TOPCIT

ESSENCE Ver. 3

Technical Field

01 Software Development

TOPCIT

ESSENCE Ver. 3

Technical Field

01 Software Development

TOPCIT ESSENCE is published to provide learning materials for TOPCIT examinees.
The TOPCIT Division desires the TOPCIT examinees who want to acquire the necessary practical competency in the field of ICTto exploit as self - directed learning materials.

For more information about TOPCIT ESSENCE, visit TOPCIT website or send us an e-mail.

As part of the **TOPCIT ESSENCE** contents feed into authors' personal opinions, it is not the TOPCIT Division's official stance.

Ministry of Science, ICT and Future Planning
Institute for Information and Communications Technology Promotion
Korea Productivity Center

Publisher TOPCIT Division
+82-2-398-7649 www.topcit.or.kr helpdesk@topcit.or.kr

Date of Publication 1st **Edition** 2014. 12. 10.
2nd **Edition** 2016. 02. 26.
3rd **Edition** 2020. 02. 26.

Copyright © Ministry of Science, ICT and Future Planning
All rights reserved.
No part of this book may be used or reproduced in any manner whatever without written permission.

TOPCIT ESSENCE Ver. 3

Technical Field

01 Software Development

TOPCIT

ESSENCE Ver. 3

Technical Field

01 Software Development

TOPCIT ESSENCE

CONTENTS

TOPCIT ESSENCE

CONTENTS

TOPCIT ESSENCE

CONTENTS

CONTENTS

TOPCIT ESSENCE

CONTENTS


Software Development

I. Software Engineering Overview

▶▶▶ Recent trends and major issues

In June 2015, Gartner, a research company specializing in the IT field, announced that the global software market in 2014 was worth USD 1 trillion and 269.2 billion, that software was occupying an ever greater share of the entire ICT market, and that the influence of software in all industries was rapidly increasing as all industries were becoming "smart" and using "smart data". Therefore, systematic software development and management has become an important factor in determining system quality and improving development productivity.

▶▶▶ Learning objectives

1. To be able to explain the characteristics and problems of software.
2. To be able to explain the background to and purpose of software engineering.
3. To be able to explain software development process models.

▶▶▶ Keywords

· Characteristics of software
· Lifecycle of software
· Requirements analysis, design, implementation, testing
· Software requirement management, software maintenance, software configuration management, software quality management

+ Preview for practical business What if we don't have sufficient software development and management capabilities?

"Software is devouring the world."

These words accurately describe the current situation facing companies. As we enter the intelligent information society represented by Industry 4.0, software has emerged as a key driver of digital transformation and enterprises have their competitors strained in the industry by creating a new business model that did not exist before the widespread adoption of software. We can see evidence for this in the way that Uber and Airbnb have stablished themselves as the providers of the world's largest transportation service and its largest accommodation service, respectively, even though they do not own any cars or hotels. As a result, many enterprises all over the world are trying to strengthen their software capabilities for digital business. The impact is so far-reaching that even the manufacturing company GE and the financial company Goldman Sachs also see themselves as software companies. We can easily find evidence of the newly elevated status of software all around us. Artificial intelligence (AI) is the most representative example. "AlphaGo", which was introduced to the world by Google DeepMind a few years ago, became an important measure by which we can estimate the extent to which the power of software will develop. At present, AI enriches our lives considerably as it is utilized in diverse fields in our daily lives, such as communication, education, shopping, entertainment, vehicle management, etc.

Autonomous driving, which has been actively researched and developed throughout the world, also belongs to the software domain. Until very recently, mechanical engineering elements had a dominant place in the car and electric apparatus industries, which were even referred to as the "flowers of the manufacturing sector". However, more than half of the industries are now composed of software code. In particular, the car industry depends increasingly heavily on software as self-driving technology is increasingly incorporated into new models.

Even though such important software is increasingly being used in more areas, the pertinent question is whether software quality is being managed, and productivity being improved, in proportion to the increased use. In particular, it is necessary to examine whether software functions are working properly as the use of software is increasing even in areas directly related to human life or property, such as autonomous driving. However, the reality is that software quality cannot be considered in the domestic software industry. For example, more mobile apps are now used as smartphones have become so popular. However, these mobiles apps are not used by a single user but rather are called by several users at the same time. Therefore, unintended defects can occur while seeking to improve service efficiency.

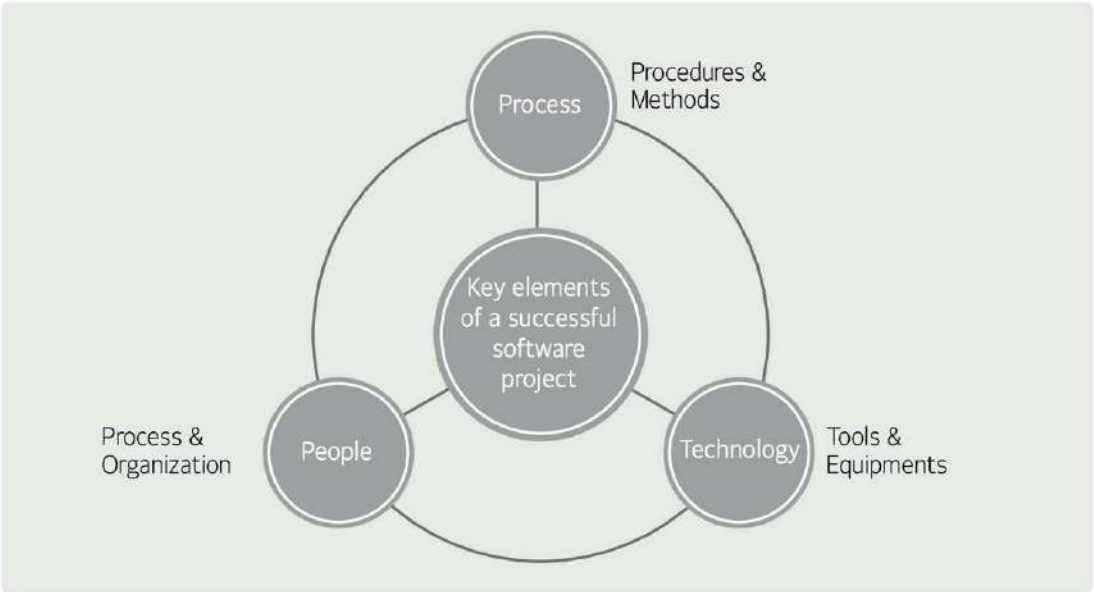
As such, even though software is prone to be defective due to its intrinsic nature, the domestic software industry tends to overlook it on such pretexts as a lack of budget or a shortage of manpower. One can understand their view that they don't have any other options. It is natural that the software development period and cost increase when quality is taken into account. This phenomenon is further aggravated by continuous issues that have been pointed out for a long time, but which still have not been resolved, namely, the difficulty of getting the right software price, the lack of developers, a development culture centered on SI projects, etc. [Source] "Quality improvement is essential in the software-centered era." - https://blog.naver.com/rsupport/221358144195

In the past, software development projects could be conducted normally without systematic implementation simply by relying on the experience of programmers. However, today's large projects involving hundreds of developers face many challenges, such as communication, schedule and cost management issues that arise from the lengthy development period and changes to ambiguous and complex requirements, etc. Therefore, it is essential to understand how to develop and manage software systematically and to improve enterprises' competencies, in order to develop software of the highest quality and promote productivity.

01 Background to and Purpose of Software Engineering

A) Introduction to software engineering

It is most important to apply software engineering technology to successfully develop software that becomes more multifunctional and bigger in scale, which provides technologies and techniques that support systematic management to resolve the expected difficulties throughout the entire process, ranging from requirements analysis to maintenance. To effectively apply software engineering technology, it is necessary to prepare and maintain three key elements in a balanced and harmonious manner and make continuous efforts to maintain these three elements, namely the process of applying the definition of systematic business methods and flow, the organization and people equipped with specialized knowledge, and the infrastructure and technology required for efficient operation of the defined work methods and organizational personnel.



B) Background to software engineering

Let's look at the history of software engineering. In the 1950s, the concept of software engineering, like hardware engineering, began to be introduced in order to implement software development projects. In the 1960s, software engineering was introduced in earnest because of the "software crisis", which occurred due to the lack of experience and competence of the engineers involved and the lack of skilled engineers, while demand for software increased rapidly. In the 1970s, software developers became even scarcer as the demand for software increased rapidly. To solve this problem, many people who did not major in software development were assigned to projects. They adopted a "coding first and correction later" approach, believing that software could be modified easily during its development.

However, structural or formal techniques were developed as many defects were discovered as a side effect of the "coding first and correction later" approach. They began to develop and use the Waterfall model, which

proceeds with analysis, design, and implementation sequentially. However, formal techniques were less usable for general software developers, and the Waterfall model was expensive and slow to implement. Recognizing these problems, methods of boosting the productivity of software development were studied to efficiently develop software by increasing its reusability in the 1980s. In the 1990s, software vendors had to reduce the 'time to market' to gain a competitive advantage. As a result, much research aimed at raising software development productivity was conducted, and models focusing on concurrent engineering - which could carry out requirement definition, design, implementation in the Waterfall model simultaneously - were adopted. As the market environment surrounding software began to change rapidly in the 2000s, the agile methodology was comprehensively adopted to effectively respond to such rapid changes.

C) Four key elements of software engineering

Software engineering is defined as "a discipline that studies the overall life cycle of software - such as development, operation, and maintenance - systematically, descriptively and quantitatively." The following section describes the four key elements of software engineering, which make it possible to produce top quality software and deliver it according to a given cost and schedule.

① Method

- The method is composed of project planning and estimation, system and software analysis, data structure, program structure, algorithm, coding, testing, and maintenance tasks.
- Methods centered on a specific language (e.g. an object-oriented method), or graphical notation, are introduced from time to time.
- A series of evaluation standards for software quality is introduced.

② Tool

- "Tool" refers to an automated or semi-automated method that is used to improve productivity or consistency when performing a task.
- There are numerous tools in the software development lifecycle (e.g. requirements management tools, modeling tools, configuration management tools, change management tools).
- When tools are integrated in such a way that the information created by one tool can be used by other tools, it is set as a system that supports software development.

③ Procedure

- A procedure combines a method and a tool so that they can be used to develop software in a rational and timely fashion.
- The procedure defines the applied method, required deliverables (documents, reports, etc.), controls to guarantee quality and assist coordinate adjustment, and the sequence of milestones that enable software managers to evaluate progress.

④ People

- As many tasks (establishment, improvement, maintenance, etc.) are performed by employees and organizations specializing in software engineering, software engineering relatively depends more heavily on people.
- It is practically impossible to summarize software development in an easily accessible or graspable manner

because far more diverse issues arise than in other engineering situations.

02 Lifecycle of Software Development

A) Definition

The lifecycle refers to the entire process from understanding the user environment and problems to operation and maintenance. The general software life cycle is composed of [Feasibility review → Development planning → Requirements analysis → Design → Implementation → Test → Operation → Maintenance] activities.

B) Purposes

- To calculate the project costs and draw up a development plan, and to configure the basic framework.
- To standardize the terms.
- To manage a project.

C) Selecting a software lifecycle

- This is an important activity for corporations to tailor the development process of a project.
- Selection is based on the risk and uncertainty of system development and understanding of it.
- The selected model should be able to minimize the risks and uncertainties associated with a given project.
- The Waterfall model, prototype model, evolutionary model, and incremental model are the most representative life cycle models.

D) Types of software lifecycle models

The lifecycle model does not explain all possible lifecycle models that could be applied to a project; rather, it is an example of the most frequently used lifecycle model. In addition, the lifecycle model can be replaced according to the characteristics of the project.

① V model

- This model clearly shows the activities that should be performed while implementing the project to project managers and developers. It also helps customers who do not know much about this area to understand the principles of software development.
- An ideal lifecycle model identifies and clarifies all system requirements.
- The V model can be used even when the requirements are unclear. However, the estimation, planning, and confirmation of a project are limited to the requirement analysis and identification phase only. That is, the detailed plan to which the V model is applied is finally completed once all the requirements have been identified and clarified. If the requirements are not clear, the initial project plan should specify the start/end dates expected by the customer and the estimated date when major risks, assumptions, dependencies, and requirements have been identified and clarified. A project involving the implementation of a standard communication protocol is a typical example of the type of project to which the V model can be applied. The V model has the following characteristics.

- It is easy to apply to a project and manage.
- A measure that can effectively manage the start/end of development activities and the project can be clearly defined.
- The model emphasizes project verification and validation. In other words, the model explains very well the association relationship between development activities, such as requirements analysis and design, and design activities. The development activities and corresponding test activities are performed at the same time during the entire development cycle.
- By doing so, we can understand which phase of the development activity should be performed again, if a software fault is found during the test.

② V model with prototyping

• Prototyping is a method of developing a system or a part of a system to understand a system or to solve such issues as risks or uncertainties. Prototyping can lead developers and customers to commonly understand what is needed and what should be developed. Prototyping can be applied to the development phase of the Waterfall model or V model, and can also be used as an independent lifecycle model.

• The uncertainty factors and risks of a project can be reduced by adding the prototyping technique to the V model. The uncertainty factors and risks of a project include the possibility of implementing the requirements in the requirement phase, the status of verifying system performance in the design phase, and risk factors when introducing new tools and developing a system by outsourcing. The prototyping technique can be performed using the following two approaches.

- ○Approach 1: Seek applicable solutions and apply them. This approach can be applicable when the problem to be solved is not clear. For example, this approach can be used when implementing the user interface of a new device, when seeking a method of improving a system's performance, or when managing errors or faults.
  - The general procedure for performing actions is as follows:
    1. Define the uncertainty factors.
    2. Seek a solution and define how to apply it.
    3. Try to apply the solution. (It can be reiterated.)
    4. Find the cause of the uncertainty factor based on the result of step 3.

○Approach 2: Enumerate several solution options and evaluate them according to the established standards. This approach is appropriate when there is a risk or uncertainty factor in the solution to be applied. For example, this approach can be used if middleware should be selected for a specific function or if the performance of a design for several environmental factors should be taken into account.

- The general procedure for performing actions is as follows.
  1. Define the uncertainty factors.
  2. Enumerate the solutions that can be selected and define the selection criteria.
  3. Evaluate the solutions according to the selection criteria.
  4. Select the most suitable solution.

③ Incremental model

- The incremental model is useful when the system development time needs to be reduced. For example, this model is appropriate when a system should be developed by the date the customer wants, even though not all of the functions can be developed by that time. That is, the core parts are developed first to make the system operable, and then the rest of the functions are implemented.
- A system is developed by extending the functions several times. The system version in each phase runs only a few limited functions, and the later version includes newly added functions in addition to those of the previous version. Of course, the final system version is a complete system into which all of the functions are incorporated.
- This model is also useful when most of the requirements are defined, but improvements can be made as time goes by.
- This model can be usefully applied to reduce the risk of external interfaces (H/W and S/W) in the early phase of development.
- Both the V model and the VP model can be applied since the requirements for each phase for development of a function expansion are generally clear.

④ Evolutionary model

- Like the incremental model, the evolutionary model is useful when the system development time needs to be reduced. However, unlike the incremental model, the development phase for the entire system is reiterated several times. Each system provides all of the functions to the user.
- Next, changes are identified when a system is developed and used, and then those changes are reflected in the development of the next system.
- Changes include changes to the function, changes to the user interface, and changes to non-functional items such as reliability and performance improvement, etc. This model is suitable when the overall system specification is not certain or when the product should be continually improved.
- In practice, this model is used in combination with the incremental model. That is, new functions can be added to a new system version, while the existing functions are improved. A mixed model such as this has the following characteristics.

- Users can be trained about a system early on even though the system is not fully functional. Users can familiarize themselves with the system and identify improvements necessary for practical business, because they can use the system after receiving initial training. By doing so, a system can be developed that is needed by practical business.
- Also, competitiveness can be improved by reducing the development period and using the system in the early stages.
- Since a system is developed over time, unexpected problems with the system can be identified and fixed early on.
- Or a system can be developed by dividing it by specialized field. For example, if the functions need to be improved from a text-based user interface to a graphic-based user interface, a development team specializing in user interface can implement the project.

03 Software Development Methodology

The software development methodology defines each development phase, the activities to perform in each phase, the deliverables, the verification procedure and completion criteria, and then defines the standardized method and procedure, and support tool for the development plan, analysis, design, and implementation phase.

A) Necessity of the software development methodology

① Improving development productivity by accumulating and reusing development experience.

② Effective project management.

③ Providing a means of communication by presenting formal procedures and deliverables and unifying the standard terminology.

④ Assuring quality at a certain level by verifying each phase and closing the phase after approval.

B) Comparison of software development methodologies

<Table 1> Software development methodology

Item	Structural methodology	Information engineering methodology	Object-oriented methodology	CBD methodology
Overview	Focusing on business activities	Focusing on data	A methodology that identifies the relationship between object and class and converts it into a design model.	A methodology that develops a reusable component or combines commercial components.
Basic principle	- Abstraction -Structuralization - Stepwise refinement - Modularization	- Information strategy plan - Business area analysis - Business system design - System development	- Definition of requirements - Object-oriented analysis (object modeling, dynamic modeling, functional modeling) - Object-oriented design - Test/deployment	- Requirement analysis - Analysis (architecture definition, use case modeling) - Design - Development - Implementation (release, training)
Characteristics	- The principle of divide and conquer - Centered on program logic - Structured with controllable modules	- Methodology of supporting corporate business support systems - Emphasis on data models - Program logic depends on data structure (CRUD) - Enterprise integrated data model	- A program unit is an object. - Data and logic integration - Advanced modularization - Reuse by inheritance No gap between analysis and design	- Evolution of the object methodology - Emphasis on interface - Interface implementation using a component - Aiming to reuse black box

Item	Structural methodology	Information engineering methodology	Object-oriented methodology	CBD methodology
Major deliverables	- Domain analysis report - Data flow diagram - Structural drawings - Program specification	- Domain analysis report - ERD - Function chart - Application structure diagram - Program specification - Table definition/list	- Business process/ conceptual diagram - Diagrams (use case, sequence, class, component, etc.)	- Business process/ conceptual diagram - Diagrams (use case, sequence, class, component, etc.) - Reuse plan - .ent, EJB
Supporting tool	Teamwork, SA	Cool Gen, SA	Rose, SA, Palstic	Cool Joe, Together
Major supported language	COBOL, C, VB, PASCAL	COBOL, C, VB, PASCAL	C++, JAVA, VB	In principle, the choice of development language is unimportant.

C) Software development phases

Software development activities are defined by the software lifecycle.

① Requirements analysis

- The hardest thing in software development is deciding exactly what to develop.
- Software requirements analysis is practically the first step in software development and is the phase of understanding what the user needs.
- It is a critical phase that can reduce the development costs in the entire development phase.
- If investment is channeled into analyzing, defining and managing the requirements well in the early stages, the entire software development period can be shortened and excessive costs and quality deterioration can be prevented in advance.

② Design

- The software requirements analysis process is a conceptual step, whereas the design process is the first step in physical realization.
- When designing a system, the structure of a system, which is composed of sub-systems, is determined, and sub-systems are allocated to the components, such as hardware or software.
- Design directly affects quality.
- If a system is not designed properly, its stability deteriorates.
- Unstable systems are difficult to maintain

③ Implementation

- The goal of the software implementation phase is to program it in a way that the requirements can be satisfied based on the design specification.
- The program should be written according to the description in the detailed design and user's guide.
- One of the most important tasks in the implementation phase is to decide on a coding standard and write the code clearly based on that standard.

④ Testing

- Testing refers to a series of processes to inspect and evaluate whether the system satisfies the prescribed requirements, and how the expected and actual results differ, using the manual or automated method.
- Testing is the final step in assuring software quality. It is a series of tasks designed to find faults so as to ensure software quality.
- Testing includes a quality evaluation of the developed software and a series of modification tasks for improving quality.

04 Agile Development Methodology

A) Types of agile methodologies

Since the release of the Agile Manifesto for Software Development, various agile methodologies have emerged. Among them, Scrum and Extreme Programming (XP) are the most widely adopted agile methodologies throughout the world. XP was mainly used in the early days, but Scrum and XP are generally used together these days, as Scrum has gained popularity. Recently, the Lean software development methodology, whereby the lean production method of the Toyota system is applied to software development, has emerged. It is mainly used with Scrum.

The main agile methodologies are as follows:

- Scrum, Ken Schwaber/Jeff Sutherland
- EXtreme Programming (XP), Kent Beck/Erich Gamma
- Lean software development methodology, Mary Popendic/Tom Poppendieck
- Agile Unified Process (AUP), Scott Ambler

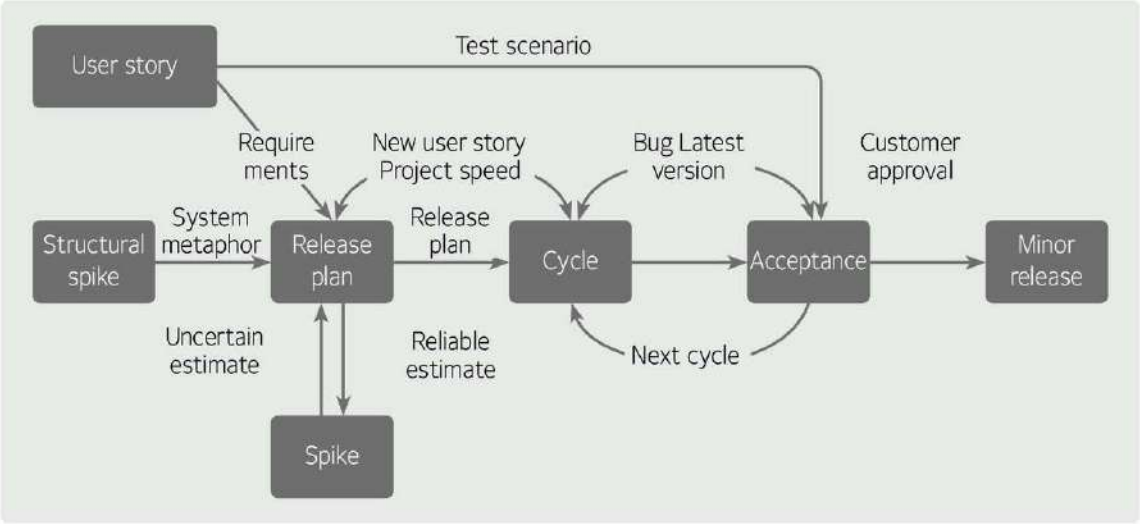
B) Agile development methodology - XP

① XP overview

XP (eXtreme Programming) is a methodology that was established by Kent Beck and several other engineers in the late 1990s, based on the lessons they learned while implementing projects. This methodology is a lightweight development method generally suitable for small and medium-sized development organizations. There is some controversy about whether to define XP as a "methodology" because it is largely related to development techniques such as Test-Driven Development (TDD), Daily Build, and Continuous Integration. In Korea, small development teams usually apply some of the techniques. In addition, more teams apply additional agile development methods such as Scrum together, rather than insisting on XP alone. XP is composed of the value and the practice of achieving the value when it is applied, and a principle is needed to maintain the balance between the two. XP is a kind of iterative development methodology. There are several iterations during the course of a project, and tasks are repeatedly performed depending on the test results.

② XP development procedure and terminology

The XP development procedure is performed as shown in the following figure.



- User stories: Refers to a tool for collecting requirements and communication. User stories briefly describe necessary matters regarding functions.
- Spike: A simple program that considers difficult requirements or potential solutions. Spike aims to increase the reliability of user stories while reducing the risk of technical problems.
- Release planning: Establishes a deployment plan for an entire project and divides one iteration into one to three weeks, and then keeps the iterations evenly.
- Acceptance test: An acceptance test performed by the customer before release.
- Smaller releases: The final phase of the XP cycle. If frequently released on a small scale, several benefits can be provided to the customer in the early stages.

③ XP values

Basically, eXtreme Programming proposes the following five values.

- Communication: Communication is the most important thing in software development at the team level. Communication errors are typically found in failed projects. Problems should be solved, and team building should be strengthened through communication between customers, developers and managers.
- Simplicity: Always ask "What is the simplest thing possible?". Keep the design simple and clear by removing unnecessary complexity.
- Feedback: Gradual improvement is more effective than pursuing perfection, considering the flow of changes which causes yesterday's output or value to become useless today. Create as many feedback as possible quickly as long as they can be handled by the team and use them for improvement.
- Courage: Cope with changes to the requirements and technology to reflect them as quickly as possible and deliver to the customer. The courage to actively solve a problem entails simplicity, and the courage to present the solution concretely reinforces feedback.
- Respect: Respect is the value that hides behind the first four. A person cannot implement the project properly without respecting the other members of the team. (Source: Korea Developer's Hope Report)

④ Extreme programming practices

XP suggests using twelve practices as described in <Table 2>, in addition to the aforementioned five values.

<Table 2> Basic XP practices

Item	Practice	Description
Development	Simple design	Keep the design as simple as possible to meet the current requirements.
	Test-driven development	Write the test programs first before writing the code and automate them using the test tools.
	Refactoring	Design the existing code by eliminating code duplication and complexity.
	Coding standard	Establish coding standards for effective communication.
	Pair programming	Have two developers sit in front of a computer and work together for development.
	Collective code ownership	Have all the developers in the team assume joint responsibility for the source code so that everyone can modify the code to make it better at any time.
	Continuous integration	Perform integration work until the work is finished.
Management	Planning game	Establish the entire project plan and cycle plan by considering the business and technical aspects and keep them updated through execution and feedback.
	Small release	Deploy executable modules as quickly as possible so that customers can frequently experience how the software works.
	Metaphor	Express the overall look of the system using pictures and stories that can be understood easily.
Environment	40 hours/week	Do not work more than 40 hours per week to maintain quality.
	On-site customer	Have the customer who actually uses the system stay at the development site.

C) Scrum

① Scrum overview

Scrum is one of the agile methodologies designed for project management and is a representative type of empirical management technique based on estimation and adjustment. Scrum for software was directly modeled after "The New Product Development Game" by Hirotaka Takeuchi and Ikujiro Nonaka published in the Harvard Business Review in 1986. This method was named "Scrum" when it was introduced by Ken Schwaber and Jeff Sutherland for software development in 1995. Scrum consists of three fundamental units, as shown in <Table 3>.

<Table 3> Type of Scrum units

Roles	Description
Product owner	The product owner manages the creation of a product backlog corresponding to the product function list, priority adjustment, or new item addition. It is recommended that the product owner should play a key role when setting up a splint plan, but should be involved in team operation as little as possible once the sprint starts.
Scrum master	The Scrum master endeavors to remove any factors that disturb the team's work. The Scrum master supports the team responsible for development while upholding the principles and values.

Scrum team	Generally, a Scrum team is composed of five to nine members. The Scrum team identifies the functions to be developed during a sprint using user stories.

② Scrum process

The Scrum process has the following three components.

- Sprint: Sprint refers to the repetitive development period in units of 1 to 4 weeks by the calendar.
- Three meetings: Daily Scrum, Sprint plan, Sprint review
- Three deliverables: Product backlog, Sprint backlog, Burndown chart

The three deliverables can be summarized as shown in <Table 4>.

