- Scalability(확장성 용이) : resources를 추가하기 편하다. 하지만 network capacity를 생각해야한다.
- Fault tolerance(결합내성) : fault가 발생하더라도 계속적으로 service를 제공 가능.
  * 추가적으로 transparency : 내부가 안보이게 interface로 이용

분산시스템의 단점
- Complexity : 설계, 구축, 관리면에서 centralized system보다 복잡하다. 각자 만든 후 시스템이 이를 수용하므로 complexity가 높다.
- Security : 외부 공격에 당하기 쉽다.
- manageability : 각 machine는 서로 다른 타입과 버전, faults propagation을 가지고 있기에 어렵다.
- unpredictability : system organization과 network load에 따라 예측할 수 없는 반응이 존재한다.