<Table 4> Scrum process deliverables

Roles	Description
Product backlog	The product backlog is a breakdown of work to be done, and the product manager mainly determines the priority on behalf of the customer. The function defined in the product backlog is called user story. The standard called 'story point' is mainly used to estimate the user's workload.
Sprint backlog	A list of works to be developed during a sprint. The user story and the work required to complete it are defined as a task. The size of each task is estimated by the hour.
Burndown chart	The sprint burndown chart shows remaining work in the sprint backlog. The chart shows remaining work for each iteration as a story point.

The three deliverables of the Scrum process can be summarized as shown in <Table 5>.

<Table 5> Scrum process deliverables

Roles	Description
Sprint planning	Goals are set for each sprint and items are selected for execution during a sprint from the product backlog. A person is appointed to take charge of each item and draw up a plan for each task.
Daily Scrum	A meeting that is held for 15 minutes each day to share the progress of the project. All of the team members should attend and discuss what they did, what to do and other issues every day.
Sprint review	A meeting held to check the work progress and deliverables in order to see if the sprint goal has been achieved. The Scrum team demonstrates what they did during the sprint to the attendees and receives their feedback. It is recommended to run a demo for all works performed during the sprint concerned with the participation of the customer. At this time, the Scrum master conducts a retrospective review to find out what went well, what was disappointing, and what should be improved upon during the sprint.

③ Characteristics of Scrum

Scrum has the following characteristics.

- Transparency: It is difficult to exactly understand the current status of the project, progress compared to the plan, and identified issues. Scrum enables the user to effectively understand the status or problem of a project using various techniques, such as the Scrum meeting, burndown chart, and sprint review.
- Timeboxing: Concentration on a project only becomes possible by limiting the amount of time it takes to implement Scrum. The daily Scrum is performed for a limited time of 15 minutes each day, and the sprint review is performed periodically for each iteration.
- Communication: Much effort is made during Scrum to facilitate communication among the team members. One of the characteristics of Scrum that makes communication among team members smooth is the procedure by which developers share their problems during the daily Scrum and discuss the difficulty/time of

implementing user stories using the planning poker.

- Empirical model: Scrum has its own process model, but many techniques place emphasis on the experience of the individuals participating in the project. As each project has its own intrinsic situation and characteristics, it is difficult to understand the project situation using the existing standardized process. Scrum accepts that something can actually be different for each team even though the basic structure is the same.

II. Software Reuse

▶▶▶ Recent trends and major issues

We can see that enterprises are increasingly reusing their software as their business becomes globalized. Software is reused to develop and maintain new features using knowledge proven in the past. Many enterprises are considering how to increase the reuse rate because it is a highly effective means of reducing costs through productivity and quality improvements thanks to reproduction of software in a short period of time.

▶▶▶ Learning objectives

1. To understand the concept, purpose, and target of software reuse, the actual method of application, and its effects, considerations, etc.
2. To understand the concept of reverse engineering, the reason for its necessity, strengths, and considerations.

▶▶▶ Keywords

- Reuse, reverse engineering
- Code reuse, software standardization
- Software maintenance

✚ Preview for practical business

Establishing a system for creating a reuse culture

Development teams need to change their perception and culture in order to incorporate software reuse into the development process. To reuse software, developers are required to change their way of thinking, and a new culture should be created from the organization and policy perspectives. The following should be performed to fully establish the system.

- Composition of the reuse management organization: The reuse management organization is needed to establish the reuse-based method of software development. This organization should introduce and establish procedures and standards related to reuse and enact and implement a policy for increasing reusability.
- Training on reuse: Training on software reuse should be conducted on a regular basis to create an atmosphere or culture conducive to software reuse within the organization. The training should be also provided to project managers and organization managers, in addition to developers. In terms of content, the training for managers should differ from that intended for developers.
- Introduction of a reward system to increase reusability: Reuse may be perceived with extreme discomfort when first introduced. Therefore, a method should be devised to increase reusability by policy. Introducing an incentive system for reuse is the representative method in this regard.
- Securing reliability for reused components: To make developers trust reused components, the reliability of the components managed in the reuse library should be sufficiently guaranteed and measures to help them find suitable components should be prepared, in addition to the change of developer's awareness.

01 Software Reuse

A) Overview of software reuse

The term "software reuse" means to develop new software using existing software or software knowledge. Reusable software or software knowledge is a reusable asset comprising design, requirements, inspection, and architecture.

① Background to software reuse

- Deterioration of quality and productivity due to a software crisis.
- Increased use of CASE tools due to the development of automation technology in software development.
- Efforts to comply with software development standardization and secure quality.

② Definition of software reuse

- Software reuse is a method of standardizing knowledge about software development (function, module, configuration, etc.) and configuring it to be suitable for repeated use in order to increase development productivity.
- Countermeasures that increase the quality, productivity, and reliability of newly developed software and reduce the development schedule and cost by reusing all or a part of software that has been recognized for its development functions, performance, and quality.
- A concept whereby developed modules, programs and deliverables are directly reused or reused after some modifications in the same application field, or are reused for different application duties or by different companies

③ Purpose of software reuse

Goal	Contents
Responsibility	Performance such as functions, stability, speed, etc. has already been proved.
Scalability	Easy to upgrade based on proven functions.
Productivity	Improvement of the overall development process, such as cost, time, risk, etc.

B) Target of software reuse

Various subjects can utilize software reuse.

① General knowledge

- Environmental information: Knowledge obtained through education and use.
- External knowledge: Knowledge obtained by participating in the development and the specific field.

② Design information

- System architecture
- System design

③ Data information

- System data
- Test cases

④ Program code

- Module
- Program

⑤ Others

- Cost benefit analysis
- User documentation
- Feasibility studies
- Prototype
- People

C) Principle of software reuse

① Generality: Software should be generally reusable, rather than designed for a specific area of application.

② Modularity: The principle of information hiding and abstraction. Software reuse should have the characteristics of minimum coupling and maximum cohesion.

③ Hardware independence: Reused software should be independent of execution hardware as much as possible.

④ Software independence: Reused software should be independent of the OS or DBMS.

⑤ Self-documentation: The accurate function, usage, and interface of the module should be described.

⑥ Commonality: Reused software should be commonly needed and usable by many developers.

⑦ Reliability: Reused software should be reliable during use.

D) Problems when reusing software for practical business

- It is difficult to find software modules that can be used in common.
- Software is not very standardized.
- It is difficult to understand the internal interface requirements of the software module.
- It is difficult to understand the side effects of the change.
- The development of software components for reuse can be more costly.
- The benefits of software reuse may only be recognized after a long time.
- It is impractical to extract components for reuse from existing software components.

E) Obstacles to software reuse and countermeasures

① Obstacles to software reuse

- Adverse reaction of managers and developers.
- Lack of motivation to apply reuse technology.
- Lack of software standardization.
- Social or legal obstacles.

② Obstacles to software reuse and measures to remove the obstacles

- Technical plan
  - Use of a new design and development methodology.
  - Development of the reuse software library.
  - Use of the automation tool (CASE).
- Administrative and institutional methods
  - Establishment of a compensation system.
  - Active management strategy.
  - Organizational change.

F) Considerations when reusing software

- Reuse software if productivity can be improved.
- Software development process based on systematic software reuse.
- Establishment of a system for promoting a reuse culture.
- Creation of a reuse environment through initial investment.
- Continual improvement and enhancement of the library.
- Software reuse that is supported by tools.
- Software productivity evaluation and measures.
- Management of a set of information on reused components.
- Deliverables for software reuse:
  Architecture, source code, data, designs, documents, estimates (templates), human interfaces, plans, requirements, test cases
- Top-down/bottom-up development approach.
- Granularity of reusable components.

G) Effects of software reuse

- Reduces the TCO of software production.
- Creates sharing and utilization effects for producing high-quality software.
- Promotes the sharing of information on system development and the sharing of deliverables from other projects.
- Has an educational effect on the system structure and the method of developing a good system.

02 Reverse Engineering

A) Definition of reverse engineering

Reverse engineering is a field of software engineering in which a developed system is deconstructed to reveal its documents, design techniques, etc. Reverse engineering is a series of activities aimed at understanding and modifying a system, which are performed during the software maintenance phase. That is, information or a document corresponding to the deliverables

in the initial lifecycle phase is created, using the program or document obtained in the last phase of the software lifecycle. Reverse engineering is the opposite concept of forward engineering in which development is performed sequentially from the design. The concept of reverse engineering can be understood by looking at the flow of input and output below.

<Table 6> Input/output of reverse engineering

Input	Output
Data or document in the form of I/O, such as the source code, object code, work procedure, library, etc.	Structure diagram, data flow chart, control flow graph, entity relationship diagram, etc.

B) The main reasons why reverse engineering is necessary

- When it is difficult to maintain a running system.
- When frequent changes decrease a system's efficiency
- When redeveloping a business system based on a file system into one based on a relational database.
- When downsizing the default mainframe

C) Advantages of reverse engineering

- Commercialized or previously developed software can be analyzed.
- Maintainability can be improved since the data and information of the existing system can be analyzed at the design level.
- CASE can be used easily by storing existing system information in a repository.

D) Types of reverse engineering

- Reverse engineering can be divided into logical reverse engineering and data reverse engineering for description.

<Table 7> Types of reverse engineering

Type	Description
Logic reverse engineering	Information is extracted from the source code and stored in the physical design information storage. Physical design information is obtained.
Data reverse engineering	The existing database is modified or migrated to a new database management system.

III. Data Structure and Algorithm

▶▶▶ Learning objectives

1. To be able to explain the definition and classification of the data structure and use the linear/nonlinear structure.
2. To be able to understand the role of the algorithm and select an appropriate algorithm depending on the situation.

▶▶▶ Keywords

· Array, list, stack, queue, deque, tree, graph
· Algorithm definition, algorithm performance analysis, sorting/searching algorithm

+ Preview for practical business

In 2003, a massive blackout occurred in the northeastern United States, plunging seven states in the U.S. and one province in Canada into darkness. When the incident was investigated, it was found that the power outage had been caused by a software error in the XA/21 system, which occurs in the multi-process method. That is, the "race condition" (a type of deadlock) caused the blackout. The error occurred because two programs in the XA/21 system simultaneously wrote and accessed the same data structure. The error corrupted the data structure, which in turn put the alarm process into an infinite loop, and even an alarm processing failure was not notified. The alarm queue increased indefinitely, and all available memory was consumed after 30 minutes. At that time, the main server was down. A failover mechanism was activated, and the backup server became the main server. However, the backup server also failed because it couldn't handle the infinitely increasing queue.

This incident is a representative case of software causing a fatal error resulting in tremendous damage to real life, because the software was developed without a proper understanding of basic software technologies such as the data structure and algorithm.

A data structure, which is a way of structurally expressing data in programming, presents a way to conveniently access data and store or organize them for the purpose of changing the data structure. An algorithm refers to a well-defined calculation process to output after solving the problem by receiving a certain value.

For example, let's consider a case in which a house is built with blocks. We can build the best house only when we select and stack the most appropriate block for a given design, such as material, color, and shape among several blocks, depending on the structure or shape of the house. However, what is far more important than this is that we have to properly understand the materials and characteristics of each block and the various utilization methods. If we don't use the blocks in the right place, a shoddy and faulty construction is inevitable when the house is completed. The data structure and algorithm are like these blocks. Various basic technologies are available that keep pace with the rapid development of information systems. Therefore, we can expect high-quality software that is optimal for a given environment, if we develop software by adopting and applying the most appropriate data structure and algorithm.

The data structure is an important technique in terms of coding efficiency. An algorithm enables us to select more easily a data structure that can solve a given problem, and also allows us to handle a given problem efficiently, considering the processing time and storage location.

01 Data Structure

A) Definition

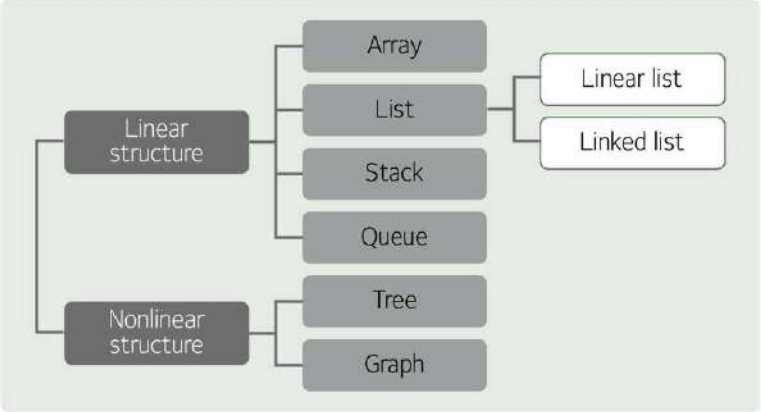
A data structure is a method of storing data in the memory device of a computer. It defines data systematically so that various data can be efficiently expressed and utilized, considering the characteristics and usages of data.

B) Classification

The data structure can be broadly divided into linear and non-linear structures. The linear structure is a method of organizing data in such a way that they are connected in a row, whereas, in the non-linear structure, data have a special form such as a hierarchical structure.

<Table 8> Classification of data structures

Classification	Description
Linear structure	Data are arranged in a straight line by considering the sequence of the data. Data are arranged in a one-to-one relationship between data and before/after, adjacent to, previous/next data. The types of linear structure include array, linear list, linked list, stack, queue, and dequeue.
Non-linear structure	It is a structure in which several data exist after one data. Data adjacent/before and after data are arranged in a one-to-many or many-to-many relationship. The types of non-linear structure include tree and graph.



[Figure 1] Classification of the data structure

<Table 9> Comparison of sequential data structures and linked data structures

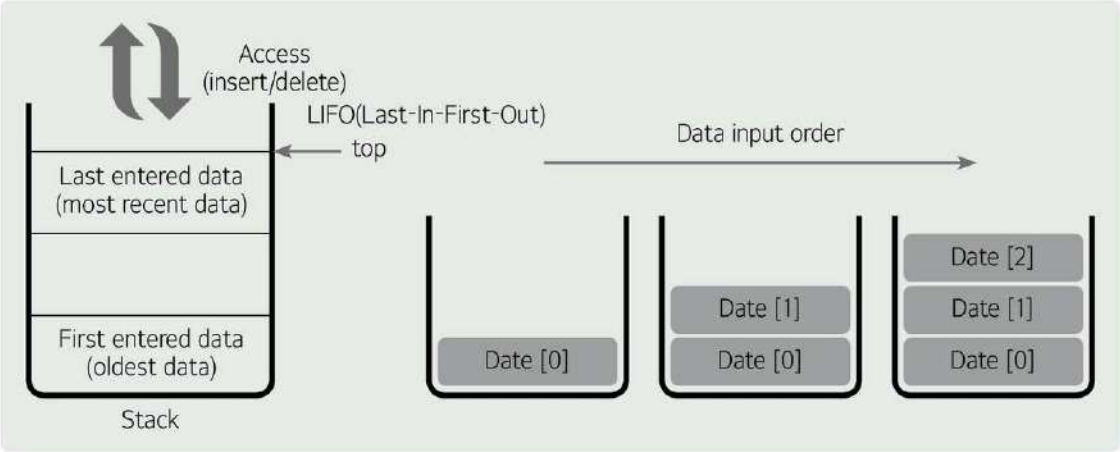
Item	Sequential data structure	Linked data structure
Memory storage method	When saving data in the memory, data are stored consecutively in order without an empty space.	Logical sequences are expressed by link, regardless of the physical location or sequence stored in the memory.
Logical/physical sequence matching status	The logical and physical sequences match.	The logical and physical sequences do not match

Item	Sequential data structure	Linked data structure
Characteristics of operations	Data are stored consecutively in order because no empty space is created even if insert/delete operation is performed.	The physical sequence does not change even though the logical sequence changes due to insert/delete operation, because only link information changes.
Programming technique	Implementation using an array.	Implementation using a pointer.

C) Stack and queue

① Stack

A stack is one of the linear lists. In this data structure, data are saved in the storage space in order of data input, and the last input data are output first. That is, an element stored in the stack can be accessed only from the place designated as the top, and an element is inserted at the top only and the inserted element is stacked on the top and output first.



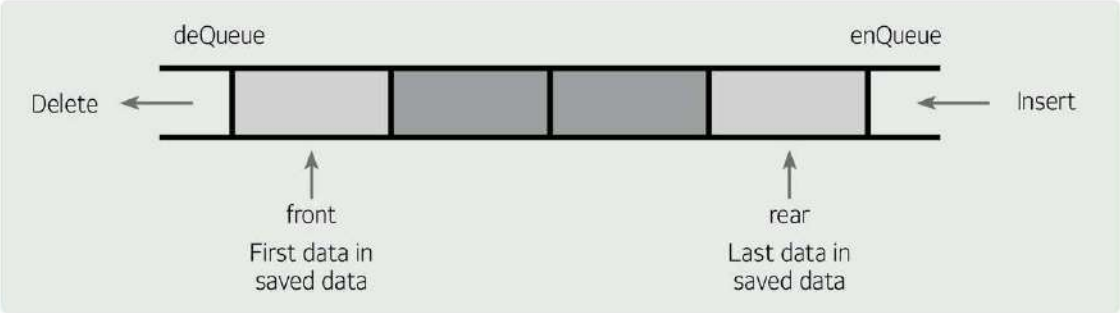
[Figure 2] Conceptual diagram of the stack

The types of stack operations are as follows.

- top(): Returns the data value at the top of the stack.
- push(): Inserts data on the stack.
- pop(): Deletes and returns data from the stack.
- isempty(): Returns 'true' if there is no element on the stack or returns 'false' if there is an element on the stack.
- isfull(): Returns 'false' if there is no element on the stack or returns 'true' if there is an element on the stack.

② Queue

Like the stack, the location of insertion and deletion is limited. However, unlike the stack, the place of data insertion is different from that of deletion. The queue has a structure in which an element can be inserted at the bottom and deleted at the top only. Elements are listed in order of insertion, and the element inserted first is at the front and thus is deleted first.



[Figure 3] Conceptual diagram of the queue

The types of queue operations are as follows.

- enQueue: Inserts data into the queue. Frees up space in the queue by moving the rear, and inserts data.
- deQueue: Deletes data from the queue. Moves the front to pass the oldest data to the position of the next data.

③ Comparison of stack and queue operations

<Table 10> Comparison of insert and delete operations on stack and queue

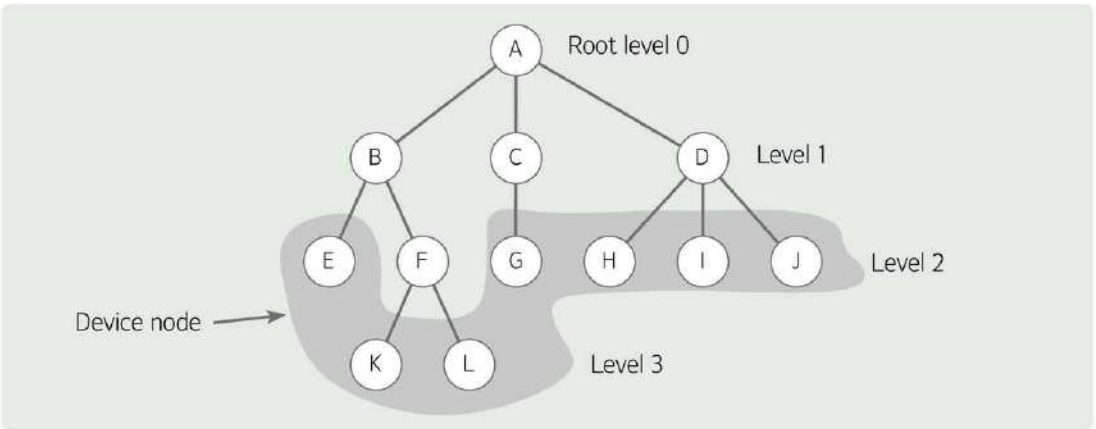
Item	Insert operation		Delete operation	
Data structure	Operator	Insert location	Operator	Delete location
Stack	push	top	pop	top
Queue	enQueue	rear	deQueue	front

D) Tree and graph

① Tree

A tree is a hierarchical data structure with a hierarchical relationship between elements. It has a tree-like structure that expands from top to bottom, and the elements have a one-to-many relationship.

The topmost node in a tree is called the "root node", and the line connecting the nodes is called the "edge". Child nodes having the same parent node are called "sibling nodes", and the tree created when the edge to the parent node is disconnected is called a "subtree".

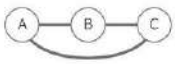
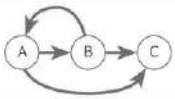
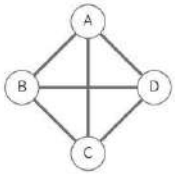
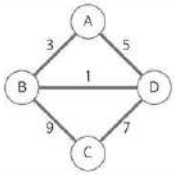


[Figure 4] Conceptual diagram of the tree

② Graph

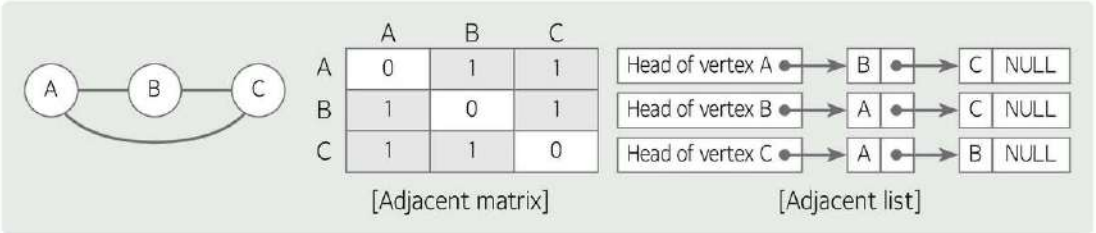
A graph is data structure that expresses the many-to-many relationship between the connected elements, and is a set of vertices that represent objects and the edges that connect those objects. Graph data structures often solve problems based on graph theory by expressing several complex problems on a graph, such as electric circuit analysis, shortest distance search, artificial intelligence, etc.

<Table 11> Types of graphs

Type	Description	
Undirected graph	A graph that is connected by undirected lines between the vertices of the graph.	
Directed graph	A graph whose connection lines are directional, where the sequence of vertices expressed in pairs is important.	
Complete graph	A graph that has as many edges as possible by connecting all the vertices with each other on the graph.	
Weighted graph	A graph in which a weighted value is assigned to the edges that pass through the vertices of the graph.	

The graph is expressed in memory using various graphing methods depending on the function being performed or method of application.

- Adjacency matrix: A method of graphing using a sequential data structure. The presence of the edge connecting two vertices of the graph as a matrix is stored in a matrix by using a two-dimensional array for the matrix.
- Adjacency List: A graphing method using a linked data structure. Nodes are connected as many times as the sequence number of each vertex using a simple linked list created by connecting adjacent vertices for each vertex.



[Figure 5] Graphing method

E) Data structure selection criteria

①Data processing time
②Data size
③Data utilization frequency
④Data update frequency
⑤Ease of programming

F) Utilization of data structures

Data structures are mainly used in data sorting, search, file organization, and index. The utilization fields by data structure can be summarized as follows.

① List: Array implementation, DBMS index, problems such as search or sorting, etc.

② Stack: Interrupt handling, sequence control of the recursive program, saving the return address of the subroutine, calculating an expression represented in postfix notation, undo function of the text editor, etc.

③ Queue: Job scheduling of the operating system, queue processing, asynchronous data exchange (file I/O, pipes, sockets), keyboard buffer use, spool operation, etc.

④ Deque: A data structure that makes the most of the strengths of the stack and queue only, and which is mostly used in the fields related to the stack and queue.

⑤ Tree: Problems like search and sorting, grammar parsing, Huffman code, decision trees, games, etc.

⑥ Graph: Computer network (local area network, internet, web, etc.), electric circuit analysis, binary relation, simultaneous equations, etc.

02 Algorithm

A) Algorithm overview

① Algorithm definition

An algorithm is a description of a series of processing procedures used to solve a given problem by stages. It is a specification that logically describes a step-wise procedure by abstracting the problem-solving method. The algorithm is not designed simply to obtain the desired result, but also to develop an efficient method in terms of processing time and storage space use.

② Conditions of algorithms

<Table 12> Conditions of algorithms

Condition	Description
Input	More than one data that are needed to execute an algorithm should be entered from outside.
Output	More than one result should be output after executing an algorithm.
Definiteness	Instructions for each processing step of the algorithm, indicating the content and sequence of the job to perform, should be clearly specified.
Finiteness	An algorithm must be terminated after execution.

Condition	Description
Effectiveness	All instructions of the algorithm should be basic and implementable.

B) Algorithm analysis criteria

①Correctness

The correct result should be produced for valid input by the algorithm within a finite time.

②Amount of work done:

The number of instances of execution undertaken to execute an algorithm; The amount of work done should be measured with important operations only that become the core of the flow in the entire algorithm, excluding general operations included by default.

③Amount of space used:

An amount of computer memory needed to store data and information while an algorithm is being executed.

④Optimality

The expression "A certain algorithm is optimal" means that the algorithm is the most appropriate considering the usage environment (execution times, memory usage, etc.) of the system that will apply the algorithm.

⑤Simplicity

Simplicity means the clarity and ease of understanding of an algorithmic expression. A simple algorithm makes it easy to prove an algorithm's accuracy and to carry out program writing and debugging.

C) Algorithm expression method

<Table 13> Algorithm expression method

Expression method	Description
Description in natural language	An algorithm is expressed using words or texts used in everyday life.
Flow chart representation	An algorithm is expressed using graphical representation, such as a flow chart or NS (Nassi-Shneiderman) chart.
Pseudo code	An algorithm is expressed in a simpler way than the natural language description method by using code written in a form like programming language.

D) Algorithm performance analysis

Algorithm performance is analyzed by estimating space complexity, whereby performance is analyzed in terms of the space required for execution, and by estimating time complexity, whereby performance is analyzed in terms of the time required for execution.

① Space complexity

Space complexity refers to the total amount of storage space or memory needed to execute and complete an algorithm using a program, which can be calculated by adding the amount of fixed space and the amount of variable space.

Space complexity = Amount of fixed space + Amount of variable space

- Amount of fixed space: Storage space that is needed permanently, regardless of the program size or number of input/output times, such as programs, variables, constants, etc.

• Amount of variable space: Storage space that stores data and variables used while executing a program, and information related to function execution.

② Time complexity

This is the time taken to execute and complete an algorithm as a program, which is the sum of the compilation time and the execution time.

Time complexity = Compilation time + Execution time

• Compilation time: Fixed time, which is not closely related to the characteristics of the program. Once compiled, the program is kept constant unless the program is modified.
• Execution time: The program execution time that is calculated by counting the number of instances of command execution, rather than by measuring the actual accurate execution time, as it depends on the performance of the computer.

Execution time is mainly used when comparing algorithms and is expressed as O(n) using the Big Oh notation[1] , which is expressed as time complexity. Execution time functions include logn, n, nlogn, n2, n3, and 2n, depending on the algorithm (by Lee Ji-young). To solve the problem, it is recommended to select an algorithm with the smallest value of time complexity after creating multiple algorithms and calculating the execution time of each algorithm.

<Table 14> Calculating execution times according to the change of n value in the execution time function

logn	<	n	<	nlogn	<	n^2	<	n^3	<	2^n
0		1		0		1		1		2
1		2		2		4		8		4
2		4		8		16		64		16
3		6		24		64		512		256
4		16		64		256		4096		65536
5		32		160		1024		32768		4294967296

<Table 15> Algorithm complexity

Complexity notation	Description
O(1)	Constant type. Finds the solution regardless of the input size.
$O(\log_N)$	Log type. Divides the input data and processes one of them only.
O(N)	Linear type. Processes all input data one by one.
$O(N\log_N)$	Division and merger type. Processes data by dividing and then merging them.
$O(N^2)$	Square type. When the basic operation loop structure is double.
$O(N^3)$	Cubic type. When the basic operation loop structure is triple.
$O(2^n)$	Exponential type. Processes by testing all possible solutions.

1 To use Big Oh notation, find an execution time function by calculating the number of execution times and select a term for n, which affects the value of this function most, and then place it in the right parenthesis of O without the coefficient.

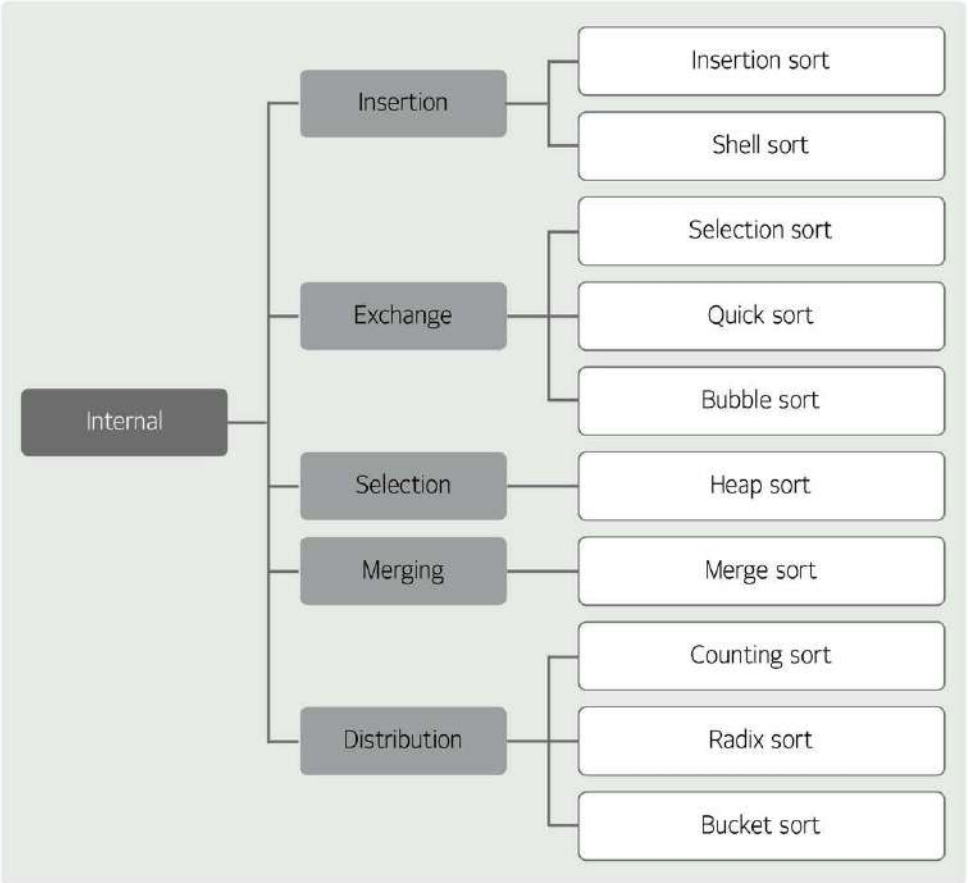
E) Sorting algorithm

① Classification of sorting

Sorting can be classified into internal sorting and external storing depending on the sorting location. The alignment method should be selected in consideration of the conditions, such as the characteristics of the system in use, the amount and state of the data, the memory space required for sorting, and the execution time.

- Internal alignment: A sorting method that places a small quantity of data into the main memory device for sorting. The sorting speed is fast, but the amount of data that can be sorted is limited by the capacity of the main memory device.
- External sorting: A method of sorting a large volume of data in an auxiliary storage device. A large amount of data is divided into several sub-files and sorted, and then the sorted sub-files are merged in the auxiliary storage device. The sorting speed is slow.

② Classification of internal sorting algorithms



[Figure 6] Classification of internal sorting algorithms

③ Comparison of the execution times of internal sorting algorithms

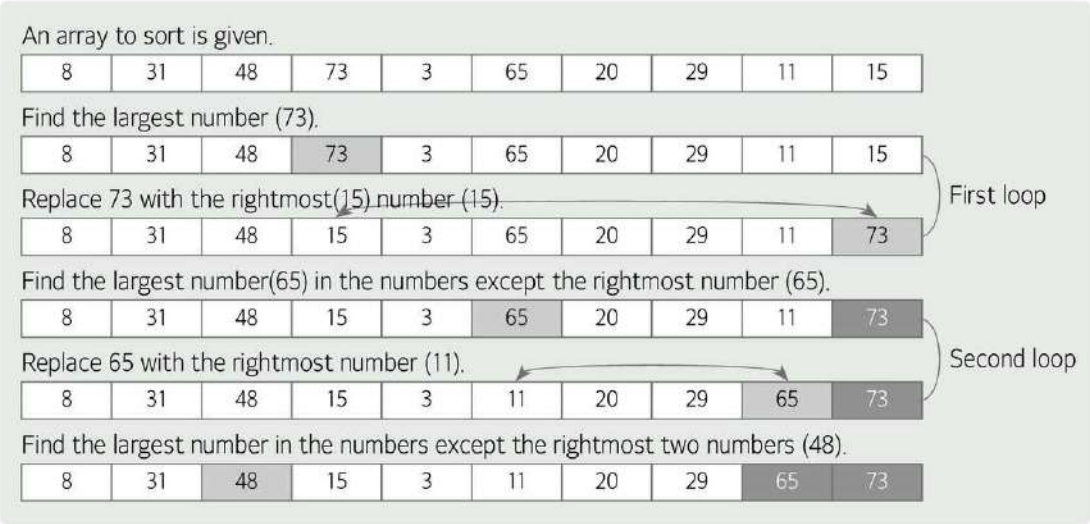
<Table 16> Comparison of the execution times of internal sorting algorithms

Sorting method	Description	Execution time			Additional memory
		Worst	Best	Average	
Insertion sort	- A method of sorting by inserting a value into the position concerned based on the assumption that data has been sorted.	$O(n^2)$	$O(n^2)$	$O(n)$	None
Shell sort	- Dividing the list of given data into subfiles that has the length of a specific parameter value, and performing insertion sorting in each subfile.	$O(n\log_2 n)$	$O(n^{1.5})$	$O(n)$	None
Selection sort	- A method of sorting repetitively as many times as the size (number) of data when finding the minimum value and moving it to the left.	$O(n^2)$	$O(n^2)$	$O(n^2)$	None
Quick sort	A sorting method based on the Divide and Conquer method. First, a random criterion is selected, and values smaller than the criterion are placed on the left and larger values are placed on the right. Then, a random criterion is selected again and values are divided to the right and left again for sorting. - Recursive call is used.	$O(n^2)$	$O(n\log n)$	$O(n\log n)$	None
Bubble sort	- A method of sorting by continuously exchanging adjacent data.	$O(n^2)$	$O(n^2)$	$O(n^2)$	None
Heap sort	- A method of sorting by configuring the maximum heap tree or the minimum heap tree.	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	None
Merge sort	- The Divide and Conquer method is used. The size of data is continuously divided in half and combined again while sorting.	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	Available
Radix sort	- A method of sorting by comparing the low digit of data.	$O(dn)$	$O(dn)$	$O(dn)$	Available

Selection sort is one of the simplest sorting algorithms. The largest element is searched in an array A [1…n] and replaced with the A[n] at the end of the array.

```
selectionSort (A[ ], n)
{
 for last ← n downto 2 {
     The largest number A[k] is
     A[k] ↔ A[last];
     }
}
```

The for loop reduces the size of the array to be sorted one by one. The array size begins as n and becomes n-1 in the next loop, after which the size continues to decrease.

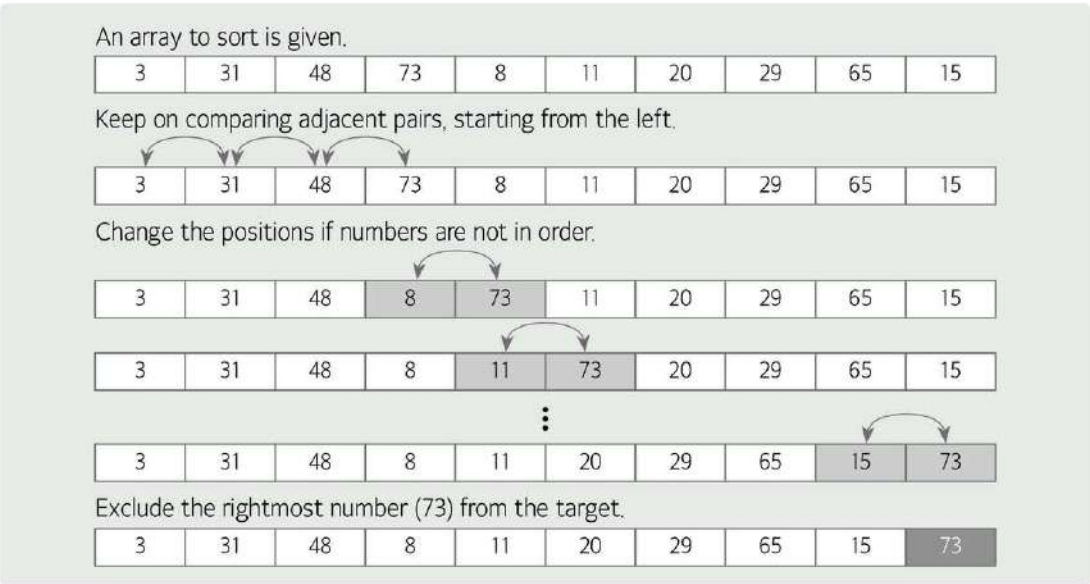


[Figure 7] Example of the selection sort algorithm

④ Bubble sort

Like selection sort, the largest element is continuously moved to the last position. However, the number adjacent to the left is compared when moving the largest element to the right. If the sequence is incorrect, the positions are changed.

```
bubbleSort (A[ ], n)
{
 for last ← n downto 2 {
   for i ← 1 to last -1
     if (A[i] > A[i + 1]) then A[i] ↔ A[i + 1];
}
```



[Figure 8] Example of the bubble sort algorithm

F) Search algorithm

① Search algorithm overview

This is a technique that efficiently finds the desired item in the data set. It can be divided into linear search and control search depending on whether data are sorted or not. There is also hashing, which searches for data by calculating a key value according to a specific function. Therefore, an optimal search method should be selected in consideration of the type of data structure and the data arrangement status.

② Classification of search algorithms

<Table 17> Classification of search algorithms

Classification method	Data sort	Type	Content and characteristics
Linear search	X	Linear search	• A method of finding a key by comparing each record sequentially from start to finish. • Easy to write a program. • Search time increases as much as the file size. • The simplest and most direct search method. • Average number of comparison times: (n+1)/2, average search time: O(n)
Control search	O	Binary search	• A search method that sets the upper limit value (F) and the lower limit value (L), finds the intermediate value (M), and searches by continuously comparing the key and the intermediate value. • Efficient as the number of targets in a file to search is reduced by half each time. • More effective when the number of records is large. (The number of comparisons even in the worst case is just one more than the average number of times.) • Average search time: O(log2n)
		Fibonacci search	• A method of searching by creating sub-files using Fibonacci permutations. • Fibonacci search is fast because it can search by addition and subtraction only, while binary search uses division. • Average search time: O(log2n)
		Interpolation search	• A method of selecting and finding a location where the search target is expected to be present. After finding the location, linear search is performed from that location. • Interpolation search is used to search dictionary, phone book, or index name, etc. • Performance of O(log(n)) on average
		Block search	• A search method that divides entire data into a certain number of blocks, determines a block that contains the data to search, and then sequentially searches for the key value in the selected block. • The effective block size is √n. • Easy to write and update a program. • Performance of O(log(n)) on average
		Binary tree search	• Search method using a binary tree • Performance of O(log(n)) for insert/search/delete on average
A method of approaching data by using a specific function	X	Hashing	• A search method that searches for data by calculating the address where they are stored using the hashing function. • Suitable for data that are frequently inserted and deleted.

G)) Graph search algorithms

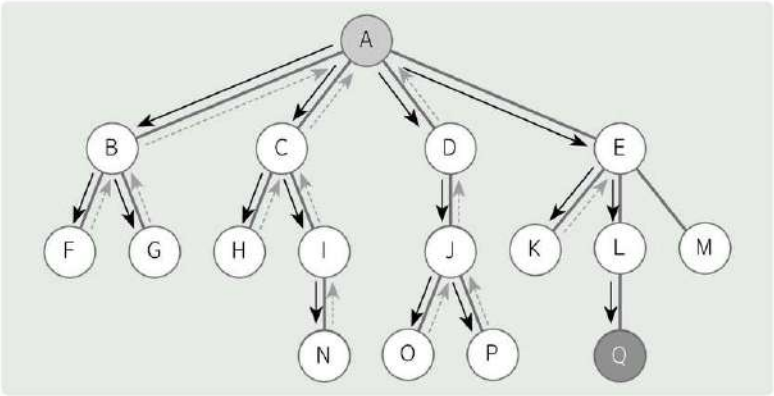
① Graph search

The most basic operation of a graph, which visits and processes all vertices in the graph once, starting from one

vertex. There are many cases in which problems cannot be solved simply by searching for a graph node, such as whether one can travel from a certain city to another city on the road network, or whether a terminal is connected in an electronic circuit. Graph search methods include depth first search (DFS) and breadth first search (BFS).

② Depth First Search (DFS)

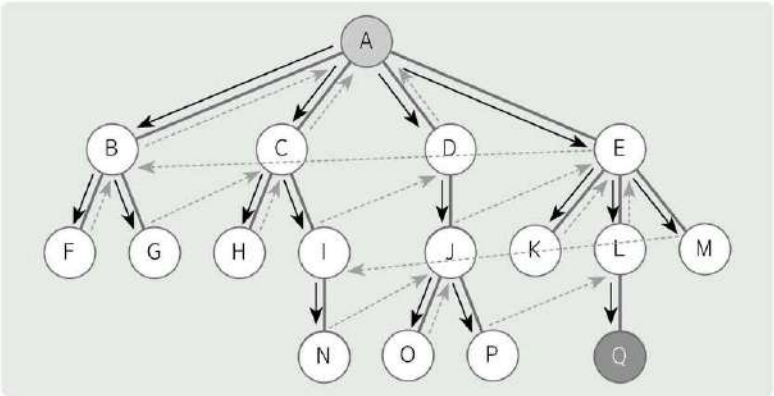
The algorithm starts at the root node and explores in one direction of the starting vertex as far as possible. When there are no more places to go, it backtracks to the vertex with the last forked edge and continues to explore the edge of another direction, visiting the vertices until the target node is found. A stack with a last-in-first-out structure should be used, since deep first search should be repeated by backtracking to the vertex of the last forked road. Even though finding the shortest distance from the start node to the target node cannot be guaranteed initially, the demand for storage space is low because only those nodes on the current path are memorized, and the target node can be found quickly if it is at a deep level. Also, this algorithm is easy to implement.



[Figure 9] Example of depth first search

③ Breadth First Search (BFS)

The adjacent nodes are searched first after visiting one starting vertex. The vertices adjacent to the start vertex are visited first, and the vertices situated farther away are visited later. Breadth first search must test whether a certain node has been visited and uses a queue, which is a data structure that can retrieve visited nodes sequentially.



[Figure 10] Example of breadth first search

H) Minimum spanning tree

① Introduction to the minimum spanning tree

A spanning tree is a special form of tree that includes and connects all vertices in an undirected weighted graph. This tree should not contain a cycle. A minimum spanning tree is a spanning tree whose sum of weighted values is the minimum. Kruskal's algorithm and Prime's algorithm are representative algorithms that implement a minimum spanning tree.

② Kruskal's algorithm

An edge with the smallest weighted value is selected from among the edges connected to the vertex, and it is checked whether the added edge creates a cycle. Edges with a smaller weighted value are selected sequentially, even though the edge with the smallest weighted value is selected but not connected to the vertex.

③ Prim's algorithm

A random vertex is selected from the graph to be analyzed. Then, a new vertex and an edge which have not been visited before are selected for each iterative process and expanded in the minimum spanning tree that has been composed up to that point.

IV. Software Design Principles and Structural Design

▶▶▶ Recent trends and major issues

The world has now entered a new era of limitless quality competition, and software cannot be an exception to the rule. Software design is closely related to software quality. In this respect, we can see that efforts to analyze the requirements are also a process for good design. The cost possibility of software failure can be significantly reduced if quality control and prevention activities are performed frequently in the analysis and design phase.

▶▶▶ Learning objectives

1. To be able to describe the types and contents of software design principles that should be considered when designing software.
2. To be able to explain the concepts of cohesion and coupling, which are the criteria for evaluating the design of a module.
3. To be able to understand structural design methods and express design contents.

▶▶▶ Keywords

· Division, abstraction, information hiding, stepwise refinement, modularization, structuralization
· Cohesion, coupling
· Transform-based design, transaction-based design, structure chart

+ Preview for practical business Ariane 5's explosion attributable to poor software design

Software passes through a design process that consists in abstracting the analyzed requirements into a computer world. The design process should include the method of handling numerous variables defined for the design, the variable types used by an input value, and exceptions due to error values. Software design faults are sometimes eliminated while repeating tests such as boundary value analysis. In addition, redundant data and server configuration to cope with an emergency should also be designed to support the process stably.

The Ariane 5 rocket exploded during its launch on June 4, 1996 due to an inadequate exception design against a mismatch of the variable types and insufficient consideration of redundancy design. The variable types in the design were 16-bit integers, yet 64-bit real type was entered. However, exception handling for integer-type conversion errors was not fully reflected in the design, causing overflow. As a result, altitude and velocity information was not sent to the central control computer.

When a failure occurred in one SRI-2, the redundant SRI-1 should have been activated. However, as the architecture was designed in this way, There could be no effect due to an input value error. Eventually, Ariane 5 deviated from its course and exploded because its engine nozzle angle and thrust could not be controlled.

According to the eventual findings, three exceptions out of seven items were omitted from Ariane 5 to achieve the SRI's maximum load target of 80%. As a result, it is said that software development was forced to bear all the responsibility for the failure. After all, high-quality software design is not guaranteed by the design itself. To complete high-quality software, use cases that ensure the widest possible test range should be reflected in the development and testing process, which in turn will make up for unexpected defects. This process of supplementing the design should be repeated.

01 Principles of Software Design

The complexity of the problem area should be reduced by continuously dividing the user requirements in the software design phase, and its results should be reassembled as proper groups by considering the independence and dependence of the role unit. This task is also expressed as the basic design principle of "Divide and Conquer". Complex problems can be solved more easily if we mentally divide a system into separate pieces. Also, if the functions are divided, or the user interface is logically divided, an easier solution can be found.

In general, a system component divided at a higher level is called a subsystem. The subsystem is a program component that can perform functions and be compiled independently. The subsystem means that a system has been divided at a higher level of the system structure. System designers divide a problem into subsystems so that several designers and designers can develop different subsystems independently later on. The clarity of the split process is pursued to make the entire system work smoothly, and to facilitate the integration of subsystems developed by different developers at a later date.

A) Abstraction

Abstraction means considering product implementation at a higher level first, rather than concerning oneself with implementation at a detailed level, in order to access a problem area gradually while maintaining a larger flow. Abstraction enables us to express only essential matters that are easy to handle, because abstraction eliminates details that are of no interest to us or are not essential. Abstraction is an important principle that is applied throughout the entire engineering process. Engineering is a process of moving from a higher abstraction level to a low level. The types of abstraction can be divided into data abstraction, control abstraction, and process abstraction.

We need to understand how components (or modules) interact with each other and the behavior that manifests itself outside when a system is divided, before understanding how the components are implemented inside in detail. That is, the concept of abstraction is based on focusing on the external interface by daring to omit methods of component implementation.

B) Information hiding

Information hiding is based on the concept that the content of each module is hidden, and that messages can be transmitted by the interface only. It is designed in such a way that one module or subsystem does not affect other modules, by limiting access to internal information. For example, when a change occurs in the design process, information hiding ensures that the change affects the minimum number of modules only. That is, if a module is defined by an interface with the outside, and detailed information including the internal structure or progress of the module is hidden from other modules, the information is hidden. Information hiding keeps the modules independent of each other.

Information hiding does not necessarily require a programming language equipped with an information hiding mechanism. Rather, it is a basic principle for designing software, and the concept of information hiding is important in system design in that it keeps the components independent of each other. Information hiding hides the internal structure of each module (abstraction), and the modules communicate with each other using the predefined interface only, without needing to know the internal structure. If a module needs to be modified,

changes can be limited to the data structure inside the module and the actions needed to access it, so that users can easily adapt themselves to the change and perform maintenance.

C)) Stepwise refinement

Stepwise refinement is realized by gradually moving from the program structure to the module details. The level of abstraction decreases during the refining process, and each function is decomposed to present a solution. Refinement is a process that requires many efforts and enables system implementation by allowing a detailed description.

Design is the stepwise refinement process of moving from a high abstraction level to a low abstraction level. The flow of engineering, ranging from problem description to requirements analysis, design and programming, can also be regarded as a stepwise refinement process. Another example of stepwise refinement is one in which a system is started as a big process and gradually subdivided into small processes that perform detailed functions, using the structural analysis method.

D) Modularization

In most cases, engineering approaches a system by dividing it into components. The component of software is a module. When expressed in a programming language, software modules are called subroutines, procedures, or functions. Generally, a system is subdivided into functions using a top-down approach when modularizing the system. In this case, a control layer appears between the modules.

In addition, modularization enables us to manage systems intelligently and to solve the problem of complexity. In other words, problems are divided (according to the aforementioned principle of "Divide and Conquer") into smaller units (modules) to solve a large and complex problem. Modularization makes system maintenance and modification easier. However, if the number of modules increases, the size of each module decreases proportionately, system performance deteriorates, and overload occurs due to an increase in mutual exchange between modules. Therefore, how to minimize the interference between modules by role and then play the role effectively by focusing on the purpose becomes the important criterion for module evaluation.

E) Structuralization

Software systems can be structuralized by the division process, which is related to the principle of divide and conquest explained earlier. Division is performed first during the requirements analysis process, and detailed division occurs in the design phase.

Analysts have the mission of finding and dividing important system elements or functions, whereas designers have the mission of structuralizing the results of analysis. How to divide a system is not a simple matter, and there are no complete guidelines for this. However, we can make use of experience and the guidelines for existing systems by understanding the characteristics of the system.

Several structural frames can be found from existing systems depending on the characteristics of the system. The amount of time and effort expended can be reduced when making a system with similar characteristics by using such frames.

02 Cohesion and Coupling

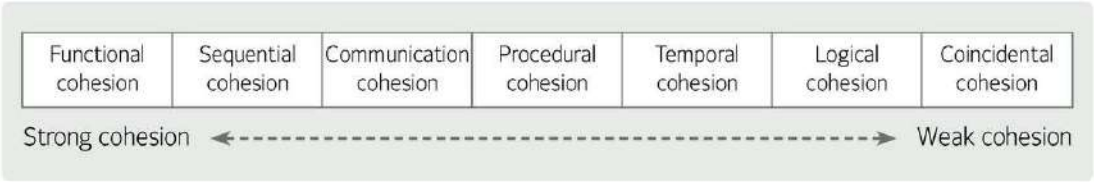
It is not easy to clearly define a good design. A good design can be defined as a design that enables efficient programming or a design that helps easy adaptation to changes so that software evolution programs can be solved well. To be recognized as a good design, design documents - which are design results - should be written in a way that is easy to read and understand, and when changes are made to the system, the effect should be localized.

Maximizing functional independence and reducing the merging of modules are the good design principles that make maintenance easier. The object-oriented design technique can also maximize independence in the end, improving the quality of a design. This section describes cohesion and coupling, which are factors that affect quality in the design phase.

A) Cohesion

Cohesion is a measure of module maturity that indicates the strength of the correlations among the internal modules. That is, it indicates how each of the components in the module is correlated with the others in order to achieve a common purpose. Cohesion indicates how the components of a module fit together, and is like cement that binds the components. Functional cohesion, wherein all elements of the module perform a single function for the upper-level module, is the highest degree of cohesion. Coincidental cohesion, wherein a module is composed of unrelated processing elements, is the lowest degree of cohesion.

Cohesion also indicates the degree to which a module performs a task and thus is another measure of measuring a module's independence of each other. Therefore, it is desirable to design software in such a way that cohesion increases among the internal elements of a module, with each component of a module or system performing a single logical function or representing a single logical entity (model component). In this case, interactions with the outside can generally be minimized. When module cohesion is increased, coupling between modules can be decreased. Conversely, low cohesion can cause high coupling. When designing software, it is desirable for modules to have high cohesion and coupling between modules to be weak. The types of cohesion include functional cohesion, sequential cohesion, communication cohesion, procedural cohesion, temporal cohesion, logical cohesion, and coincidental cohesion.



[Figure 11] Type and degree of cohesion

① Functional cohesion

- When all functional elements inside a module are performed in connection with a single problem.

② Sequential cohesion

- When the output data from one activity in a module are used as the input data of the next activity.

③ Communication cohesion

- When components performing different functions while using the same input and output are gathered.

④ Procedural cohesion

- When a module has multiple related functions, and the components of the module perform those functions sequentially.

⑤ Temporal cohesion

- When creating a module by combing several functions that are processed at a specific time.

⑥ Logical cohesion

- When a module is created using processing elements that have similar characteristics or are classified in a specific form.

⑦ Coincidental cohesion

- When each component inside a module is composed of unrelated elements only.

B) Coupling

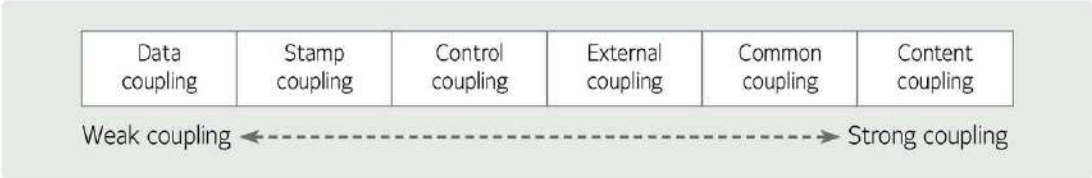
Modules have an associative relationship with other modules. "Coupling" refers to the complexity of correlations between modules. Coupling between the modules increases when modules interact more and depend on each other more. In particular, when the interface is not set correctly or the functions are not correctly divided, an unnecessary interface appears and increases the degree of dependence between modules and coupling.

Coupling is a method of indicating a correlation between program elements and is closely related to module independence and cohesion. If two modules function perfectly, whether they are adjacent to each other or not, they are completely independent of each other and do not interact at all. Data coupling in which modules communicate with parameters is the most ideal - and weakest - coupling. Content coupling in which a module directly refers to or modifies another module's internal function or data is the strongest coupling, and should be avoided.

If coupling is strong, it is difficult to change a module independently, and changes in one module affect other modules significantly, causing a ripple effect. The greater the ripple effect, the more difficult it is to maintain the system. The ripple effect is a phenomenon in which, when one thing is affected, it goes on to affect many other things. If this effect occurs in system development, it lands many people in trouble.

Therefore, coupling needs to be reduced as much as possible when designing software. When modules share variables or exchange control information, coupling between modules increases. Because of this (to weaken coupling), we are instructed not to use global variables, if possible, and to use parameters when learning

programming. Types of coupling include data coupling, stamp coupling, control coupling, external coupling, common coupling, and content coupling.



[Figure 12] Type and degree of coupling

① Data coupling

- Data coupling refers to coupling when the interface between modules is composed of data elements only.
- A module passes data as a parameter or argument while calling another module, whereupon the called module returns the processing result of the received data.
- Data coupling does not have to know the contents between modules at all. It is the most desirable coupling, as changes in one module do not affect another module at all.

② Stamp coupling

- Stamp coupling refers to coupling in which a data structure such as an array or record is passed over to the interface between modules.
- In stamp coupling, two modules retrieve the same data structure. If the data structure is changed, that is, if the format or structure is changed, all modules that do retrieve it, along with the module that does not actually retrieve the changing field, are affected.

③ Control Coupling

- Control coupling refers to coupling in which a control element (function code, switch, tag, flag) used to control a logical flow is transferred from one module to another.
- It occurs when the parent module knows and controls the detailed processing procedure of the child module or when the processing functions are split into two modules in the design.

④ External coupling

- External coupling refers to coupling in which data (variables) declared externally by one module are referenced by another module.
- Each module can limit the scope of referenced data.

⑤ Common coupling

- Common coupling refers to coupling in which several modules share a common data area.
- Common coupling weakens modules' independence because any changes to the common data area affect all of the modules that use the common data.

⑥ Content coupling

- Content coupling refers to coupling in which one module directly refers to or modifies the internal function or internal data of another module.
- When one module is branched into the middle of another module, it is also content coupling.

03 Structural Design Method

The process of transferring the results of structured analysis, a functional modeling technique, to structured design is called data flow-oriented design. When a structural method is applied, a system is logically divided and modularized at an early stage of design, and top-down refinement is performed by subdividing the system from top to bottom.

The notation used in the structured analysis process to represent the data flow is called a "bubble chart" or a "data flow diagram" (DFD). The DFD used in the structured analysis technique describes the data flow and function from a logical point of view, whereas the structure chart used in structured design represents the structure and design of software in detail.

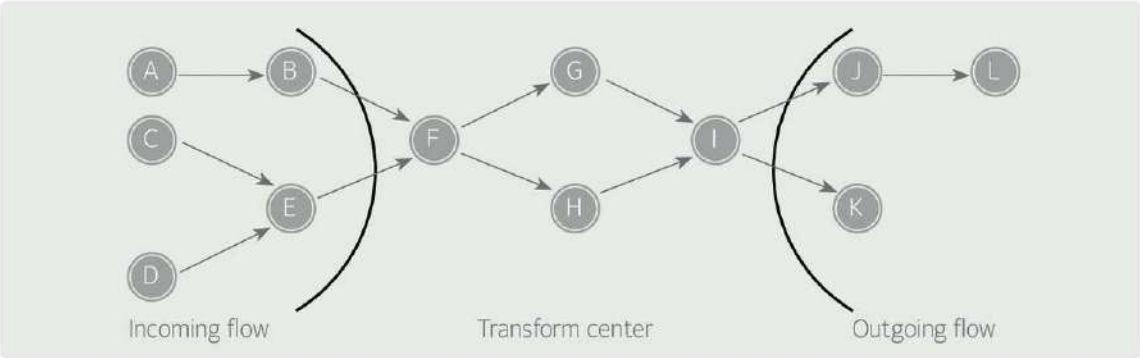
The structure diagram is a notation frequently used to represent the structure of a program. Actually, the structure diagram also includes other contents, in addition to the representation of the program structure. The structure diagram shows the flow of data and control between modules, and represents aspects of the program control structure - such as iteration and selection.

The structure diagram shows the exchange of data and control information between modules, major loops, and decision making, and is a model that represents the contents of interaction between the module system and the modules at the design stage. The data flow and the control flow can be distinguished in the structure diagram as follows. Data are used in calculations to create new data or modify them. However, controls are not used directly in calculations, but instead represent the sequence or condition of calculations. The flag that is commonly used in programming has the value of "on" or "off", and is a control signal used to imply condition occurrence or demarcate the boundary.

Structured design presents guidelines for converting a requirements specification into a design document. The conversion is determined by the type of information flow in the system. The information flow can be largely divided into two types, transform-based and transaction-based information flow. A design document can be created using these two types of characteristics that are present in the requirements specification while moving from analysis to design.

A) Transform flow-oriented design

Transform flow-oriented design is a technique that maps the system, which receives and processes information and outputs the result to the outside world and the corresponding computer structure. This technique divides a system into components and creates a hierarchy between the modules that perform the basic functions. Transform flow-oriented design can divide a system into the following three parts: The first part receives input and refines it as data that can be used by the system. The second part executes the data processing function. Most of the processes that perform calculations belong to this category. The third part receives the processed information and converts and outputs it as a proper output. A system is largely divided into three parts in this way. A set of processes that manages input is called an "incoming flow", a set of processes that handles the input information is called the "transform center", and a set of processes that outputs the processed information is called an "outgoing flow".

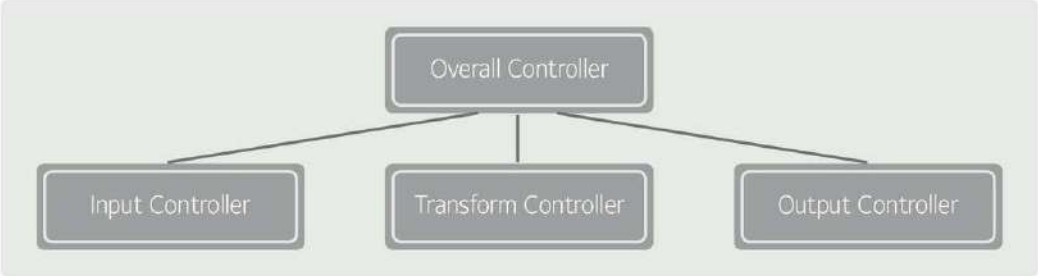


[Figure 13] Input and output boundary of the data flow diagram

Now, it's time to start creating a program structure centered on the data flow. The guidelines needed to create a transform-based program structure are based on identifying the input flow, transform center, and output flow of the data flow diagram. Mapping the structure table in the data flow diagram is performed by the guidelines. Three components can be found if the root program structure is created in the transform flow-oriented data flow diagram. The three components are as follows.

- Input control module, which processes input.
- Transform control module, which processes transform.
- Output control module, which processes output.

The input control module receives input data from the lower-level modules and sends them to the upper-level module. If necessary, the input data are refined and transmitted to the upper level. The output control module receives output data from the upper level and sends them to the lower-level modules. If necessary, this module also refines the output flow and transmits it to the lower-level module.
The number of control modules is subject to system complexity, and the number of control modules that process transform is determined by transform centered complexity. The figure below shows a simple program structure based on transform.



[Figure 14] High-level program structure based on transform flow

TOPCIT ESSENCE

V. Software Architecture Design

▶▶▶ Recent trends and major issues

Recently, software architecture has been highlighted as having a fundamental and useful key role in effectively reflecting changes in information technology and developing high-quality software most effectively in a timely manner. Architecture design is the process of identifying the composition of a system and how its components interact with each other at the beginning of the design process. This phase is required to manage the complexity of the system.

▶▶▶ Learning objectives

1. To be able to list the basic concept and components of software architecture (SA).
2. To be able to explain the representative types of software architecture.
3. To be able to explain the methods of expressing software architecture design.

▶▶▶ Keywords

- Module, component and connector, subsystem, framework
- Repository structure, MVC (Model-View-Controller) structure, client-server model, hierarchical structure, pipe filter structure
- Context model, component diagram, package diagram, deployment diagram

✚ Preview for practical business Roles of the software architect

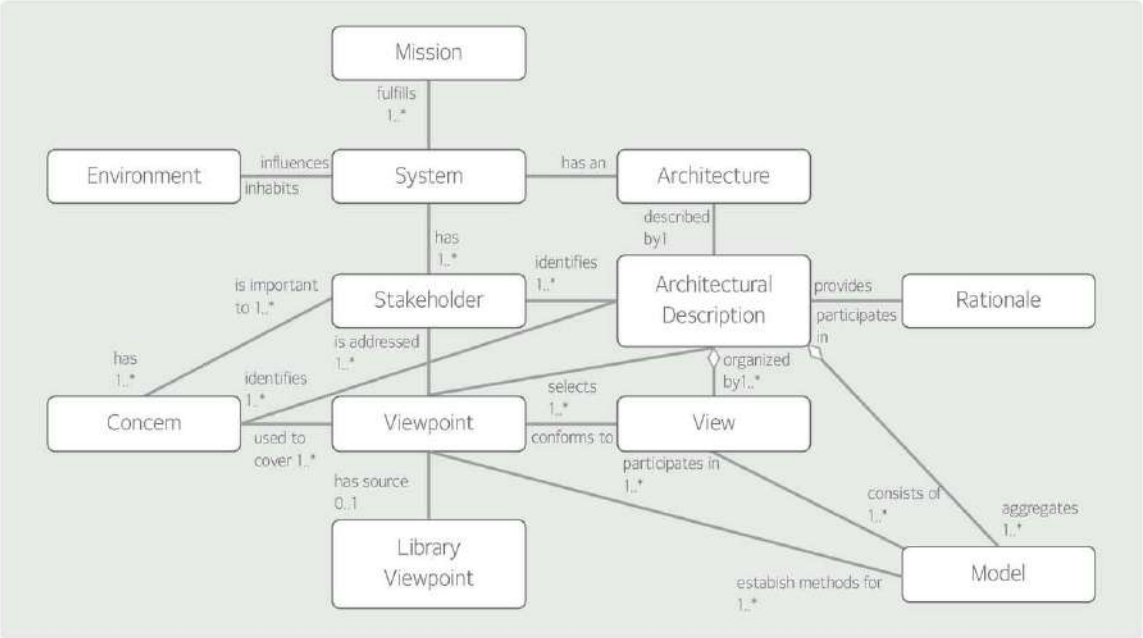
- Creator of a vision - The requirements and constraints should be well understood and understanding of the realized technologies should be improved, based on a professional perspective and experience in the type of product to be made. A software architect plays the role of completing the framework of the software to be developed. The architect's creativity is important, and the architect should be able to present what has been arranged.
- Key technical consultant - A key technical consultant advises the development organization regarding technical matters, based on his or her expertise. It is particularly important to properly select a design pattern and a development framework because it is the first step in ensuring that the quality of a software product is commensurate with its characteristics. To this end, continuous knowledge realization and competency development are required.
- Decision maker - Ultimately, an architect leads a design team in the development of a software product and makes decisions about major areas or factors that affect the overall design. To this end, increasing knowledge about the domain in which the product will be used or applied is also defined as an important factor.
  Recently, this role has become more important, given that a proper decision appropriate to the situation must be made quickly and evaluated, such as agile development, etc.
- Coaches - Similar to the role of the key technical consultant (described as the second role), the coach trains the defined architecture and manages its improvement and feedback. The role of the coach, as a technical leader, and coaching is exercised by various sponsorships and its position is also strengthened.
- Coordinates - The different opinions of the various project participants should be coordinated and mediated to maintain and ensure design integrity. A person who can mediate from the technical standpoint is required because these factors may develop into an important risk factor. An architect can be seen as an authoritative technical mediator in this regard. Therefore, proper communication skills are also important, in addition to technical competence.
- Implements - When introducing a new technology, its impact should be evaluated from the design stage, and the feasibility of its implementation should also be evaluated using a prototype. Generally, developers grow into architects by improving their skills and experience.
- Advocates - New architecture should be continuously evaluated and a ground for introducing and continuous investment should be provided. Technical skills and architecture should be advanced continuously through this.

(Source: Applied Software Architecture)

01 Software Architecture Design

A) Software architecture overview

Software architecture can be regarded as a blueprint which is used to systematically handle various factors that directly or indirectly affect the development of software and increase complexity. The definition of software architecture can be expressed in various ways. Booch defined architecture as "a set of important decision-making rules about the software structure". Myron Ahn[4] said that we can judge various things from software architecture, such as modules, processes, data, and their structure; the relationships between components; the method of extending and modifying these components and relationships; the components of the technology in use; and the method of implementing and modifying the flexibility and performance of the system. When starting a project, software architecture is used as a means of communication and a decision-making tool, and is also considered when determining the overall system structure and organization of a development project.



[Figure 17] Components of software architecture (IEEE-1471)

B) Software architecture design procedure

Software architecture is designed by the procedure of requirements analysis, architecture analysis and design, and architecture verification and approval.

Requirements are specifically identified through request for proposals, interviews, and meetings, and functional and non-functional requirements are classified and specified. During the architecture analysis procedure, quality factors should be identified and prioritized. Then, architectural styles and candidate architectures are identified and processed when designing architecture. Architecture selected in this way is evaluated and detailed, and

then submitted for final approval.

The structure of a system should be set in such a way that the user's requirements can be satisfied in the early stages of design. It is desirable to divide the system in order to solve complex problems. A complex problem can be solved easily if the system is analyzed after dividing it. An easier solution can be found if functions are divided or the user interface is divided logically.

System components split from the top level are generally called subsystems. Subsystems are the program components that generally include data and the control structure, and can perform functions independently and can be compiled. In addition, subsystems are distinguished by the service or function they provide.

A framework is a support unit that collects knowledge, which can be reflected in architectural design repetitively, when designing a subsystem. The framework represents a general structure that can be extended to create a concrete subsystem, and increases the level of design abstraction by providing concrete implementation methods.

Architecture design is a process of setting up a system configuration to satisfy the system requirements. The relationship between each component (module) in the program structure can be described using the results of the requirements analysis process. The main purpose of structure design is to develop a modular program structure, and to represent the control and interface between the modules.

02 Software Architecture Style

The following section describes the representative types of architecture found during software development.

A) Repository structure

When one subsystem creates data and other subsystems use that data, the repository is a structure that enables all subsystems to share the data by storing all shareable data in one place. The structure of the shared repository is suitable for sharing a large amount of data.

B) MVC (Model - View - Controller) structure

The MVC structure is a framework that is frequently used for GUI design. It is an approach in which several expressions of one object can interact with each other. Therefore, when the expression of one object is modified, all other expressions are updated accordingly. Modifications can be simplified, and reuse can be made easy using this method.

C) Client-server model

The client-server model is composed of a set of servers and clients. The model is composed of a client requesting a service and a server providing the service, and there may be multiple client instances. Generally, the client-server model can utilize the network system effectively as it is implemented as a client-server model.

D) Hierarchy

A hierarchy can be defined as a system composed of several layers that provide its own specific service. It has the advantage of making problem solving easier when applied. If a problem occurs, it can be checked in an easy layer. Also, it is compatible with other equipment because the equipment is standardized. The seven layers of the OSI (Open System Interconnection) model, a network protocol developed by the International Standards Organization (ISO), is a representative example of the hierarchical structure.

03 Method of Expressing Software Architecture Design

A) Context model

The boundary between the initial requirements analysis system and the external environment should be set. The context diagram represents a system as a large process before dividing the system. This model shows system IO data by describing the system area that should be developed, determining the boundary between the system and the external environment, and presenting the interface with the outside.

This model focuses on the interface between the system and the external environment. When beginning an analysis, analysts should focus on understanding the data exchanged between the user and the system, as well as the type of mutual exchange, first. When the boundary of the system is set, the inside of the system is analyzed to realize it. It is the same as giving priority to goals in developing a system.

B) Component diagram

A component is a part that can be reused, so that a well-made part can be purchased from other fields and plugged in, to speed up software development and increase productivity. In essence, reusability is the foundation of this technology. Reuse technology is designed to speed up software development and increase productivity by reusing proven parts as much as possible.

Components should communicate and cooperate with other systems or external devices. The standard for component implementation and documentation should be established to guarantee the interoperability of the components. Here, "components association" is a process of assembling components. There are several types of association including sequential association, hierarchical association, and additional association.

C) Package diagram

Commercial software developed for a large number of users is called a "package", and subsystems are often released as packages. Once defined as a package, the relationship of dependence between packages can be minimized by hiding the inner details of the package from the outside. The package composed of a subsystem is a set of functionally related classes.

The package diagram represents the relationship of dependence between subsystems. It is suitable for representing software architecture because it represents a subsystem that is abstracted at a higher level. If subsystems are divided to minimize the association relationship between subsystems, dependence between objects can be minimized and complexity can be reduced.

VI. Object-Oriented Design

▶▶▶ Learning objectives

1. To be able to understand the concept of object-oriented analysis and modeling.
2. To be able to explain the concept and principles of object-oriented design.
3. To be able to perform static and dynamic modeling and express it in UML (Unified Modeling Language).
4. To be able to list the concepts of the design pattern and representative patterns.

▶▶▶ Keywords

- Use case, sequence diagram, activity diagram
- Object, class, encapsulation, inheritance, polymorphism, association, set
- Class, property, relationship, association, operation, class diagram
- Interaction diagram (sequence diagram, communication diagram), state diagram, activity diagram
- Singleton pattern, factory method pattern, façade pattern, strategy pattern

+ Preview for practical business SOLID - Five principles of object-oriented design

- SRP (the Single Responsibility Principle)
  As one class has only one method, there should be only one reason for modification.
  However, using a class becomes increasingly difficult as the problems become more complex and enormous in an actual environment, so it is better to apply it in the form of a utility.

- OCP (Open Closed Principle)
  A class should be closed to modifications and open to extensions. It is related to polymorphism and abstraction, which are the representative characteristics of object-oriented programming. When improving functions, it is desirable to improve inheritance rather than direct modification. In addition, abstract classes should be implemented in consideration of changes in logic.

- LSP (Liskov Substitution Principle)
  This is the most important principle among the design for inheritance. This principle should be followed because the IS-A relationship is established when a class is inherited. That is, it is related to inheritance which the superclass can always refer to the subclass.

- ISP (Interface Segregation Principle)
  Communication between objects is established using the interface of the exposed object. This principle is related to the exposing method. It should be designed in such a way that a different type of interface is provided for each user. As an example, we can consider that the interface for general users and administrators should be different.

- DIP (Dependency Inversion Principle)
  When there is a using class and a used class, and the interface of the used class is changed, there is a dependence which the change to the using class should also be checked. Even though a slight change occurs, the reality is that it is difficult to guarantee its integrity and integrity. Therefore, this principle stipulates that the design should be made as flexible as possible through abstraction using the interface, so as to minimize the possibility of such changes.

01 Object-Oriented Analysis and the Modeling Concept

Object orientation takes a given problem area as a set of objects that are present in the real world and represents the interaction between those objects. This can be understood as a virtualization process that moves the problem area to the computer world in the same way as viewing the real world. It has the advantage of increasing the reusability of an object and deepening the understanding of the participants. The object-oriented analysis technique identifies an object required by software by applying three viewpoints (information, dynamic, and functional viewpoints), and finding the properties and behavior of the object. The object-oriented development method has the advantage of applying the same methodology and expression technique to the entire software development process, ranging from analysis to design and programming.

① What is modeling?

Modeling is the process of simply diagraming the performance or operation process of the target system in order to analyze it, or expressing the characteristics of the target system mathematically. We have to express the real world to understand it, and modeling enables us to express the real world.

We should be able to provide software views from various perspectives and understand the requirements of software. Modeling results become the key part of the requirements specification and provide the information needed to carry out the next phase of the project. Modeling results are used as a tool for dialogue between users and developers, and are of great help in identifying the requirements needed when starting a project and in understanding the system outline and frame that are needed in the development stage (including design, implementation, and testing).

② Three viewpoints of modeling

The use of software varies depending on the view. The view of software can be expressed in the following three perspectives.

<Table 18> Three viewpoints of modeling

View	Contents
Functional viewpoint	The functional model describes a system from the viewpoint of the functions performed by software. The functional model is a viewpoint about the output of the given input, and describes the related operations and constraints.
Dynamic viewpoint	The dynamic viewpoint describes the state of the system and the causes of changes (event, time, etc.) in its state by focusing on the operation and control of software.
Information viewpoint	The information viewpoint is used to understand the static information structure of software. It finds an information object used by the system and clarifies the characteristics of that object as well as the relations and associations between objects.

③ Use case

If the use case technique, which is well-known in object-oriented analysis, is used, communication between customers and system developers can be smooth and customer requirements can be effectively understood. Use cases can provide many benefits to the various stakeholders of a given project, including the customer,

project manager, developers, designers, and so forth. By understanding the customer requirements quickly, customers can be induced to participate actively using the use case technique, the functional requirements of the system can be determined at the beginning of the project, and the results can be documented.

The use case technique identifies stakeholders, classifies them into homogeneous groups according to their roles, and classifies them into an actor. Each actor has a different view and use of the system. A use case, which is the use of the system by an individual actor, can be identified based on this. A use case corresponds to a "use example" that shows for what use (purpose) each actor uses the system. Each use case is a set of scenarios used by a particular actor to accomplish a certain function or task.

Scenarios should be created for each use case. A scenario represents the flow and process of an event, and can include information exchanged between systems and actors, and the situation, environment, and background, etc. in which interactions occur. The use case technique is used to verify user requirements because it represents the interaction between the user and the system and can clearly express what the system performs.

④ Information modeling

Information exchanged between actor and system can be found using the use case scenario. Using the information, information modeling should be performed to identify the information that should be saved and managed inside the system, and the result should be displayed as a class diagram of UML (Unified Modeling Language). By using information modeling, the basic classes required in the system can be identified, and the correlation between classes can be found from the relationships between classes, and the properties of each class can be found. Generally, a class diagram can be obtained in this phase, which expresses classes, properties inside the classes, and the relationships between the classes only.

⑤ Dynamic modeling

In the previous section, the properties and relationships of the classes, which constitute the system, were displayed in the class diagram by performing information modeling. Based on this, the method of performing dynamic analysis can be described as follows. Dynamic analysis is the process of finding the operations of classes by paying attention to changes to the state or operations of the objects constituting the system, or the interactions between objects.

UML identifies interactions between objects using the sequence diagram, and then identifies the operation of each class based on the identified interactions. It is common to create a sequence diagram for each use case. Use case scenarios are created by regarding a system as a black box, whereas sequence diagrams further expand use case scenarios and represent the process of interaction between objects inside the system.

⑥ Functional modeling

Sometimes, various functions should be performed to carry out an event (operation) identified by a sequence diagram. Internally, these functions contain complex logic, and can be expressed as operations in a function on a smaller scale. The activity diagram can identify a potential or new activity by expressing the various logics performed within an operation as an activity. The activity diagram is used to accurately understand the process of handling events in a class. The diagram is used to understand the complex process handling procedure or to identify the additional operations of a class.

02 Object-Oriented Design and Principles

Requirements analysis is performed by creating from a class diagram and a sequence diagram to an activity diagram during the process of object-oriented analysis. In general, requirements analysis is described at the conceptual level from the customer's viewpoint, while the physical aspect of how it should be stored is not described.

Emphasis is placed on how to represent the user requirements - previously identified during the object-oriented analysis process - in the software in the object-oriented design phase. Details about implementation are suggested in the design phase to determine a specific method of physically realizing the requirements. Object-oriented design has the advantage of enabling design without the need for a large-scale conversion process by reusing the notation used in object-oriented analysis.

A) Object and class

A class is a collection of similar objects. An object represents a thing that exists independently in the real world, for example, Mr. Kim Cheolsu (example of an employee) or Seoul (example of a place). Each object is described by a group of specific properties and is distinguished from other objects by property values. For example, Kim Cheolsu has such properties and property values as his social security number "830425-1XXXXXX" and his date of birth "April 25, 1983".

There are sets within a given system that naturally have similar properties. For example, there are many students in a college, and these students have similar information. They share the same properties (e.g. student number, major, grade, etc.), and each student (object) has a unique value for each property. At this time, a set of objects having the same property is called a class. That is, an instance of a class is an object. For example, "Kim Cheolsu" and "Lee Younghee", as objects that belong to the student class, share the same properties (student number, major, grade), but "Kim Cheolsu" has unique property values such as his student number "925462", his major "CS", and his grade "3.8", while "Younghee Lee" also has his own unique property values, such as his student number "956234", major "Mathematics", and grade "4.2".

When investigating the characteristics of data required by the system, it is more convenient to describe a grouped class than it is each object. The task of grouping similar objects is called classification, and the objects grouped through classification share the same types of properties, constraints, and behaviors.

B) Encapsulation

The goal of design is to create software that is easy to understand and modify: and in this regard designing modules with a high level of independence is a starting point. Software looks mature when the constituent elements perform their respective functions independently. The functional independence of software components is a byproduct of the modularization process and the aforementioned concept of information hiding. The functional independence of a module can be maximized when the unit modules are processed more completely and when the processing dependency with other modules is minimized.

Encapsulation, one of the important concepts of the object-oriented development method, is both a method of obtaining improved abstraction and independence through information hiding, and a method of protecting properties and operations together as a group. Object-oriented languages have a function that supports encapsulation.

C) Inheritance

Each class has its own properties and operations, as well as properties and operations common to several other classes too. When there are similarities between classes, these similarities can be collected to define a new class, which is called generalization. For example, the classes "professor" and "student" have common properties (e.g. resident registration number, name, address, phone number). At this time, a generalized class, "people", can be defined by combining professors and students. In this case, "student" and "professor" are called a subclass of "people", and "people" is the superclass of student and professor.

In this case, the superclass "people" has the common properties and operations of the subclasses "student" and "professor", and the subclasses "student" and "professor" have properties and operations that are not shared between subclasses. Generalization shows important characteristics wherein the superclass displays the common properties and operations of certain subclasses, and superclass information is inherited by subclasses. Inheritance through generalization enables us to simplify the definition of a class and create a new class by using the pre-defined class.

D) Polymorphism

One of the characteristics of object-oriented programming is polymorphism. Polymorphism means that operations with the same name are performed differently depending on the class, and means that one function name and one operator can be used for multiple purposes. Polymorphism in object-oriented programming is mainly used in inheritance relations and provides a degree of flexibility that allows subclasses to respond to one operation defined in the superclass using their own unique method. Polymorphism is based on a concept that allows the superclass to call the method (a series of task sequences to be executed according to the message) of the subclass. It is also classified into "overriding", which redefines the method defined by the superclass in the subclass, and "overloading", which defines multiple methods using different types of parameter and different numbers of parameters.

The sender of the message simply calls the operation of the superclass only and does not have to know the type of object (subclass), and the proper operation of the subclass is automatically determined according to the type at runtime. Determining the operation using the object of the subclass at runtime is called "run-time binding". Polymorphism and run-time binding are essential concepts for understanding design patterns.

03 Static Modeling and Dynamic Modeling

A) Static modeling

Static modeling is based on the same concept as the information modeling described above. It is designed to reveal the static information of an object without the intervention of the concept of time. Information modeling describes the data by finding out the structure of the database used in the system. The information model finds the basic objects required by the system, and is composed of relationships that indicate the properties of the objects and the relationships between those objects.

The class diagram of UML is an information model that shows the static information structure of a system. It is used to represent the classes required for the system and the relationships between them. Each class consists

of various properties and operations that represent the properties of the object concerned. The documents created in the preceding phase can be used to find a class, its properties, and the relationships between classes. Problem description documents or use case scenarios can be used to identify a class.

B) Dynamic modeling

The classes constituting a system can be found and the properties and relationships of the classes can be identified using the static modeling described above. Now let's learn how to perform dynamic modeling based on the results of static analysis. While static modeling literally focuses on the static structure of a system, dynamic modeling is a process of finding the operations of classes by paying attention to changes in the state or behavior of the objects that constitute a system, or the interactions between objects.

Dynamic modeling identifies class operations based on interactions between objects. Class operations are defined to perform the functions requested by the messages of other objects. In general, a UML sequence diagram is used to represent the interactions between objects. The sequence diagram emphasizes the transfer process of the messages exchanged between objects over time, while the collaboration diagram expresses the cooperative relations and exchanges of message between objects. These two diagrams are also called an interaction diagram.

The activity diagram is used to accurately understand the process of handling events in a class. It is used to understand the procedure for handling complex processes or to identify the additional operations of a class. This activity diagram can help to understand the activities occurring in each use case and the dependencies between them. In addition, if the operation of a specific object is composed of a complex structure internally, it can be represented using an activity diagram.

The table below divides the UML diagram into the functional model, static model, and dynamic model.

<Table 19> UML diagram>

Model type	Diagram	Contents
Functional model	Use case diagram	Structurally expresses various cases of using actors and systems, that is, the relationship between use cases.
Static model	Class diagram	Expresses a structural relationship between the classes and interfaces that constitute the system.
	Object diagram	Expresses the structural state of objects at a specific point in time.
	Component diagram	Expresses the relationship between component structures.
	Deployment diagram	Expresses the physical structure of the execution system including software, hardware, and network.
Dynamic model	Sequence diagram	Expresses the dynamic messages exchanged between objects inside the system in order to handle external system events over time.
	Collaboration diagram	Expresses the same contents as the sequence diagram from the viewpoint of the interrelation between objects.
	Activity diagram	Expresses the flow of activities inside the system.
	Statechart diagram	Expresses state transitions inside the system.
	Package diagram	Expresses the relationship between packages after creating a package by grouping several model elements including classes or use cases.

04 Design pattern

A) Concept of the design pattern

The software design pattern can be likened to the patterns used to make clothes. The design pattern (omitting software) can be defined as "a general, reusable solution to a recurring problem in a specific context of software design." The design pattern is a description or a sample of a problem-solving method that can be used in various situations. It formulates the best practices that programmers should implement, and generally represents the interactions or relations between classes and objects.

Classification by purpose →

Classification by scope ↓

	Creation	Structure	Behavior
Class	- Factory Method	- Adaptor(class)	- Interpreter - Template Method
Object	- Abstract Factory - Builder - Prototype - Singleton	- Adoptor(object) - Bridge - Composite - Decorator - Façade - Flyweight - Proxy	- Chain of Responsibility - Command - Interpreter - Mediator - Memento - Observer - State - Strategy - Visitor

[Figure 18] Classification of design patterns

Design patterns can be classified as shown in the figure above according to their purpose and scope. Design patterns can be divided into the creation pattern, structural pattern, or behavioral pattern depending on the purpose.

① Creation pattern: A pattern involved in the process of object creation.

② Structural pattern: A pattern related to the composition of classes or objects.

③ Behavioral pattern: A pattern that characterizes the ways in which classes or objects interact and distribute responsibility.

Design patterns can also be classified according to their scope. The scope indicates whether the pattern is mainly applied to classes or objects.

④ Class pattern: A pattern that handles the correlation between classes and subclasses. The correlation is mainly inheritance, and is statically determined at compile time.

⑤ Object pattern: A pattern that handles the object correlation and can be changed at the runtime. It has dynamic characteristics.

B) Representative design patterns

Software can be effectively developed using the design patterns shown below, in order to improve the

convenience of reusing repetitive codes and to manage complex codes more easily.

<Table 20> Representative design patterns and problem types

Large category	Problem type (purpose of use)	Related design pattern
Patters for object creation	Creating an object by product family	Abstract Factory
	Creating a whole object by creating partial objects	Builder
	Creating an object using a delegate function	Factory Method
	Creating an object through replication	Prototype
	Limiting object creation up to N	Singleton
Patterns for structural improvement	Changing an interface for reusing the existing module	Adapter
	Clear separation of interface and implementation	Bridge
	Establishing and managing the partial-whole relationship between objects	Composite
	Dynamically adding and deleting an object function	Decorator
	Clearly separating and defining a subsystem	Façade
	Sharing small objects	Flyweight
	Performing tasks using delegate objects	Proxy
Patterns for behavior improvement	Spreading requests to executable objects	Chain of Responsibility
	Manipulation by generalizing the task to perform	Command
	Verification and work processing based on simple grammar	Interpreter
	Sequentially accessing several objects of the same data type	Iterator
	Simplifying a M:N object relationship to N:1	Mediator
	Restoring or storing the previous state of an object	Memento
	One source multiple use	Observer
	Smoothly changing behavior execution when adding an object state	State
	Selecting and applying one of several algorithms with the same purpose	Strategy
	Reusing the basic frame of the algorithm and changing detailed implementation	Template method
	Efficiently adding or changing a task type	Visitor

① Object creation patterns

• Abstract Factory pattern

Definition: When creating a family of objects that are either closely related or hardly related, an interface is provided that can create the object concerned without specifically knowing its class.

- Advantages: Applied when creating a specific object and actually using it without specifically knowing the type of object to which the client codes belong.
  A new product family can be added independently of the existing source codes when creating a new product family.
- Disadvantages: All factory classes should be modified when adding a new product to the product family.
- Use examples: When developing a compiler that should be configured differently for each system or operating system.

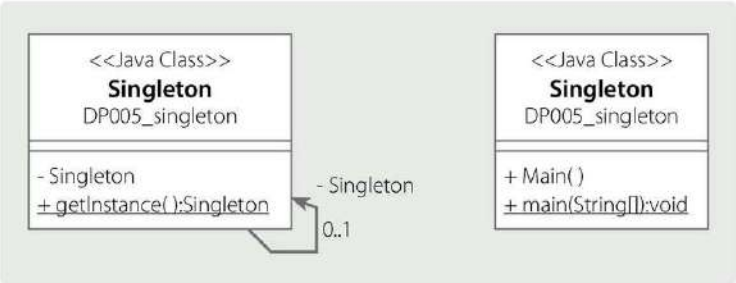
• Builder pattern
  - Definition: Used when the returned objects are not the simple inheritance target of the basic output object, but are completely different user interfaces composed of different combinations of objects.
  - Advantages: An object having different expression methods can be created in the same way.
  - Disadvantage: It is easy to create a new object, but it is very tricky to modify each component of the object.
  - Use examples: When developing a translator system that translates entered Korean into various languages and outputs the result.

• Factory Method pattern
  - Definition: A method of indirectly creating an object using a proxy function, rather than directly calling the object constructor.
  - Advantages: Programming is possible using the same form, regardless of the object to be created. Therefore, it is flexible and scalable.
  - Disadvantages: A new subclass must be defined each time the type of object to be created changes.
  - Use examples: When a program is double clicked in the OS, the program is operated by the OS.

• Prototype pattern
  - Definition: A method of creating an object by duplicating an existing object.
  - Advantages: There is no need to create a separate class to create an object. An object can be added and deleted at runtime.
  - Disadvantage: All classes, which are the data types of objects to be created, should implement the Clone() member function.
  - Use examples: When creating a graphic editor like MS Visio.

• Singleton pattern
  - Definition: Used when fixing the number of class objects at less than N.
  - Advantages: Easy to manage the number of objects in the class.
  - Use examples: When a device manager, which needs only a single object, manages objects.

• Diagram of the Singleton example class



Singleton.java - Class with only 1 instance

```
1.  package DP005_singleton;
2.
3.  public class Singleton {
4.      private static Singleton singleton = new Singleton();
5.      //Set to private to prevent access from outside, and set to static so that it is available throughout one program.
6.      private Singleton( ){ // Declare the access modifier of the constructor as private so that it cannot be created outside.
7.          System.out.println("An instance has been created."
8.
9.      }
10.     public static Singleton getInstance( ) {
11.         return singleton; // Transform the created Singleton instance.
12.     }
13. }
14.
```

Main.java - Test class with the Main() method

```
1.  package DP805_singleton;
2.
3.  public class Main {
4.      public static void main(String[ ] args) {
5.          System.out.println("Start.");
6.          Singleton obj1 = Singleton.getInstance( ); // An instance is returned.
7.          Singleton obj2 = Singleton.getInstance( ); // An instance is returned.
8.
9.          if(obj1 == obj2){ // obj1 and obj2 refer to the same value.
10.             System.out.println("obj1 and obj2 are the same instance.");
11.         }else{
12.             System.out.println("obj1 and obj2 are different instances.");
13.         }
14.         System.out.println("End.");
15.
16.     }
17. }
18.
```

② Structural patterns

• Adapter pattern
  - Definition: The programming interface of a class is changed to that of another class. This pattern can be used to make irrelevant classes work together in one program.
  - Advantages: A new class and its function can be added easily.

• Bridge pattern
  - Definition: The class interface and implementation contents are separated, thus enabling the developer to change or replace the implementation contents without having to change the code contents of the client.
  - Advantages: Separation of interface and implementation. Implemented objects can be exchanged and configured at runtime. #ifdef~#endif syntax can be replaced.

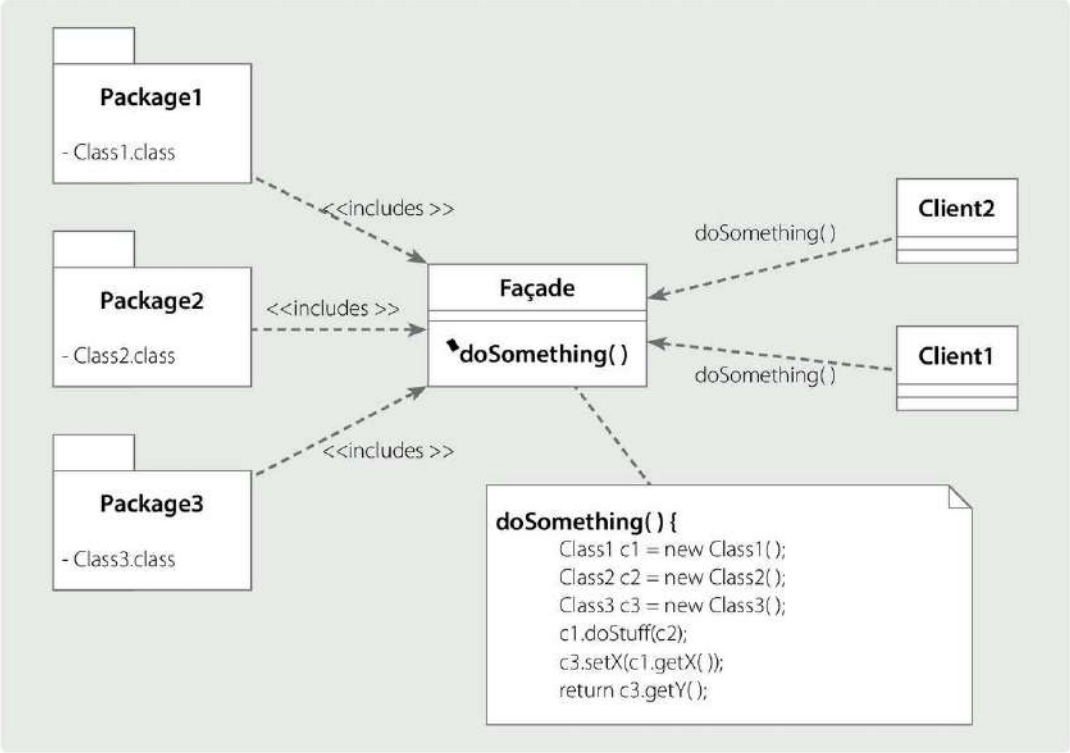
• Composite pattern
  - Definition: A pattern that enables the user to handle a specific object and the objects containing a specific

object in the same way as if a directory and files were combined and treated as a directory entry.
- Advantages: Convenient because the basic object and the constituent object do not have to be separated when writing a source code.

• Decorator pattern
- Definition: A class structure used to dynamically add a function to a specific object or to delete the added | function.
- Advantages: Quite simple and flexible when adding a function to an object.

• Facade pattern
- Definition: When several classes have a close relationship and play a role as a whole, this method enables the user to receive the desired function using the representative class even when no class to represent the role has been defined, and external clients do not directly handle each class.
- Advantages: A simple interface can be provided for a complex subsystem. A complex or recursive dependency relationship can be removed by stratifying the dependency relationship between classes.

• Facade class diagram



- Java source code
The following example of Java code is an abstract example in which the user (you) accesses the internal parts

(CPU, HDD) of the computer using the facade (computer).

```
/* Complex parts */

class CPU {
    public void freeze( ) { ... }
    public void jump(long position) { ... }
    public void execute( ) { ... }
}

class Memory {
    public void load(long position, byte[ ] data) {
        ...
    }
}

class HardDrive {
    public byte[ ] read(long lba, int size) {
        ...
    }
}

/* Façade */

class Computer {
    public void start Computer( ) {
        CPU cpu = new CPU( );
        Memory memory = new Memory( );
        HardDrive hardDrive = new HardDrive( );
        cpu.freeze( );
        memory. load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
        cpu, jump (BOOT_ADDRESS);
        cpu.execute( );
    }
}

/* Client */

class You {
    public static void main(String[ ] args) throws ParseException {
        Computer facade = /* grab a facade instance */;
        facade.start Computer( );
    }
}
```

• Flyweight pattern
  - Definition: A design that separates information into shareable and non-shareable information in order to share information, and which defines shareable information as an object and performs information sharing.
  - Advantages: Storage space can be reduced, and multiple objects can be handled easily.

• Proxy pattern
  - Definition: A pattern used to represent a complex object or an object that takes time to create within a simpler object. As creating an object uses many resources and/or takes a lot of time, this pattern delays

object creation until the developer actually needs it.

③ Behavioral patterns

• Chain of Responsibility pattern
- Definition: When a request is sent to a specific object after configuring a chain based on the relationship between objects, this design makes another object in the chain handle the request if the object concerned cannot process the request.
- Advantages: The degree of system coupling can be reduced, and responsibilities among objects can be distributed more flexibly.
- Disadvantages: Not recommended if time prediction is important, such as the real-time system, because the request processing time and processing status are not accurate.

• Command pattern
- Definition: While the Chain of Responsibility pattern sends a request within the chain of the class, the Command pattern sends a request to a specific object only. The request also includes a request for a specific processing task in an object, and the request concerned is sent using a known public interface.
- Advantages: An object-oriented alternative to the callback function. It is useful when supporting the Undo/Redo function.

• Interpreter pattern
- Definition: A class structure that defines relatively simple grammar itself as a class to perform necessary functions efficiently without a separate data structure, in addition to a grammar check.
- Advantages: Easy to change, expand or implement grammar.

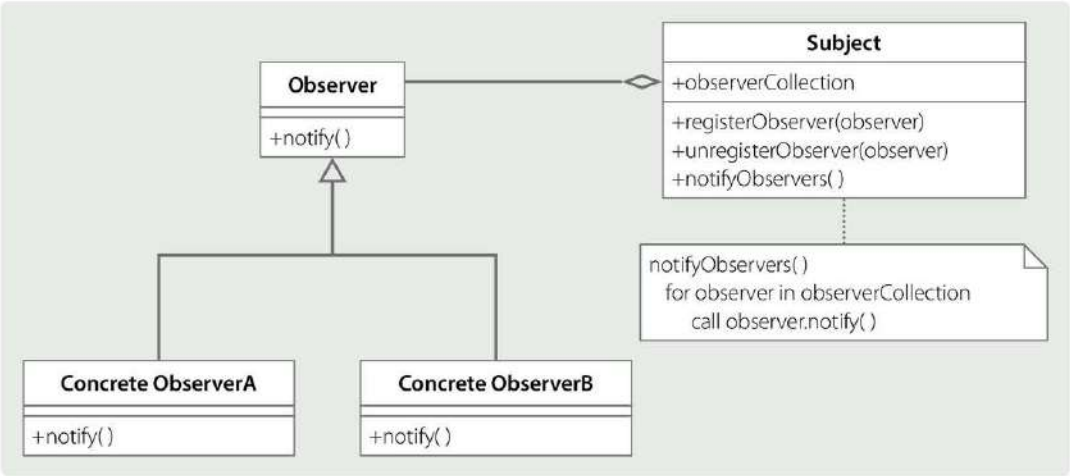
• Iterator pattern
- Definition: Movement using a data list or collection is allowed, using a standard interface, without considering the internal structure of the object.

• Mediator pattern
- Definition: A class structure that can convert a complex M:N relationship into a M:1 relationship by disconnecting the direct relationship between classes when the classes have established a complex M:N relationship directly.
- Advantages: Class coupling can be reduced.

• Memento pattern
- Definition: A class structure that defines the status information of an object as a separate class and stores the object of that class in a list data structure whenever needed, so that the status of the object can be easily restored at a specific point in time.

• Observer pattern
- Definition: A design that creates a one-to-many dependency between objects when the state of one object changes, so that related objects can be automatically notified and updated.

• Diagram of the Observer pattern class



- Example of Java source code

```
/* File name: EventSource. java */

package obs;
import java.util. Observable; // This is the subject that sends asignal to the observer.
import java.io. BufferedReader;
import java.io. IOException;
import java.io. InputStreamReader;

public class Event Source extends Observable implements Runnable
{
    public void run( )
    {
        try
        {
            final InputStreamReader isr = new InputStreamReader( System. in );
            final BufferedReader br = new BufferedReader ( isr );
            while ( true )
            {
                final String response = br.readLine( );
                setChanged( );
                notifyObservers( response );
            }
        }
        catch (IOException e)
            e.printStackTrace( );
        }
    }
}
```

```
/* File name: ResponseHandler.java */

package obs;

import java.util. Observable;
import java.util. Observer; /* This is the observer. */

public class ResponseHandler implements Observer
{
    private String resp;
    public void update (Observable obj, Object arg)
    {
        if (arg instanceof String)
        {
            resp = (string) arg;
            System.out.println("₩nReceived Response: "+ resp );
        }
    }
}
```

```
/* File name: myapp. java */
/* The program starts here. */

package obs;

public class MyApp
{
    public static void main(String args[ ])
    {
        System.out.print ln("Enter Text >");

        // The event issuing agent is created. - Receiving a string from stdin.

        final Event Source evsrc = new Event Source( );

        // The observer is created.
        final ResponseHandler respHandler = new ResponseHandler( );

        // Make the observer subscribe to the event published by the issuing subject.
        evSrc.addObserver( respHandler );

        // Beginning of a thread that issues an event.
        Thread thread = new Thread(evSrc);
        thread.start( );
    }
}
```

- State pattern
  - Definition: When a new state can be added to the internal state of a certain object continuously, this method facilitates the addition of a new state. When the state of an object, including the added state, is changed, the object state information is defined and used as a class inheritance structure so that performance can be changed without modifying the existing source code.
  - Advantages: The sentence that compares state values inside the object is removed. Helps maintain information consistency.

- Strategy pattern
  - Definition: Provides a method of selecting the most suitable character by enabling the algorithm to change character.
  - Advantages: Easy to select a desired algorithm; and a new algorithm can be applied without modifying existing code.

- Template Method pattern
  - Definition: If concrete implementation is different when inheritance is used, but the basic frame of the algorithm is the same within a larger scheme, the same basic frame can be created and managed by the superclass as a module.

- Visitor pattern
  - Definition: Separates work targets and work items and defines them in a class structure so that new types of work can be added easily, and various tasks can be performed for multiple objects without any inconveniences.

VII. User Interface (UI)/User Experience (UX) Design

▶▶▶ Recent trends and major issues

The user interface is one of the most important elements of a computer system. Recently, the important of the user interface as an important factor in improving user satisfaction, such as user-centric planning and design, has been increasing daily.

▶▶▶ Learning objectives

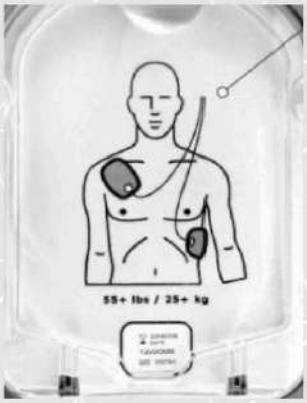
1. To be able to understand and apply the principles of user interface design.
2. To be able to explain Human Computer Interaction (HCI).
3. To be able to understand and use the GUI (Graphic User Interface) properly.

▶▶▶ Keywords

- UI/UX
- HCI (Human Computer Interaction)
- GUI (Graphic User Interface)

+ Preview for practical business Even human life can be harmed by a poorly designed UX.

In the medical device field, where software directly impacts human life, people are more sensitive than in any other field since minor defects in a medical device can have devastating consequences for a patient, even including death. So which of the following could have more catastrophic consequences, a faulty in a physical device or a user's mistake during an operation? Most people answer the former; however, user mistakes resulting from complex operational methods and such like can pose a greater risk. For example, if the pad attachment position of a cardiac defibrillator is not intuitively described, emergency measures will not have any effect, and the result is likely to be the opposite. The following examples show how important UX is.



Therac 25, a radiation therapy device developed in the early '70s, was upgraded to address the inconvenience caused by the separation of physical device setting and prescription processing by the administrator console. In July 1985, an accident occurred during radiation therapy at the Ontario (Canada) Cancer Foundation, where a patient died from excessive exposure to radiation in November of the same year. A report on the analysis of medical accidents revealed that the accident was caused by a wrongly designed error message during the upgrade. In the end, the accident was attributed to the fact that the radiation therapist was insensitive to the unknown error message of Therac 25. (Similar error messages were distinguished by the number at the end of each message, which cannot be clearly identified.) Because of this, the device was abnormally interrupted five times, exposing the patient to excessive radiation during treatment.

Recently, many companies have striver to maximize the use of the customers using the information on the limited size screen to secure more customers and increase sales in the field of mobile business using smartphones. Moreover, a growing number of companies are contemplating using UI/UX to provide customer-oriented valuable services and forming expert organizations.

01 User Interface Overview

The word "Interface" refers to a device that faces each other between two objects as the composition of the word ("inter"+"face") implies. The user interface refers to a device or software that ensures a smooth exchange of information between user and system. It acts like a window for software. For example, if a user inputs/outputs commands or selects a menu using a keyboard or mouse to use a specific software, it belongs to the user interface. The interface should be designed with a focus on the following points.

A) Consistency

The user interface should be created consistently. However, it is generally difficult to maintain consistency when developing a large system, since several people design and implement user interfaces. User interface standards should be established before development to make user interfaces consistent, and user interfaces should also be checked after their development to correct any errors.

B) User-centered design

The user interface should be designed with a focus on the user because it is for the user, who should be able to control it effectively. Therefore, it is essential to design the user interface in such a way that users can learn "input and output language" easily.

C) Feedback

The user interface should provide meaningful feedback. When a user presses a wrong button or tries to perform an operation incorrectly, the user interface should enable the user to easily understand his or her mistakes.

D) Confirming destructive behavior

When a user tries to delete data or files, the user interface should confirm it before executing it. If possible, the user interface should be designed in such a way that the user cannot cause critical errors. The Undo function is also a useful method in this regard.

02 User Experience Overview

User experience comprises the ideas and actions of the designer who tries to improve the user's environment. It refers to comprehensive experience such as experience, emotion, perception, attitude, and response when a user wants to achieve a purpose by using a system or service. It is a process of researching a convenient screen design and environment for the users.

A) Differences between user experience (UX) and user interface (UI)

User experience (UX) refers to the study of invisible things, whereas user interface (UI) refers to the concrete

work to make invisible things visible. In addition, UX is the feeling, attitude, and behavior of the user, whereas UI is the environment in which the user faces the interface directly. UX is also a study on improvements based on statistics relating to user experience, while UI is the realization of such improvement studies. That is, UX refers to experience of what is visible, whereas UI refers to the interface itself that is shown to the user. The main differences between UI/UX can be briefly summarized as follows.

UX	UI
Process	Result
Dish	Food
Planning	Design
Emotion	Reason

03 UI/UX Design Tools

The following summarizes the representative UI/UX tools that can be used to organize and visualize the thoughts and ideas of users and designers, analyze developed services and run user tests, and can improve or accelerate their work.

A) MAKE - Turning ideas into products

① Organizing information - GitHub, Trello

② Wire-frames - Microsoft Visio, Moqups

③ Prototyping - Adobe Photoshop, ketch3

B) Check - Checking user analysis and response methods

① Visual analytics - Beusable, Clicktale

② Analytics & metrics - Google Analytics, Mixpanel

③ AB testing - Google Optimize, Optimizely

④ Record users - Beusable, hotjar

C) Think - Checking market feedback continuously

① Recruiting users - Pivot Planet, Clarity

② Online surveys - Polldaddy, hotjar

③ Capture in-site feedback - LiveChat, mouseflow

④ Testing layouts remotely - Chalkmark, UsabilityHub

TOPCIT ESSENCE

VIII. Programming Language and the Development Environment

▶▶▶ Recent trends and major issues

The use of Java and C-like languages has become increasingly popular among enterprises due to the trend of developing software and apps on the mobile/cloud. However, these languages have been recognized as the two major programming languages and those most frequently selected by developers. In addition, it is important to learn the related skills beforehand because these languages are the basis for learning other programming languages and are used in various business fields.

▶▶▶ Learning objectives

1. To be able to describe the characteristics of programming languages (unstructured language, structured language, object-oriented language).
2. To be able to compare the characteristics of major programming languages (C, C++, Java, Python, JavaScript).
3. To be able to understand the software development framework and explain its types.
4. To be able to understand and explain the integrated development environment (IDE).

▶▶▶ Keywords

· Programming language, compiler, interpreter
· Code reuse
· Software development, Spring and standard e-government frameworks
· Integrated development environment

+ Preview for practical business

Selecting a programming language

In a real enterprise environment, the selection criteria for programming languages for the development of products and services are determined in various ways. Hardware, OS environment, introduced solutions, business characteristics, application of the latest development languages, and the status of available developers are just some of these criteria. However, it is true that an alternative language is selected if there are only a few developers and labor costs are excessively high, even though the given development language is eminently usable.

For example, if Java is selected to implement all the functions of a given project to develop a system with numerous batch-like tasks, then the project is likely to face the difficult problem of having to improve batch performance. Alternatively, if the C language is selected in order to improve speed only, there is a high possibility that the project will be faced with the difficult problem of productivity and test coverage.

Therefore, the reality is that many projects select a programming language based on experience, and various languages and architectures are applied in the same project. In particular, complexity and diversity increase significantly in recent development projects that require mobile services in various fields.

Software development framework and integrated development environment (IDE)

These days, software developers collaborate with developers around the world who use different languages and different versions of the compiler and may have to write new codes or maintain the software, which itself may use an old library that is more than 10 years old. To adapt to the rapidly changing business environment, the methods of collaboration, customer support, and coding are changing. Also, an integrated development environment based on a continuous integration environment is created to provide services on time. The framework consists of an execution environment, development environment, operation environment, and management environment. It also improves development productivity by maximizing modularization and reusability, and provides various libraries that can improve software quality.

01 Programming Language Overview

A) Concept of the programming language

A programming language enables users to write a program using a familiar language, that is, a language that is similar to the language they use every day. In other words, programming languages should be used to enable humans to communicate with computers. Thus, programs written in each programming language are translated into machine language using a compiler or an interpreter, so that computers can understand the languages.
In addition, it is necessary to secure the productivity and quality goals using a proper programming language in each area, because each programming language has different uses.

<Table 21> Low-level language and high-level language

Item	Description	Representative language
Low-level language	Machine-centric language that is executed quickly and has a different code for each machine. There is a machine language composed of a binary system that is easily understood by the computer, and an assembly language that corresponds to the machine language. There is no compatibility between computer models.	Machine language, assembly language
High-level language	It is easy to write a program that is highly readable and productive compared to low-level language because it is made so as to be similar to natural language, which can be easily understood by humans. A program can be written independently of the machine and converted into low-level language by a compiler or interpreter.	FORTRAN, COBOL, BASIC, C, C, Java, Python

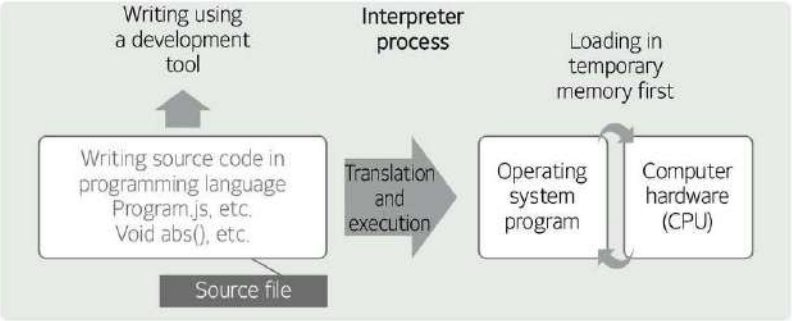
The machine language is written in only 0 and 1 and can be directly understood by hardware, whereas the assembly language expresses the machine language in simple characters. Both the machine language and assembly language are referred to as a "low-level language". However, as it is practically difficult for humans to manually write programs in either machine language or assembly language, high-level languages were created to resolve this problem. High-level languages, represented by C, C++, Java, and Python, enable us to write programs using everyday languages and symbols.

<Table 22> Areas of use of programming languages

Area of use	Major programming languages
Science and technology (used for numerical calculation)	FORTRAN, ALGOL60
Business	COBOL
Artificial intelligence	LISP, PROLOG
System programming	PL/S, BLISS, ALGOL, C/C , Java, Python
Special purposes	GPSS (language used for simulation)

B) Interpreter languages

Interpreter languages convert a source program into a low-level language directly, without requiring any intermediate steps, and execute the program at the same time. Machine language programs are not created by translating high-level languages into intermediate codes and then executing the translated codes with a software interpreter; rather, such programs are executed by calling sub-routines corresponding to the functions of each intermediate code. LISP and PROLOG are representative interpreter languages.



[Figure 19] Interpreter process

① Strengths of interpreter languages

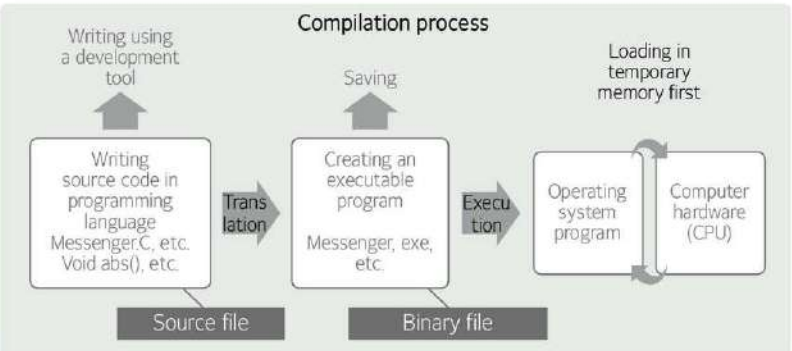
- Programs are executed if needed, without waiting for complete machine language translation.
- Memory can be saved as the source language form is maintained until the program has been fully executed.

② Shortcomings of interpreter languages

- The source program has to be decoded whenever it is executed again, which takes time.

C) Compiler languages

"Compilation" means converting a specific language into another language having an equivalent meaning. The compiler translates a high-level language into a machine language, which is a low-level language, in order to make an object module, after which this module is linked, loaded, and executed. FORTRAN, PASCAL, and C/C are representative compiler languages.



[Figure 20] Compilation process

① Strengths of compiler languages

- Translated object codes can be saved.
- Executable immediately after compilation.
- The reused program can be executed quickly after compilation

② Shortcomings of compiler languages

- Conversion to machine language takes a long time.
- Memory is wasted sometimes because one line of raw program is translated into hundreds of lines of machine language.

02 Characteristics of Major Programming Languages

A) C language

① Introduction to the C language

The C language was developed by Dennis Richie, who worked at AT&A's Bell Lab on the UINX operating system in the 1970s. Most computer languages are motivated by the C language, and most of the famous operating systems, like iOS and Android, are written in C.

② Characteristics of the C language

- Execution is fast, and memory can be managed efficiently and easily.
- Functions can be implemented with paragraphs because the notation is concise.
- Procedure-oriented language: Programs are executed according to the specified sequence.
- More difficult to use than Java because a program in consideration of the array and memory is needed.
- Porting is difficult when the execution environment and machine are changed.

B) C++ language

① Introduction to the C ++ language

C++ is suitable for system programming because it contains most of the features of the C language, and is also suitable for object-oriented programming by supporting classes, operator overloading, and virtual functions.
If software is developed using an object-oriented programming technique, it is easy to write a large-size program, and programs are easy to read because software reusability, scalability, and maintainability are increased.

② Characteristics of the C ++ language

- Object orientation characteristics: C++ reflects the characteristics of object orientation as a language that is developed from the C language to support object-oriented programming.
- Encapsulation and data hiding aspects: The class, a characteristic of C++, is frequently used in encapsulated form, and the actual internal working structure is hidden. Therefore, the user needs to know how to use the class without knowing how it works.

- Inheritance and reusability aspects: C++ has reusability using inheritance. "Inheritance" refers to the process of creating a new class by re-creating the common characteristics of several classes as a single class and extending it. Program reusability can be increased, and software productivity can be enhanced by using inheritance.
- Polymorphic aspect: Polymorphism means that several forms have one name. For example, a person named "Hong Gildong" is an office worker who works hard at work but is also an instructor who gives lectures at an evening academy.

C) Java language

① Introduction to the Java language

James Gosling began developing Java at Sun Microsystems from 1991 as a simple, bug-free programming language to control home appliances. Development started with the C++ language in the initial stages. Even though it improved several problems of C++, home appliance companies did not pay much attention to it. However, the powerful output form called "applet" gained popularity as Java was applied to the Interest by Hot Java owing to the rapid spread of Internet penetration. The advantages of the Java language were recognized because its small and simple structure enabled efficient change and execution, and because it improved upon several functions that caused errors in existing languages. Currently, Java has become the most popular, widely used programming language for apps and mobile devices, and is used to create apps for home appliances and Android devices.

② Characteristics of the Java language

- Java is grammatically similar to the C/C++ language because it derives from the C++ language.
- The complexity of C++ has been simplified.
- Automatic garbage collection is always performed.
- Object-oriented language: Perfect object-oriented language that applies the concept of an object.
- It is independent from the platform as the Java program is executed by the Java virtual machine.

D) Python language

① Introduction to the Python language

Python is an interpreter language developed by Guido Van Rossum in 1990. This object-oriented, conversational language uses frequently used English keywords for high readability and productivity. Python is mainly used as a web programming language based on Django, a web framework based on open source. Recently, it has been widely used in big data analysis and the development of artificial intelligence programs.

② Characteristics of the Python language

- The data type is checked at runtime using dynamic typing.
- It is easy to use as grammar is easy and similar to English sentences.
- Advantageous to asynchronous coding using a single event loop
- Multi-paradigm programming language (supporting procedural languages, functional programming languages, and object-oriented programming).

E) JavaScript language

① Introduction to the JavaScript language

JavaScript is an object-based script programming language that is mainly used in the web browser. It includes a function for accessing the built-in objects of other application programs. JavaScript is also used for server-side network programming such as the Node.js runtime environment.

② Characteristics of the JavaScript language

- Programming is simple, but it may be vulnerable to security.
- Highly scalable and useable due to active sharing among developers because it is operated as an open source.
- Easy to learn because no separate compilation process is required.
- Recently, more and more developers have started using libraries and frameworks related to JavaScript, such as Angular.js, D3.js, Node.js, and React.js.

03 Software Development Framework

A) Concept of the software development framework

A software development framework is a set of code libraries, interface protocols, and configuration information, all of which is used to develop an information system efficiently and to provide the basic framework for software composition. This framework is the representative technology to which the concept of "Inversion of Control" (IoC) is applied, and it enables developers to complete an application by extending the upper-level application functions (business logic, etc.)

That is, this framework provides an environment in which developers can create software easily, by providing structured tools and libraries for data display on the screen, a method of handling data, and interfacing with external modules, instead of developing software from scratch.

The software development framework solves the problem of integration and consistency that occurs while developing a set of classes and interfaces which can cooperate with each other to solve the specific problem of software, and reduces the coding work of developers while increasing their development productivity. The framework also standardizes developers upwards to guarantee a certain level of quality and reduce the risk of new software development using proven technology.

The table below shows the advantages of using the software development framework.

<Table 23> Advantages of using the software development framework

Strengths	Description
Improves code quality	Defines and handles in advance parts that can be written incorrectly during repetitive coding work, thus improving code quality by minimizing the risk of bugs when writing codes.
Increases development productivity	Maximizes development productivity by providing various functions necessary for development in advance, such as components, communication processing methods, and data handling methods, etc.

Improves maintainability	Easy to analyze and modify using a development environment that is structured and systematized to adapt to changes in the person in charge or the user environment.
Reduces the risks posed by new systems	Reduces the risk of new systems by reusing proven technology and the service based on the standardized structure.

B) Spring Framework

① Introduction to the Spring Framework

The Spring Framework is an open-source application framework for the Java platform, which provides a comprehensive basic structure for the development of Java-based enterprise applications, and various services necessary for dynamic web application development. The Spring Framework is utilized as a core basis for the standard e-government development framework on account of its proven stability and flexibility.

② Characteristics of the Spring Framework

The Spring Framework uses a lightweight container called POJO (Plain Old Java Object) to manage the life cycle of the Java object, such as creation and destruction, and guarantees loose coupling between objects by using DI (Dependency Injection), which sets the dependency relationship between objects using configuration files or annotations. It also uses AOP (Aspect-Oriented Programming) to support the combination of functions that are commonly used in multiple modules, but which are not related to core business logic, such as logging, security, transactions, etc., if they are executed after separation. In addition, it provides database libraries such as JDBC, MyBatis, Hibernate, JPA, etc., and supports the interface with various APIs necessary for developing enterprise applications such as JMS, mail, and scheduling.

③ Composition of the Spring Framework

<Table 24> Components of the Spring Framework

Component	Description
Spring Core	This is the core part of Spring. It provides BeanFactory, which implements the IoC container to separate function and configuration.
Spring Context	The basic function of Spring, like Spring Core, which provides the method of access to functional objects (bean) implemented based on Spring.
Spring DAO	An abstraction layer for JDBC that is integrated with the ORM framework and improves transaction management.
Spring ORM	A package for integration such as JDO for object/relation mapping, JDO, Hibernate, and iBatis. Spring ORM supports object-relational models.
Spring AOP	Maintainability and ease of change are supported by separating cross-cutting concerns such as logging, security, transaction, etc.
Spring Web	A package that provides the basic functions needed to develop general applications and is used for integration with WebWork or Struts.
Spring WebMVC	A package that implements general WAP functions like Struts as a Spring version.

C) Standard e-Government framework

① Introduction to the standard e-Government framework

Designed for public projects, the standard e-government framework standardizes and implements the basic functions necessary for developing and operating a Java-based system. The framework provides the implemented environment (execution, development, management, and operation) and common modules that are the basis for developing various types of software required by the private sector and the government. Developers can develop a system by reusing the common modules in the base environment provided by the standard framework and by developing unique functions exclusively for each service.

② Purpose and characteristics of the standard e-government framework

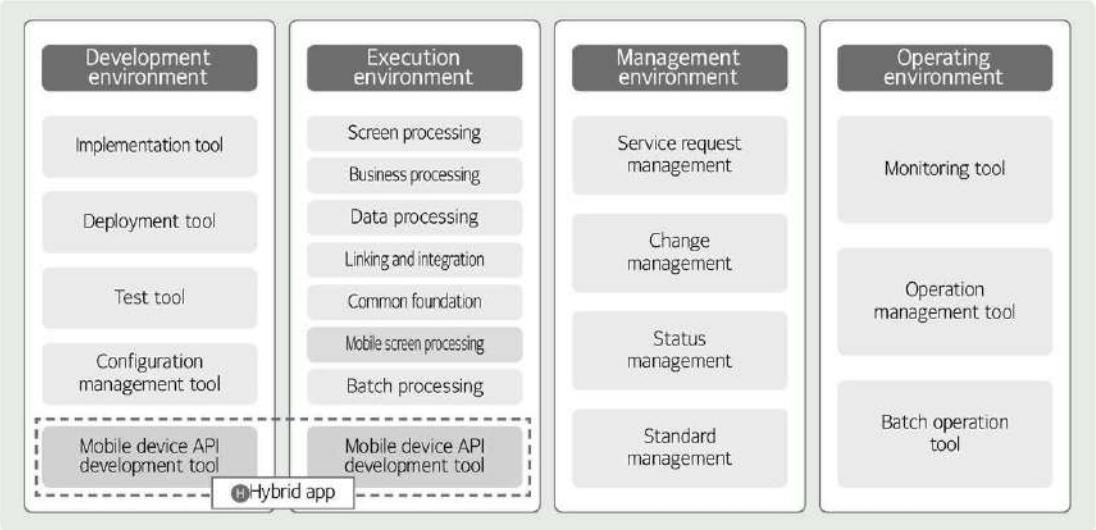
The framework aims to standardize application software and improve quality and reusability by establishing a development framework that can be applied to public projects.

- Improves e-Government service quality.
- Improves the efficiency of investment in informatization.
- Enables fair competition by conglomerates and small and medium-sized businesses using the same development base.

<Table 25> Characteristics of the standard e-Government framework

Characteristics	Description
Compliance with open standards	Eliminates dependence on a specific service provider by using open source-based technology.
Linkage with commercial solutions	Ensures interoperability by presenting a standard that can be linked with various solutions.
Aiming for national standardization	Software development standards are established at the national level by operating an advisory committee composed of members from the private and public sectors and academic circles.
Change flexibility	Easy to replace by modularizing the components and minimizing the impact of changes between modules through interface-based linking.

③ Composition of the standard e-Government framework



[Figure 21] Components of the standard e-Government framework

The framework is composed of the execution environment, development environment, operating environment, management environment, and common components, thus providing the application architecture, basic functions, and common components that are necessary for developing a web-based information system.

<Table 26> Components of the standard e-Government framework

Component	Description
Execution environment	An application program environment that supports the easy standardization of screen, server program, data, and batch processing function development needed when developing an e-Government business program.
Development environment	An environment needed to develop a business program comprising such tools as the data development tool, test automation tool, code test tool, template project creation tool, common component assembly tool, server environment management tool, batch template project creation tool, batch job file creation tool, etc.
Operating environment	An environment for managing and operating the service running in the execution environment (monitoring, deployment, management system, etc.) An environment for operating a batch environment (batch execution, scheduling, result monitoring, etc.)
Management environment	A module used to distribute the development framework and common services to each development project for management.
Common components	A set of reusable development components that can be used in common by e-Government projects when developing application software.

04 Integrated Development Environment (IDE)

A) Concept of an integrated development environment (IDE)

An Integrated Development Environment (IDE) is a type of software used for development purposes that provides an environment in which tasks related to all program development can be handled with one software, such as coding, debugging, compilation, and distribution. Separate software had to be used in the traditional development environment in order to use such diverse functions as the compiler, text editor, debugger, and remote server accessor. However, the IDE bundles all these functions into a single tool and provides it via an interactive interface. Visual Studio, Eclipse, and Xcode are popular IDEs that support various programming languages.

<Table 27> Components of the integrated development environment (IDE)

Component	Description
Editor	A tool used to input and edit codes for programming.
Build tool	A tool used to convert code written in programming languages into machine-readable code, such as the compiler, interpreter, linker, etc.
Debugger	A tool for detecting errors while running a program.
Project management	A tool that is used to manage a project by providing a system of collaboration system between teams.

B) CI(Continuous Integration)

① Concept of CI (Continuous Integration)

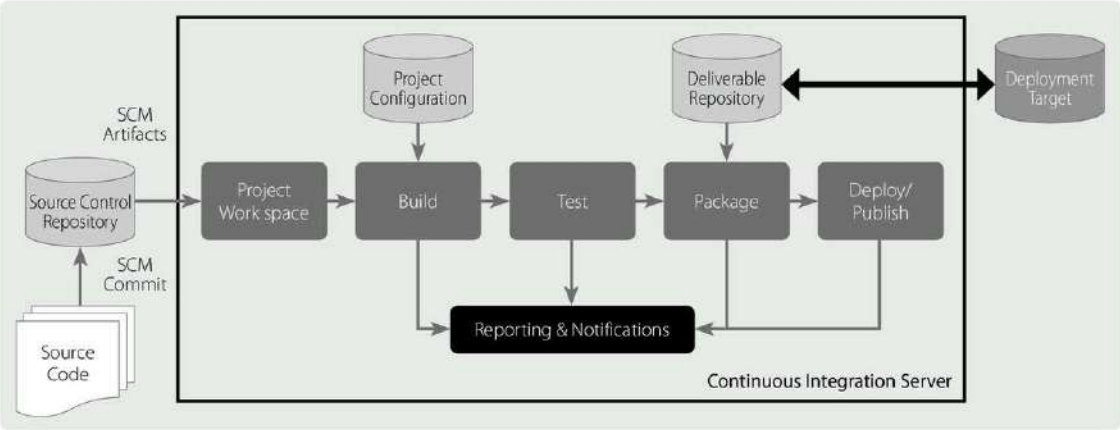
Continuous Integration (CI) is a software development activity that frequently integrates the results of a team composed of several members. Each developer performs automated build and testing, and then merges changes to the code in the central repository. Codes are merged several times per day (at least once per day) after quickly checking and verifying any errors during automation build.

From a software engineering perspective, CI is performed to continuously control quality during software development. CI aims to minimize the time required to improve software quality and distribution by replacing the classic method of quality control.

② Necessity of CI (Continuous Integration)

- Builds configuration management.
- Provides an environment in which developers can concentrate on development activities only.
- Detects and eliminates problems that may arise when integrating codes in advance.
- Supports a development process (agile technique, etc.) that receives feedback frequently based on frequent deployment.

③ Composition of CI (Continuous Integration)



[Figure 22] Composition diagram of CI (Continuous Integration)

A CI server is required to perform CI (Continuous Integration) that can conduct daily build, which requires at least one accessible source code repository, as well as build script sets and build procedures, and a test suite for built artifacts.

C) Software build

"Software build" refers to the process of converting source code files into independent software artifacts that can be executed on a computer, or the results thereof. As one of the software quality assurance activities, a daily build reduces integration risks, prevents quality deterioration, monitors the progress of initial fault analysis, and boosts the morale of developers.

D) Daily build and operation test

- The process of comprehensively compiling software products again every day and running a series of tests to verify basic operation.
- Daily build can simultaneously increase a project's efficiency and improve customer satisfaction, primarily by reducing such risks as software integration failure, poor quality, and low project visibility, and by saving time.
- The daily build and operation test can be utilized for all projects, including large-size projects, small-size projects, off-the-shelf software, and business systems.

E) Software deployment

"Software deployment" refers to any act of creating a software system to be used, where the deployment process is composed of interactions together with possible transitions between them. These activities can take place either from the developers' point of view or the consumers' point of view. It is difficult to define with any exactitude the deployment procedures for each activity because deployment is unique to every software system. Therefore, "deployment" should be understood as a general procedure that can be defined by the specific requirements or characteristics of the customer or user. Software deployment activities include release, installation and activation, deactivation, uninstallation, update, built-in update, version tracking, and retirement.

TOPCIT ESSENCE

IX. Software Testing and Refactoring

▶▶▶ Recent trends and major issues

Recently, the amount of time, money and effort expended on software testing when developing software has been rising steadily. IT companies also recognize its importance to such an extent as to form a QA (Quality Assurance) team or department specializing in software testing.

▶▶▶ Learning objectives

1. To be able to explain the concept of testing and compare test case design methods.
2. To be able to describe test levels (types) and purposes.
3. To be able to explain the concepts of software refactoring and major refactoring activities..

▶▶▶ Keywords

- Testing process, testing type
- Specification-based technique, structure-based technique, experience-based technique
- White box testing, black box testing
- Software code smell and refactoring

+ Preview for practical business The necessity of software testing and the reality of enterprises

In March 2005, bus companies lost KRW 500 million won per day in free rides caused by faults or failures of T-Money bus terminal software.
In October 2007, Toyota Motors paid with an operator USD 1.6 billion in compensation due to the sudden unintended acceleration of the "Camry" model's ECM firmware fault.
In August 2009, the launch of "Naro", Korea's first space launch vehicle, was delayed due to a software fault in the automatic launch device software related to pressure measurement.

The abovementioned incidents are widely publicized cases of software faults. Software is ubiquitous and directly related to the existence of companies and the daily life of countless individuals. The most obvious question that springs to mind is why such faults even occur when software is so important. Everyone thinks they know software testing well. However, when software testing is actually performed, it is not performed as well as expected due to various problems. Currently, enterprises are facing many problems due to the irrationality of the contract itself, frequent changes to the requirements and the unreasonable demands of customers, the limitations of support at the organizational level, impractical procedures and paperwork, numerous cases of reporting and meetings, and reports on such reporting and meetings. More specifically, enterprises have difficulty in running software testing due to the following reasons.

- It is a well-known fact that anywhere from 40% to 60% or more of the total development cost is spent on software testing, as testing is very important. However, the reality is that those involved in software development are unable to pay sufficient attention to testing because it is difficult to perform their own fixed tasks only, and that few developers are actively interested in any testing other than unit testing.
- As development itself is conducted with insufficient personnel and resources for a period shorter than the actual schedule, it is not easy to consider investing in testing.
- Testers who perform testing tasks sometimes know too little about testing.
- Testing is performed without an accurate understanding of the testing techniques.
- Decision makers or managers do not fully recognize the importance of testing. Even though test engineers with a sense of mission try to run testing properly, executives sometimes force them to stop testing without understanding its importance.

[Source: Factors constraining software testing http://eunguru.tistory.com/158]

The reality is that the service operation team and the development team also perform testing tasks internally in companies, except for conglomerates. While the number of people remains limited, the number of channels and workloads is increasing continuously and the test period is being shortened, resulting in the release of incomplete services and, consequently, higher maintenance costs.

- Companies recognize testing as unnecessary and as an additional expense rather than an investment.

These days, however, software is gradually crossing over from the traditional domain and spreading to every sector (home appliances, wireless devices, industrial devices, medical devices, etc.) through the IoT, big data, mobile, and cloud. Enterprises are increasingly placing greater emphasis on improving the efficiency of development and operation and recognizing the importance of software quality. As the users' expectation is rising, enterprises need to run software testing systematically, resulting in higher investment, the recruitment of increasingly professional human resources, the use of more automated tools and process establishment.

01 Concept and Process of Testing

A) Concept of testing

Testing is a method of checking or confirming that the operation, performance and stability of an application or system satisfies the demands of the user or customer, by identifying or detecting defects or faults etc. The general principles of testing are as follows.

① Testing shows the presence of defects

Testing activity is designed to reveal the presence of defects. However, it is almost impossible to prove that software has no defects.

② Exhaustive testing is not possible

Even a very simple program cannot be tested for all cases.

③ Testing should start during the initial stages of development

If tested early, the development period can be reduced and defect prevention activities can be performed, eventually resulting in a reduction of costs.

④ Pesticide paradox

The repetitive use of the same pesticide mixes to eradicate insects will over time lead to the insects developing resistance to the pesticide, thus rendering the pesticides ineffective. The same applies to software testing. If the same set of repetitive tests is conducted, the method will be totally ineffective in discovering new defects.

⑤ Testing is context dependent

The approach as well as the methodology and severity differ for each field.

⑥ Absence of error - fallacy

Finding and fixing defects will be of no help whatsoever if the 'system build' is unusable and does not fulfill the user's needs and requirements.

B) Testing process

Testing should be coordinated and managed by emphasizing a process comprising various components. The table below shows the main activities of the process concerned.

<Table 28> Main activities of the testing process

Process	Main activities
Analyzing and designing tests	• Reviewing the test basis. • Identifying test situations/requirements/data. • Assigning a test technique. • Evaluating testability. • Creating a test environment.
Implementing and executing tests	• Specifying test cases: Priority selection, data creation, procedure writing. • Preliminary testing. • Running tests and recording the results. • Comparing expected results.
Evaluating and reporting completion conditions	• Checking whether the completion conditions are met. • Creating the first test report.
Test deadline activities	• Checking the deliverables and storing the test ware. • Evaluating the test process.
Planning and controlling tests	• Setting the test purposes/goals and researching the targets. • Developing test strategies and analyzing the risks. • Establishing strategies and test completion conditions.
Estimating tests and forming organizations	• Planning tests and controlling test management. • Reporting and planning/designing reporting. • Reporting progress.

C) Test design

① Test design overview

The test design is used to check to what extent the target has been tested by identifying and executing test cases. How to proceed with the test design depends on the testing context, such as the composition of the test organization, the maturity of the testing and development process, time constraints, and the number of participants in the test. The test basis is analyzed to identify the test conditions, and the test cases and test data are designed and specified using the test design technique based on the test conditions identified during the test analysis process.

② Composition of test cases

A test case is composed of a set of input values, pre-conditions when executing, execution procedures, expected results, and conditions after execution. It is prepared to check a specific test condition.

<Table 29> Components of a test case

Component	Contents
Test case ID	A number or an identifier used to identify a test case.
Test case name	A simple and clear description of the test content.
Precondition	Information on the preconditions, such as the environment required to run a test, or test data.
Test running procedure	A concrete test running procedure composed of up to 7 steps.
Expected result	A basis for deciding whether the test has been performed as intended.
Result (pass/fail)	The result of running the test case.
Traceability	Information on requirements related to the test case and applied techniques.
Importance	The criteria for selecting a test target when time is limited.
Remarks	Description related of the intent and purpose of creating test cases.

③ Test case design techniques

Test design techniques can be classified into specification-based techniques, structure-based techniques, and experience-based techniques depending on the referred test basis, an origin of the test design.

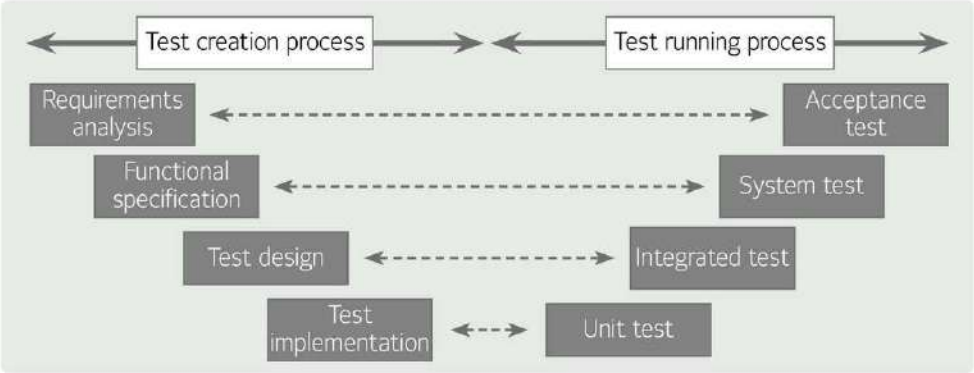
<Table 30> Test case design techniques

Item	Technique	Description
Specification-based technique	Equivalence partitioning	A test case is designed in such a way that a test is performed with the representative value placed in an equally divided area.
	Boundary value analysis	Boundary values are also included when designing a test case because there is a high possibility of a defect being found in the input value corresponding to the boundary of the equal division.
	Pairwise testing	A table is created in such a way that the values needed for the test are combined with other values at least once, and a design is executed using the table.
	Decision table testing	A test case tests the combination of input values indicated in the decision table and the stimuli (cause).
	State transition testing	The relationship between events, actions, activities, states, and state transitions are designed based on a state transition diagram.
	Use case testing	Test cases are extracted from use cases when a system is modeled as a use case.
Structure-based techniques	Control flow testing	All possible event flow (path) structures can be tested when executed using a component or system.
	Coverage testing	Test cases are designed to achieve coverage, which indicates how far the system or software structure has been tested by the test suite.
	Elementary comparison testing	Test cases are identified in such a way that the combination of input values is tested using the concept of the modified condition/decision.
Experience-based techniques	Exploratory testing	An unofficial test is designed that uses the information obtained while running tests, so that testers can control the test design actively while running the test and so that a new and better test can be designed.
	Classification tree method	The test cases expressed as a classification tree are combined with the representative values of the input and output domains.

02 Testing Types and Techniques

A) Testing types

There are various types of testing depending on various levels, as shown in <Table 30>. The overriding principle is that the testing subject, testing purpose, and test environment should be considered for each type. .



[Figure 23] Test level types

Software testing requires test plans and strategies for each test level, which is a set of test activities corresponding to the development stage. Specific details, such as the test basis, test target, test approach, tester or organization, etc., which are the development deliverables to be referred to when identifying the goals of each test level and test case, should be defined.

<Table 31> Test types

Test type	Purpose	Performing Tester/Test Organization	Environment
Acceptance test	To check compliance with the requirements.	User	User environment
System test	To check the overall functional and non-functional tests in an environment similar to the actual environment.	Test organization	Environment similar to an actual user environment
Integrated test	To find a defect in the interface between the unit modules.	Development organization or testing organization	Development environment or test environment
Unit test	To detect any defects in the unit modules.	Development organization	Development environment

B) Testing techniques

Testing techniques are broadly divided into white box testing and black box testing, which have the following characteristics.

① White box testing

- White box testing is called structural or code-based testing.
- It is mainly used in unit testing to find defects in testable software (module, program, object, class, etc.) and to verify the function of them. The source code is used to perform control flow testing, condition/decision coverage testing, and elementary comparison testing.
- This technique can be applied to integrate testing using a structural approach.
- White box testing is divided into static analysis, which is used to detect a pre-defined error by analyzing the internal structure of the implemented source code, and dynamic analysis, which is used to detect an error

that could occur in an actual situation.

• The table below shows the representative white box testing techniques.

<Table 32> White box testing techniques

Item	Contents
Structural technique	Measures and evaluates the logical complexity of the program.
Loop test	Only the loop structure of the program is tested.

② Black box testing

• Black box testing is called function or specification-based testing.

• It is mainly used in system testing to verify both functional and non-functional requirements. Functional requirements are verified based on the specification, whereas non-functional requirements are tested for such factors as performance, availability, security, etc., based on the defined specification.

• Testing focuses on whether software runs according to the requirements specification.

• Functional testing and non-functional testing are performed based on the requirements specification and the external interface, without referring to the internal structure of the software module/component or system.

• The table below shows the representative black box testing techniques.

<Table 33> Black box testing techniques

Item	Contents
Equivalence partitioning	• Tests are conducted by selecting test cases with various input conditions. • E.g. Testing by separating conditions into x < 0, 0, x, 100, and x > 100 within the range of 1 to 100.
Boundary value analysis	• Tests the accuracy of the result based on the boundary value. • E.g. Testing with x=0, x=100, x=-1, x=200, etc. within the range of 1 to 100.
Cause-effect graph	• Detects an error by expressing the impact of input values on output values using a graph.
Error guessing	• Detects errors that could be overlooked with sense and experience. • E.g. Checks without an input value or inputs a number in the text field.

03 Refactoring

A) Concept of refactoring

① Refactoring overview

The term "refactoring" refers to the improvement of a program's internal structure without changing the program operation when viewed from the outside. That is, the external function of the software is not modified, but the internal structure and relationship are simplified to improve the maintainability and quality of the software. The following two points are important in defining refactoring.

• Even if refactoring is performed, the program operation as viewed from the outside does not change.

• Refactoring improves the internal structure of a program.

Refactoring is performed for the following reasons.

- Refactoring makes error detection and debugging easier.
- Changes in software requirements can be responded to more effectively.
- Complex code is simplified, and source code readability is improved.
- Program productivity and quality are improved.

② Time for refactoring and procedure

- Codes are written first without plans, and the second similar codes are written in duplication. However, refactoring is performed when writing the third similar codes. Therefore, it is important to decide on the time to perform refactoring.
- Refactoring is generally performed when the time taken to add a new function to the existing code is inefficient; when the design changes in such a way that addition is not easy; when eliminating bugs; or when reviewing codes.
- Refactoring procedure

<Table 34> Refactoring procedure

Target selection	Maintenance, inspection, and XP methodology application
Composition of the organization	Including a senior who plays the role of a mentor
Performance control	Change control, configuration management, CCB control
Performing technique	Design pattern, AOP execution
Testing	Unit/integration test, regression test
Result arrangement	Documentation, actualization, application of the operating system

- When performing refactoring, it is necessary to make small-scale changes (single refactoring) and test whether it works. If it works well, one can proceed to the next factoring phase. If not, it is necessary to solve the problem, undo the refactoring work, and then keep the system running.

B) Concept of a code smell

① Concept of a code smell

The term "code smell" means the code to be refactored, such as a program that is difficult to read or has duplicated logic, which makes it difficult for developers to read or maintain it. A code smell is the part that is difficult to understand, modify or expand. If it is difficult to read the code, refactoring should be considered.

② Common code smells

Type	Description
Duplicate code	Functions or data codes are written in duplication.
Long method	The inside of the method is too long.
Vast class	One class has too many properties and methods.
Long parameter list	A method has too many parameters.
Class modified for two reasons	If the method of one class is modified for more than two reasons, the class is responsible for more than one job.

Type	Description
Modifying multiple classes at the same time	When minor changes are required in several related classes each time a specific class is modified.
Data clumps	Data that should be handled together are not in one class.
Primitive obsession	Only basic types like int are used without creating a class.
Switch statements	Operations are split using a 'switch' statement or an 'if' statement.
Lazy class	A class that doesn't do much.
Speculative generality	Excessive generalization such as "Expansion will be performed someday".
Class interface mismatch	The class interface (API) is not appropriate.
Incomplete library class	It is difficult to use existing library classes.
Parallel inheritance hierarchies	A sub-class should be created in another place of the class hierarchy when a sub-class is created.
Comment	There are detailed comments to explain the insufficient code.

C) Typical refactoring techniques

Refactoring techniques		Description
Method cleanup	Extract Method	When there are code snippets that can be grouped together, a separate method is created with a name that clearly expresses the purpose of the code.
	Replace Parameter with Method	If an object calls a method and passes the result to another method as a parameter, the receiver calls the method.
Function Transfer between Objects	Extract Class	If one class is doing the task that should be performed by two classes, a new class is created, and the related fields and methods are transferred to it from the old class.
	Extract Subclass	If a class has a function that is used by some instances only, a sub-class is created that manages the functions used by the instance only.
	Extract Interface	If several clients are using the same subset of the interface of one class, the subset is extracted as an interface.
Name	Rename Method	A name with a built-in type is changed to a name that can deliver the intent well without associating with the type.
Speculative Generality	Inline Method	If the main text of a method is as clear as the method's name, delete the method after moving the main text into the caller that calls the method.
	Collapse Hierarchy	Integrates the super class and sub-class if they are not very different.
Duplication	Replace Magic Number with Symbolic Constant	If there is a numeric literal that has special meaning, a constant is created with the name that reveals the intent and the number is replaced with a constant.
	Pull Up Field	If two sub-classes have the same field, the field is moved to the superclass.
	Pull Up Method	If multiple sub-classes have the method that performs the same task, the method is moved to the superclass.

X. Software Requirements Management

▶▶▶ Recent trends and major issues

The software requirements management process is the most basic and important in the entire life cycle of software development, and various techniques and tools are developed for the efficient management of requirements.

▶▶▶ Learning objectives

1. To be able to explain the concept and process of software requirements management.
2. To be able to explain the processes for requirements tracking management and change management.

▶▶▶ Keywords

- Requirements management process
- Requirements specification technique
- Requirements change management

Preview for practical business

Projects fail mostly because of inappropriate requirements management.

According to data from Standish Group, a global market research organization, 50% of IT project failures in overseas countries are related to requirements management, and 70-80% of the total project cost is caused by the failure of requirements management. The above data also apply to domestic projects. The main causes of these results can be summarized as follows.

- Inadequate definition of requirements/insufficient communication: Insufficient understanding of the requirements in the early development process due to limited resources and timelines, and incomplete dialogue between stakeholders.
- Lack of change management: When changing the customers' requirements or the development deliverables, its impact and relevancy are not considered.

[Source: Present and future direction of domestic requirements management market, The Electronic Times]

01 Requirements Management

A) Definition of requirements management

Requirements engineering can be largely divided into requirements development and requirements management. Requirements development consists in defining what to do, while requirements management consists in checking whether the planned and defined requirements are accurately reflected and in managing changes to the first requirements on a continuous basis.

B) Importance of requirements management

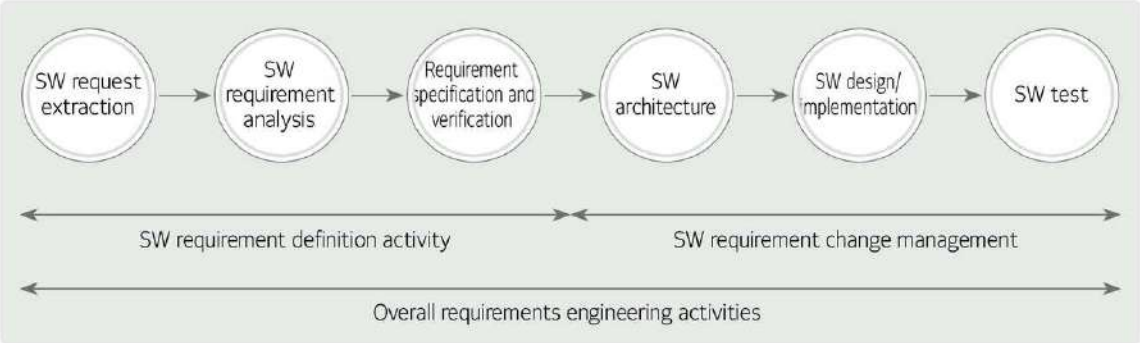
Proper requirements management should provide an effective means of communication among the stakeholders, prevent delivery delays and going over budget through systematic requirements management from the initial phase of the project, and execute and manage the user requirements specification.

C) Purposes of requirements management

Requirements management aims to satisfy the customer's requirements by understanding them from the customer's point of view and to produce high-quality software within a limited schedule and period.

D) Requirements management process

The figure below illustrates the requirements management activities in the entire software development life cycle.



[Figure 24] Requirements management activities

The table below describes the detailed activities and corresponding deliverables of the requirements management process.

<Table 35> Details of requirement management activities

Process	Description	Deliverables
Requirements extraction	• Defining the business requirements. • Identifying the participants. • Extracting the initial requirements.	Candidate requirement

Process	Description	Deliverables
Requirements analysis	• Modeling the candidate requirements. • Determining the priority of the requirements. • Discussing the requirements.	Agreed requirement
Requirements specification	• Defining the requirements specification standards; creating the specification. • Saving information related to the traceability of the requirements.	Formal requirement
Requirements verification	• Reviewing the requirements specification; verifying the terms; setting the baseline.	Baselined Requirement
Requirements change management	• Requirements change control, traceability control, version control (history management).	Consistent Requirement

E) Principles of requirements management

The principles of requirements management are as follows.

- Giving priority to requirements-based customer value/obtaining the stakeholders' consent to the requirements.
- Identifying the exact target of the required system (expectation management/scope management).
- Analyzing the impact of a change to the requirements by running the Change Control Board (CCB), and setting a baseline for each change step.

02 Requirements Specification

Requirements should be defined by specification techniques, and the correct specification can be defined only when the principles are considered.

A) Requirements specification techniques

Requirements specification techniques can be divided into formal specification and informal specification depending on the method.

<Table 36> Requirements specification techniques

Classification	Specification techniques	Description
Formal specification	VDM	• Vienna Development Method • State-based graphical specification method
	Mathematics-based technology	• Provision of a framework for development of the specification and systematic verification of the system
Informal specification	FSM	• Finite State Machine • Expressing state transition by input signals
	SADT	• Structured Analysis and Design Technique • Graphic-based structural analysis model
	Use case	• User-based modeling
	Decision Table	• Writing probabilities and cases for decision making
	ER modeling	• Representation of the entity relationship

B) Principles and main contents of requirements specification

The IEEE (Institute of Electrical and Electronics Engineers) explains the content and quality of the recommended software requirements specification and provides overview samples of various requirements. The specification principle can use the requirements shown below as the criteria for quality checks.

<Table 37> Requirements specification principles

Principle	Description
Verifiability	• The requirements specification should be verifiable
Modifiability	• The requirements specification should be modifiable.
Clarity	• The requirements specification should be presented to each stakeholder.
Accuracy	• The requirements specification should be accurately described.
Traceability	• The source and principle of the requirements specification should be traceable.
Consistency	• There should be no conflicts between the requirements.
Completeness	• All important details, such as functionality, performance, and constraints, should be documented.
Interpretability	• Requirements: Should provide consistency of interpretation.
Understandability	• The requirements specification should be easy for the stakeholders to understand.

The table below describes the main contents of the requirements specification defined as a standard by the IEEE. It can be referred to as the contents and details when creating a requirements specification.

<Table 38> Requirements specification

Contents	Description
1. Overview	• Overall description of the software • Purpose of the software requirements specification • Intended audience of the software requirements specification • Relationship with the software purpose, constraints, characteristics, profits, business strategy, and common goals • (overall software description)
1.1 Purpose	
1.2 Related teams/Persons concerned	
1.3 Development scope	
2. Overall description	
2.1 Project perspective	Describes the origin of the software, replacing existing systems, independent product status, etc.
2.2 Product functions	Describes the list of functions only. The full details are described in Chapter 4.
2.3 Characteristics of users	• Describes the users' characteristics (experience, educational level, disposition, etc.). • Describes user frequency by user.
2.4 Software operating environment	Describes the hardware environment to run software, OS, software environment, etc.
2.5 Decision and deployment	• Hardware constraints, memory constraints, specific technology use, multilingual support, security • Design conventions, programming conventions
2.6 Assumptions and dependencies	Assumptions that affect this software (Describes dependency on outsourced components, components of other projects)
2.7 Data requirements	Input/output data by function from the user's point of view
2.8 User scenario	Scenario from the user's viewpoint that can describe this system
3. Interface	
3.1 User interface	Sample screen image, standard buttons, standard error messages, keyboard shortcut, etc.

Contents	Description
3.2 Hardware interface	Protocol exchange, interaction between system hardware components and software products
3.3 Software interface	Defines the data or message that access this software from other software components.
4. Functional requirements	• Describes the detailed requirements for the functions listed in Chapter 2. • Purpose: Purpose of/reason for the function • Inputs: Data coming into the function (validation check, troubleshooting) • Outputs: Data coming out of the function (process error message, parameter range, form, shape, distance, volume output)
4.1 <Function Name>	
5. Non-functional requirements	
5.1 Performance requirements	Number of connected users, response time, file/table size, number of transactions occurred per unit time.
5.2 Security requirements	Data and communication encryption, access control, rights management, etc.
5.3 Software quality requirements	Availability, flexibility, interoperability, maintainability, portability, reliability, usability, etc. Describes the software quality requirements.
5.4 Business rules	Operation rules, standards when individuals/organizations do business using software. CBP (Current Biz Process) and FPM (Future Process Model)

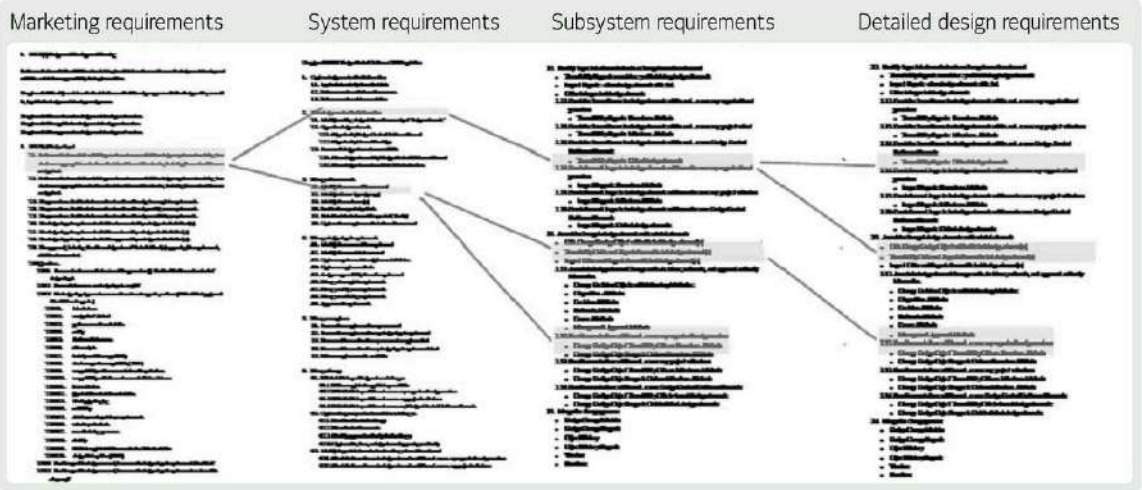
03 Requirements Change and Tracking Management

A) Overview of requirements traceability

Once the requirements baseline is established and the project starts, changes to the requirements will inevitably occur. Requirements change management is the process of formally controlling all changes that occur based on the requirements baseline. The consistency and integrity of changes can be provided by controlling and tracking formal changes to the requirements that arise in the course of executing the project. Errors occur while changing the requirements for the following reasons:

- Requirements errors, conflicts, and inconsistencies occur.
- The participants develop a deeper understanding and greater knowledge of the system
- Changes occur in the system environment and organization.
- Technical, time, and cost problems arise.

The above types of errors can be reduced by defining and configuring the traceability of the requirements. Tracking the requirements does not mean that each requirements document is traced, but rather that the requirements in each sentence of the requirements document are traced. The concept of tracing each sentence can be easily understood by referring to the figure below.



<Figure 25> Example of traceability by sentence

When traceability is given to each requirement in the development deliverables, as shown above, we can clearly understand how the marketing requirements are reflected in the system requirements, and which subsystems are allocated and reflected in the design. In addition, we can estimate the addition development M/M caused by changes to the marketing requirements requested by the user; understand how the upper-level requirements are affected when developers try to change some of the design requirements at their own discretion; and identify the requirements that were omitted in the intermediate steps.

The tracking of requirements is only effective when managed in units. The following values can be obtained when tracking management is performed for each requirement in all the document deliverables requiring tracking management.

- Reduces the need for re-working by preventing the omission of higher-level requirements.
- Raises the quality of individual requirements by securing traceability (clarity, traceability, testability, etc.).
- Analyzes the impact of changes.
- Enables effective collaboration between various teams/companies.
- Secures test coverage and improves product quality by securing traceability with test cases.
- Secures consistency between deliverables via the management of various changes.
- Increases productivity by reducing incorrect communication caused by changes.

XI. Software Configuration Management

▶▶▶ Recent trends and major issues

The importance of software configuration management is gradually attracting the interest of enterprises, as it is now recognized as an important factor that affects quality. The use of CASE tools for configuration management is regarded as important, and configuration management becomes an issue when implementing a large-scale project based on a distributed computing environment.

▶▶▶ Learning objectives

1. To be able to explain the concepts and activities of software configuration management.
2. To be able to use software configuration management tools.

▶▶▶ Keywords

· Software configuration, software configuration management elements, change control procedure
· Version management tool, configuration management tool

✚ Preview for practical business

Do we have to manage software configuration?

The story of a development team:
A certain project was commenced with a deadline of about three months, but its end was not visible even after two and a half months. Mr. Ko, the project manager, also in charge of the architecture, became increasingly anxious with every passing day. When the project was started, he thought that three months would be enough, given its scale. That is why he didn't create a separate management system - because four members seemed few enough to engage in face-to-face communication. However, about a month later he began to feel that the project had gradually got bogged down. Suddenly, a team member left the company for health reasons and all his tasks were stopped. Mr. Ko hastened to look for a replacement to fill the position. However, the departed member had not created any proper documents, so Mr. Ko had to select one member of his small team to train the new recruit for three days.

After that happened, he hurriedly began to list up the most immediate requirements and the design deliverables. Of course, listing those deliverables was not included in the project period, so he had to work overnight for almost a week to avoid delaying the project.

After that, the project seemed to go well to some degree. However, more serious problems occurred after about two months. After developing the major functions and frameworks, Mr. Ko was busy preparing for a demonstration for customers because it was due soon. However, his team was unlucky. A team member's computer became infected with a virus, and all the source codes and various scripts were erased in an instant.

The source of the calamity was the method of deliverables management. That is, the deliverables were not centrally managed. Instead, only documents were managed on the central file server, and the source code of each member was stored in the team's PCs during development. Until that happened, each developer was trusted to upload their source codes to the file server. The developer who lost all his source codes due to a virus infection was the least diligent of them.

The team members had to work overnight again for couple of days to update the latest source code uploaded to the file server. During that period, the customers' trust was wearing thinner and thinner. When the developed system began to be delivered to the customers, they made various requests, and the development team began to modify the source code on demand.

However, the software version of each customer varied as time went by. In the end, the development team was unable to track the customers' requirements and software versions. The source code was patched frequently, and bugs that did not exist before began to appear one by one. The team had to perform tedious debugging for each source code line almost every day.

If they had known and utilized such concepts as branching and merging of configuration management, baseline, baseline promotion, change tracking, and configuration audit, they would not have struggled with new bugs that appeared every day by systematically responding to requests for changes.
Further efforts are required to fully understand and utilize these concepts, but the effect is certainly worth it.

If you have ever managed to complete a software development project without configuration management, you must be very lucky. However, I think all of us must have experienced this difficulty at least once or twice. What if Mr. Ko's team had only started the project after establishing a software development process, even though it was simple, and established a configuration management plan according to the situation? It is quite probable that a ridiculous thing like this would never have happened.

In conclusion, configuration management surely helps software development. It takes more activities to understand and utilize those concepts, but the effect is certainly more than that.

[Source: Software changes determine quality - ZDNet]

01 Overview of Software Configuration Management

A) Definition of software configuration management

Software configuration management (SCM) is a series of activities developed to manage software changes during the software development process. Software configuration management is the task of finding and controlling the causes of software changes, checking that software has been properly changed, and notifying the relevant person of the result. It is applied to all stages of software development and is performed during the maintenance phase as well as the development phase so as to reduce the overall cost of software development and guarantee the minimization of risk factors during the development process.

Here, "configuration" generally refers to the program created in the process of software development, the program description document, data, etc. Software configuration management is a management technique designed to systematically manage various deliverables created in the software development process. That is, it manages actions related to the history of all component changes that occur from the software development phase to the maintenance phase.

The table below describes common problems that occur when software configuration is not (properly) managed.

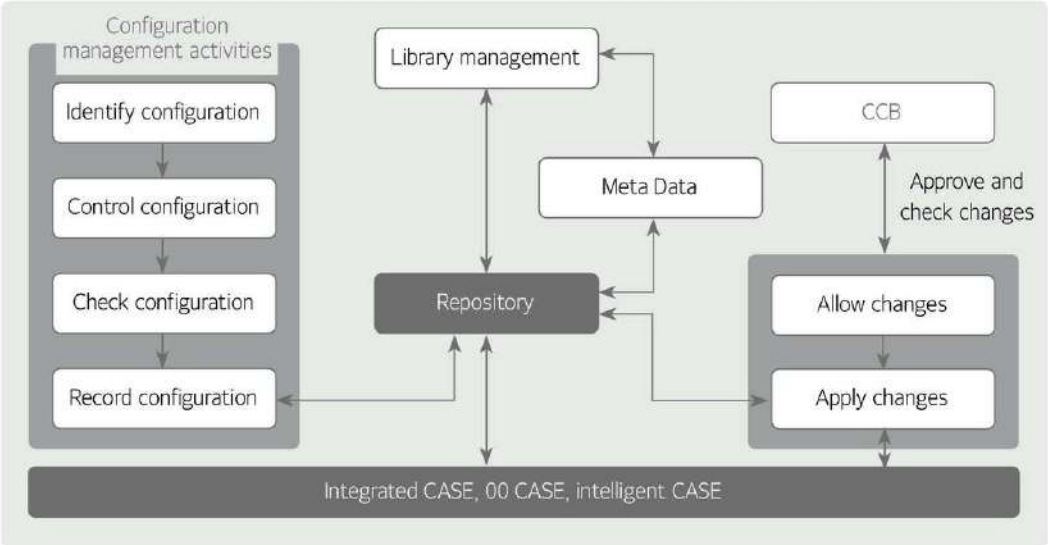
<Table 39> Background of the necessity of configuration management

Cause of the problem	Contents
Lack of visibility	Software has no visibility because it is intangible.
Difficulty in control	It is practically difficult to control software development because it is invisible.
Lack of traceability	It is difficult to trace the entire process of software development.
Lack of monitoring	It is difficult to manage a project continuously due to a lack of visibility and difficulty in tracking.
Ceaseless change	Ceaseless changes to the user's requirements.

02 Conceptual Diagram and Components of Configuration Management

A) Conceptual diagram of configuration management

The conceptual diagram below describes the entire activity and correlation of configuration management.



[Figure 26] Conceptual diagram of configuration management

B) Components of configuration management

The table below describes the major items of configuration management.

<Table 40> Components of configuration management

Component	Contents
Baseline	Technical control points of each configuration item. The standard point in time that controls all changes.
Configuration item	The default target that is formally defined and described in the software life cycle.
Configuration product	The tangible and realized target of configuration management that is formally implemented in the software development life cycle. Technical documents, hardware products, software products.
Configuration information	Configuration information = configuration item + configuration product.

03 Configuration Management Activity

A) Configuration management activity

Configuration management activities are composed of configuration identification, control, audit, and recording. The table below explains the details.

<Table 41> Configuration management activities

Activity	Contents
Shape identification	• Identifying a configuration management target and numbering the management items. • Goal of identifying a configuration item: To define a document structure in a clear and predictable way, and to facilitate tracking and management by recording information. • Contents of configuration item identification: Product/Various documents/Configuration item number.
Configuration control	• Reviewing and approving a software configuration change request and controlling it so that it can be reflected in the defined baseline. • Managing change requests/Controlling change/Supporting the operation of the configuration management organization and configuration control over development companies and outsourcing companies.
Configuration audit	• Means of determining the integrity of the software baseline. • Setting a software baseline successfully with a successful configuration audit. • When changing the baseline, checking whether it matches the requirement. • Verification, validation.
Configuration recording	• A function of recording the various statuses and execution results of software configuration and change management, and creating reports by managing the database.

B) Effects of configuration management

The effects shown in the table below can be achieved when configuration management activities are applied to software development and management.

<Table 42> Effects of configuration management

Item	Contents
Administrator's side (operator)	• Providing the standard for systematic and effective project management. • Ease of controlling a project. • Securing project visibility and guaranteeing traceability. • Presenting a baseline for quality assurance.
Developer's side	• Minimizing the impact of software changes, ease of management. • Software quality assurance technique. • Proper management of software changes. • Improving maintainability.

C) Considerations for configuration management

The following matters should be considered to ensure efficient configuration management.

- Form an appropriate operation organization, using specialized management tools.
- Continuous management and management standards are needed, along with a method of solving any accompanying problems.
- Tailor the level of configuration management properly for small projects.
- Set the configuration management items, and apply any changes based on a formal agreement.
- Change software that is in use very carefully.

04 Configuration Management Tools

Source code management is one of the important aspects of software development. A system is needed to save and manage the source codes for effective history management of the various versions, change management, and collaboration between the team members. The system is called the VCS (Version Control System) or SCM (Source Code Management) system.

Now let's review the history of the software configuration management system. At first, configuration management consisted of version updates using a shared folder in the local, based on a file system. After that, distributed management became popular as the client/server type management system based on the network began to be used. The table below shows the configuration management tool by type.

A) Configuration management tools

The table below describes the types, applied technologies, and features of the representative configuration management tools. Subversion and Git, representative open-source configuration management tools, and Microsoft's Team Foundation Server (TFS), a commercial tool, will be described in the next chapter.

<Table 43> Configuration management tools

Item	Tool	Remarks
Folder sharing	RCS, SCCS	
Client/server	Subversion (SVN), CVS, Perforce, ClearCase, TFS	Central repository

Distributed repository	Git, Mercurial, Bitkeeper, SVK, Darcs	Provides a distributed repository. It is advantageous when there are many offline development tasks.

B) Subversion (SVN)

Subversion is an open-source version control system. It is also abbreviated as "SVN" using the command used in the command-line interface. Subversion was first developed by CollabNet Inc. in 2000 to replace the limited CVS.

Currently, it is being further developed with developer communities all over the world as a top-level Apache project. Subversion is based on the server-client model. The server can run in a working computer or in a separate computer connected over the network. In the case of centralized version control systems like Subversion, multiple clients check-out and use files from the central server.

C) Distributed repository (Git)

Distributed version control systems such as Git copy the entire repository, instead of directly downloading the last snapshot by the client. If a problem occurs at the server, the server can be restored using the copy held by the client. In the case of a distributed repository, source codes can be saved in multiple servers or the local PCs of multiple developers, rather than in just one central server. Therefore, each server or PC becomes a source code repository. The source code saved in each repository is not the same version, but the code is stored in different branches. That is, branch versions A, B, and C can be saved in the server A, whereas branch versions A, C, and D can be saved in the server B. Therefore, there is no concept of a master version in the system itself because the branch versions in each repository are all different and the locations for accessing and retrieving the source code are all different.

D) TFS (Team Foundation Server)

Team Foundation Server (TFS) is a Microsoft product that provides various features, such as source code management (Team Foundation version control), reporting, requirements management, project management (agile software development, waterfall model), automated build, lab management, testing, and release management. Although TFS can be used as a backend for numerous integrated development environments, it is designed to offer the best benefits when used as a backend for Microsoft Visual Studio or Eclipse (Windows and non-Windows platforms). TFS is an integrated repository based on the SQL Server. TSF stores all information related to the software development activities of the development team - such as source codes, deliverables, development activities, etc. - to enable the team to collaborate more effectively.

XII. Software Maintenance

▶▶▶ Recent trends and major issues

IT companies spend 60% of their application lifecycle costs intensively in the software maintenance phase. Accordingly, they are more interested in maintaining software systematically and thus perform activities aimed at extending the actual use period of software through systematic management and at acquiring high-quality software.

▶▶▶ Learning objectives

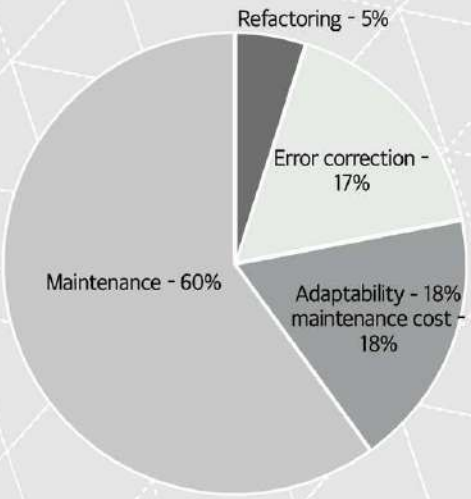
1. To be able to explain the concept and types of software maintenance.
2. To be able to explain the activities and processes of software maintenance.

▶▶▶ Keywords

· Concept of software maintenance
· Process of software maintenance

✚ Preview for practical business Do we need to maintain software?

Maintenance costs account for 60% of the entire software process cost, of which 60% is spent on functional enhancement. Of the remaining 40%, 17% is spent on error correction, 18% on maintaining adaptability to support other platforms, and 5% on refactoring.



Software maintenance is the continuous modification and supplementation of fully developed software that is currently in use, for the purpose of adapting it to changes in the environment, correcting errors, and improving its performance. To reduce maintenance, the system must be properly developed so that it can flexibly adapt to environmental changes.

After its development, software will be continuously modified and supplemented from the beginning of its use until its final disuse for various reasons. Software is a system that is directly connected to a person's consciousness or behavior.
Therefore, if the software is not well maintained, it will lose its vitality, that is, it will no longer be fit for use. Therefore, software maintenance is more important than initial development, and requires more effort and cost.

[Source: Doosan Encyclopedia]

01 Concept and Types of Software Maintenance

A) Definition of software maintenance

Software maintenance is the last phase of the SDLC (Software Development Life Cycle). It is an operation-oriented operational phase that extends the life of software, and comprises a series of tasks aimed at correcting software errors and improving its functions and performance by modifying the original requirements. Software maintenance is a series of software engineering works that eliminate defects, improve performance, and adapt software to the changed environment after its initial delivery to the customer.

B) Purposes of software maintenance

Software is mainly maintained to improve software performance, repair defects, port software to enable it to run in a new environment, and consists of a series of preventive measures including correction. As mentioned above, software maintenance is a very important phase in securing the efficiency of the software life cycle for the following reasons.

- Noticeable increase of maintenance costs that are higher than the existing development costs among the entire software development and maintenance costs.
- Increased administrative tasks such as documentation due to the complexity of the software.
- Emphasis is placed on the importance of management due to the use period being longer than the development period.
- Increased maintenance works due to the spread of package development, rather than outsourced development.

C) Type of software maintenance

Software maintenance can be classified according to reason, time, and target. Sometimes, a contract is signed according to these classification criteria.

<Table 44> Classification of software maintenance

Classification criteria	Type	Contents
Maintenance by reason	Corrective maintenance	Maintenance, handling, execution, implementation, and error identification due to errors.
	Adaptive maintenance	Processing to adapt to changes in the data environment and the infrastructure environment.
	Perfective maintenance	Maintenance for adding a new feature, changing, and improving quality.
Maintenance by time	Scheduled maintenance	Periodic maintenance.
	Preventive maintenance	Maintenance as a preventive measure.
	Emergency maintenance	If maintenance work needs to be approved later.
Maintenance by target	Data/program maintenance	Processing when necessary, such as data conversion.
	Documentation maintenance	Processing for changing the document standard or when necessary.
	System maintenance	System maintenance.

02 Software Maintenance Activities

A) Software maintenance procedure

Software is generally maintained in the following order.

<Table 45> Maintenance procedure

Maintenance procedure	Contents
Understanding current software	• Program structure analysis, variable/data structure, application field, business knowledge.
Requirements analysis	• Maintenance type, strategy establishment (modification/new). • Identifying targets such as the changed program.
Identifying and modifying the scope of impact	• Impact of software change on existing functions • Program change, modification.
Test/maintenance	• Performing document modification, configuration management, and maintenance.

"Requirements analysis", which is an important activity in the above maintenance sequence, aims to quickly understand how a maintenance request affects the current system. "Identifying the scope of impact" may be easy or require tremendous efforts, as described in the example, depending on how easy it is to maintain the software in the existing system. Ease of maintenance is a characteristic that indicates the level of difficulty in maintaining software, and there are the following types.

- Ease of use: A characteristic that indicates how easy it is to understand an application or program.
- Ease of modification: A characteristic that indicates how easy it is to modify an application or program.
- Ease of testing: A characteristic that indicates how easy the process is to show the accuracy of an application or program.

Therefore, if systematic activities are performed to improve ease of maintenance while developing software, as shown in the following, the requirements of users or customers can be continuously satisfied, and rapidly increasing maintenance costs can be handled economically even in the event of changes in the environment in the future.

- Analysis activities: Determining the customer's requirements and constraints, and proving the validity of the product to be developed.
- Standards and guidelines: Establishing various kinds of standards and guidelines to facilitate software maintenance.
- Design activities: Emphasizing and making use of clarity, modularity, and ease of change.
- Implementation activities: A standard structure and coding style should be applied.
- Supporting documents: The guide and test manual required for maintenance activities.

B) Types of software maintenance organizations

IEEE/EIA 12207 defines maintenance personnel as "an organization that performs maintenance activities". The term

"maintenance personnel" often refers to each individual who performs maintenance activities, so as to distinguish them from developers. Maintenance personnel can learn much from developers who have a deep knowledge of the software concerned. If maintenance personnel meet developers or participate in development from the beginning, maintenance can be reduced. Sometimes maintenance personnel become perplexed because the software engineer in charge has moved to another job or is out of contact. Maintenance personnel should take over and immediately support the developed product (e.g. code or document), and evolve and maintain the developed product throughout the entire software life cycle.

The table below shows the various subjects for each activity in the general maintenance phase. Each activity subject performs and bears the responsibility for the maintenance activity concerned. The maintenance management committee approves or rejects the maintenance of changes, and finally reviews and approves the applied change.

<Table 46> Subject by software maintenance activity

Phase	Activity	Activity subject
Request	• Filling in the MRF (Modification Request Form). • Making a CR (Change Request).	Users, customers
Analysis	• Classifying maintenance types, determining the severity. • Analyzing the content of the maintenance request, and the impact. • Prioritizing maintenance works.	Analyst
Approval	• Approving maintenance work depending on the content of the analysis. • Approving maintenance work.	Maintenance management committee
Execution	• Conducting maintenance on the maintenance target. • Creating a software change report (SCR). • Changing the related documents.	Maintenance personnel

The table below describes the types of software maintenance organizations. The "organization by application field" is the most frequently applied type - mainly to fields requiring knowledge about the application business.

<Table 47> Types of software maintenance organizations

Type	Contents
Organization by work type	• Analyzing user requirements, and designing, implementing, and testing a system. • Each team performs its separate roles and responsibilities. • The organization can be specialized in programming knowledge and skills, but incurs coordination costs between analysts and programmers.
Organization by application field	• Classifying organizations according to the application. • The organization has expertise in the development of applied knowledge.
Organization by life cycle	• Classifying organizations into development and maintenance. • The organization can be specialized in development and maintenance technology. • The cost of coordinating the development/maintenance organization is needed.

XIII. Trends of Open-Source Software

▶▶▶ Recent trends and major issues

The recent trend of software development is focused on the conviction that a company's competitiveness depends on how quickly it develops and releases software on the market. In this respect, time to service, not time to market, is becoming a key factor. A quick response using a development methodology that is reliant on proprietary software is not possible in this environment. As a result, software development and the necessity of using open-source software are increasing as several technologies can be used.

▶▶▶ Learning objectives

1. To be able to explain the concept and license of open-source software.
2. To be able to understand and use open-source software licenses.

▶▶▶ Keywords

· Concept of open-source software, open-source software license
· Compliance with the open-source software license

+ Preview for practical business

Do we need to manage open-source software?

As more open-source software is used, the need for license management is also increasing, because disputes over licensing can damage a corporate image and cause financial losses too. In particular, the use of open-source software seems to be spreading further in this era of the Fourth Industrial Revolution, which can also increase dispute factors.

As the source code of open-source software is opened, there are several benefits when developing a technology or product that uses it, such as reducing the development time and cost, and breaking technological dependence. However, though anyone can use an open-source license, users should follow the license terms when distributing it to a third party intending to use it for technology or product development.

Last year, a domestic software company called Hangul and Computer was sued for breach of copyright by Artifex in the U.S. on a charge of violating the open-source license. Artifex claimed that Hangul and Computer had violated the license when selling "Hangeul" programs equipped with "Ghostscript" (a PDF conversion library), which was developed and distributed by Artifex with a GPL. Artifex filed a lawsuit in the California court, and Hangul and Computer's request for its dismissal was rejected. In the end, Hangul and Computer had to pay the settlement money to end the lawsuit.

One official at the National IT Industry Promotion Agency said, "This incident is a case which shows how a corporate image can be damaged and financial losses incurred by unauthorized use of the GPL code. This case reminded us again of the importance of complying with open source software."

One official at the National IT Industry Promotion Agency said, "The incident was the case that tells us deterioration of corporate image and financial loss due to the unauthorized use of the GPL code. The case reminded us again of the importance of complying with the open-source software."

Disputes over open-source licenses are increasing because we did not introduce and manage the open-source license systematically due to the erroneous perception that it is free of charge. In addition, open-source licenses are complex as there are more than 2,000 types of license. According to a NIPA survey, 37% of domestic SMEs violated the license in 2016.

It was pointed out that we need to establish an open-source license management system quickly, because the use of open-source software seems to be increasing in the field of new technology such as the Internet of Things (IoT), artificial intelligence (AI), big data, cloud, etc.

It is essential that enterprises and public institutions using open-source software take countermeasures such as establishing governance and utilizing license analytics and verification tools.

[Source: iNews 24, "Misusing open source software can be nightmare. Using open source software is like walking on a tightrope]

01 Concept of Open-Source Software

A) Definition of open-source software

Open-source software is free software that can be used, copied, and modified by anyone while protecting the rights of the creator.

As the software industry began to develop, a move was made to impose restrictions on the use and modification of software based on the intellectual property rights and license agreement in 1980. To oppose to this move and protect the rights of free software use, Richard Stallman founded the Free Software Foundation (FSF) and launched the free software movement, which ultimately became the origin of open-source software.

Open-source software includes FSF free software in Korea. However, free software pursues the meaning of emphasizing users' freedom, while open-source software pursues the idea that the source code must be shared for joint research to make software "better".

The term "open-source software" was widely used at the time the Open Source Initiative (OSI) was formed in 1998. The OSI manages the promotion and certification of open-source software. The OSI has defined the minimum standard with which the license should comply (Open Source Definition, OSD), and awards an OSI certification mark to open-source software certified by the OSI.

Server software such as the Apache web server, PHP, MySQL, Hadoop, and Openstack, development tools such as Eclipse and gdb, and operating systems such as Linux are representative types of open-source software.

B) Definition of the open-source software license

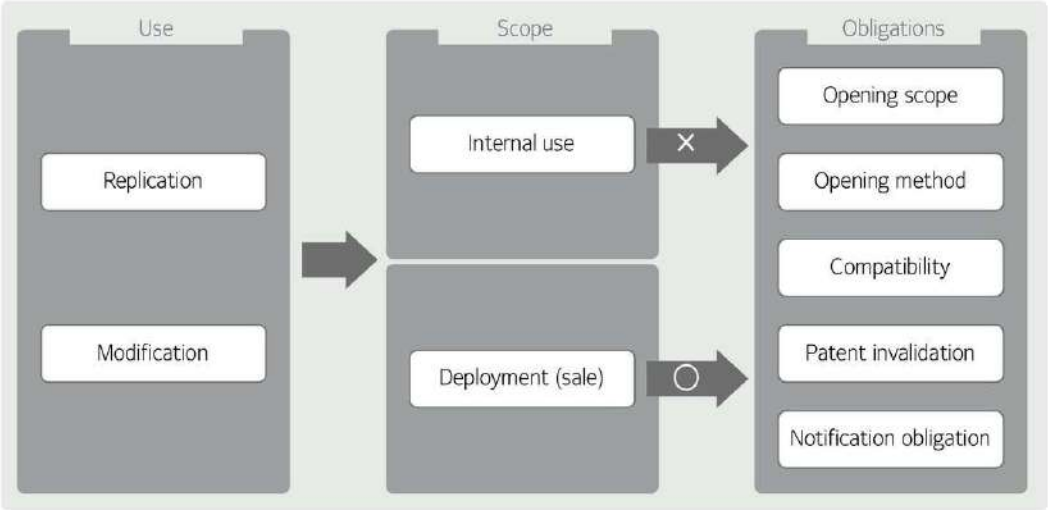
An open-source software license refers to an agreement made between an open-source developer and a user on the method of use and the scope of the conditions of use. Open-source software should be used by protecting and preserving the rights of the creator and complying with the conditions set by the open-source developer. If the user violates these conditions, the user must assume legal responsibility for violation of the license and infringement of the copyright.

Representative licenses include the General Public License (GPL), Lesser General Public License (LGPL), Berkeley Software Distribution (BSD), and Mozilla Public License (MPL). These open-source licenses have been created and imposed to force disclosure of the source code within the legal boundaries.

02 Open-Source Software License

A) Application scope of the open-source software license

We need to understand the scope of using open-source software before understanding the license classification. Obligations are not imposed when open-source software is used internally; rather, they are applied only when open-source software is distributed and sold outside.



[Figure 27] Obligations of the open-source software license

B) Comparison of open-source software licenses

Open-source software licenses have common characteristics such as free use, distribution without restrictions, source code acquisition, and source code modification. However, there are some differences with each license regarding the obligation to disclose secondary works again, and the combination of open source with proprietary software.

The GPL is the most stringent license, which specifies the condition that combining GPL code with proprietary software is not allowed. When a program is distributed under the GPL, which combines proprietary software code and the GPL code, the proprietary software code must also be opened.

① GPL (GNU General Public License) 2.0

GPL 2.0 is the license most frequently adopted for open-source software. It was created and distributed by the Free Software Foundation (FSF) and is very strict compared to other open-source software licenses. The main matters to be observed are as follows.

- When distributing software, the distributor should specify the copyright notice, distribution under the GPL, and the absence of warranty liability for GPL software.
- When modifying software or linking new software (both static and dynamic linking), the source code must be opened in accordance with the GPL.
- When distributing its own patented program, a distributor cannot receive royalties from users who use the program under the GPL.

② GPL (GNU General Public License) 3.0

The basic contents of GPL 3.0 are similar to those of GPL 2.0, but it has been supplemented with matters related to DRM (Digital Rights Management, technology and service to protect copyrights), software patent issues, and compatibility issues. The main matters to comply with are as follows.

• When including the source code under GPL 3.0 in the user's product or distributing it with the user's product, installation information should be provided with the source code.

The interests related to DRM, which are protected by the laws of each country, should be waived. If a contributor has improved the source code, that contributor should provide a license stating that his/her contribution is non-discriminatory and does not require any fee with regard to the patent.

③ LGPL (Lesser General Public License)

The LGPL enables users to use some libraries in a way that the source code can be used in a more eased form than the GPL. If a strict license such as the GPL is applied, commercial software developers will be reluctant to use the open-source library because they have to open the source code of the software that uses the library. In other words, the LGPL is designed to encourage the use of an open-source library from a strategic point of view. The main matters to comply with are as follows.

• When linking an application program with the LGPL library (both static and dynamic linking), developers do not have to open the source code of the application program concerned.
• The patent is handled in the same way as the GPL.

④ BSD (Berkeley Software Distribution) license

The BSD license is a representative open software license that does not require the developer to disclose the software source code. The scope of BSD license permission is wide because projects deployed with the BSD license have been implemented using the financial resources provided by the U.S. government. As the software was paid for with the taxes of U.S. citizens in advance, people can use or create the software in the way they want. The main matters to comply with are as follows.

• When distributing the software, copyright notice and absence of warranty should be specified.
• The source code can be used for commercial software without limitations because there is no obligation to disclose the source code of the modified program.

⑤ Apache license

The Apache license is applicable to all software of the Apache Software Foundation (ASF), including the Apache web server. Like the BSD license, it does not require disclosure of the source code. The main matters to comply with are as follows.

- The trademark right for the name "Apache" should not be violated.

• When distributing the software, copyright notice and absence of warranty should be specified.
• The code distributed under the Apache license can be combined with the code distributed under the GPL 3.0.
• The source code can be used for commercial software without limitations

because there is no obligation to disclose the source code of the modified program.

⑥ MPL (Mozilla Public License)

The MPL was developed to disclose the source code of the Netscape browser. It defines the scope of the source code to be disclosed more clearly. The modified part of the MPL code itself should be distributed under

the MBL again, but there is no obligation of disclosure if the modified part is combined with other codes or if new codes are written. The main matters to comply with are as follows.

- When distributing the software, copyrights, no warranty, and distribution under the MPL should be specified.
- The part that modified the MPL code should be distributed under the MPL again.
- When the MPL code and another code are combined, the source code for the combined program excluding the MPL code does not have to be disclosed.

C) Considerations when utilizing the open-source software license

Individual enterprises should clearly understand the advantages and disadvantages of using open-source software before using an open-source software license.

The major advantages are as follows:

- The initial development cost is low because the source code can be downloaded from the web, and modified and redistributed free of charge.
- Technology can be developed quickly and flexibly as the latest technical information, problems, and solutions are shared between communities.
- Interoperability between different software is guaranteed because the open format or protocol are mainly used.
- Open source programs are relatively stable compare to proprietary programs because many outstanding developers are participating in development.

The major disadvantages are as follows:

- Documents are not as systematic as those of commercial programs.
- The development roadmap is not systematic because the source code is developed on the web freely based on voluntary participation. To utilize open-source software, the future development plan of the software concerned needs to be considered.
- Generally, it is required that open-source software should be distributed without a patent royalty.

XIV. Trends of Software Development

▶▶▶ Recent trends and major issues

Recently, new technologies for improving quality and development productivity have been released continuously in the software industry due to the increase in demand for business reading based on the rapid development and efficient use of resources. To understand this trend, we need to understand the latest trends in development tools, programming languages, development frameworks, and software architecture - like the micro-services that support independent deployment and excellent scalability.

▶▶▶ Learning objectives

1. To be able to explain the technology trends of software development and programming languages.
2. To be able to explain the technology trends of the development framework and software architecture.

▶▶▶ Keywords

· Software development tool, programming language, development framework, software architecture

+ Preview for practical business

Do you need to make software development more efficient?

It has been reported that the work productivity of Korea is very low compared to that of other advanced countries in the global labor market. The Korean IT industry is no exception to the rule. The IT industry has been making efforts to increase efficiency in its own way. Development productivity should have been improved to develop a system that became vaster and more complex in line with technological advances. As a result of these efforts, numerous platforms and frameworks, such as RAD, agile, CBD, and OOP, and various CASE tools have been released.

Tools that can be used appropriately by keeping pace with the current era of rapid change have been continuously developed. The "issue of efficiency" is the main driver behind countless tools. Efficiency became important in terms of securing higher productivity, improving quality, and completing a given project within a set period. As the tools described above were released, higher productivity and quality in software development could be secured to some degree.

[Source: IT Daily, "Necessity of efficient development methods for SI projects with the implementation of the 52-hour workweek policy"]

01 Technology Trends of Software Development Tools and Programming Languages

A) Technology trends of software development tools

① Highlighting the cloud-based integrated development environment

Integrated development environment software supports all tasks related to programs and development, such as coding, debugging, compilation, and deployment, in one unified program development environment. It has been highlighted as a development tool capable of meeting the calls of open source and global IT companies for rapid development. This environment is evolving into a cloud-based one, such as Codenvy, GoormIDE, and Cloud9, which enables development regardless of the location and does not require the installation of an integrated development program.

② Integrating software development tools and ALM (Application Lifecycle Management)

Performance management tools, requirements management tools, issue management tools, etc. that support development were designed to manage specific purposes and specific steps. Since the introduction of the cloud, it has been evolving into a tool that encompasses development/test/operation due to the complexity of the infrastructure and the market environment, which demands ever faster IT services. The main purpose of the performance management tool was to prevent application delays and improve performance. Recently, such tools have been integrated with ALM, which detects changed codes and manages the changes, and supports build/distribution comprehensively.

B) Technology trends of programming languages

① TypeScript, an extended version of JavaScript

JavaScript is one of the most popular and powerful programming languages in the world. It is often used to make an interactive web page by adding some effects to a web page, creating a game, animation, or drop-down menu, and effectively controlling the web interface. This programming language is basically useful for front-end development and is used by 88% of all websites.

Recently, TypeScript has been highlighted as an extended version of JavaScript. TypeScript is an open source project created by Microsoft in 2012, and is used as the base language of Angular.

• Key features of TypeScript
- It includes the functions of JavaScript as it is based on JavaScript.
- It is a "static type" language that checks types at the time of compilation.
- It offers ease of compilation in all browsers and environments, making it highly useful.
- It enables easy coding and learning because everything is a class or component.
- It is applicable to large-size application programs.

② ava 9 release

Java is an object-oriented programming language and has become the most widely used language in web

development. The newly released Java 9 includes about 90 new features.

• Key features of Java 9
- Increased modularization enables optimization in small devices, and the scalability and performance of Java have been improved.
- The Ahead of Time (AOT) compilation function, which compiles the code before execution in a virtual system, reduces the startup time of the application program.
- REPL (Real Eval Print Loop) is supported, which enables the reception of feedback on the program before compilation.

③ Kotlin

Kotlin is a functional programming language developed by JetBrain, the creator of Android Studio. It quickly gained in popularity after Google adopted it as the official programming language for Android in 2017. Kotlin provides an optimal environment for application development based on Android, as it is simple yet offers excellent performance.

• Key features of Kotlin
- Compatible with the existing Android APIs, and an app can be developed using concise syntax.
- Very similar to Java, thus allowing Java developers to use Kotlin quickly and easily.

④ Swift

Swift was developed by Apple to support application development for its products such as iOS, macOS, watchOS, and tvOS. It was designed to support easier and faster development on its platform.

• Key features of Swift
- Easy to develop an app owing to the simple coding process, while providing excellent performance.
- Interactive and easy to understand.

02 Technology Trends of the Development Framework and Software Architecture

A) Technology trends of software development tools

① Angular4

Angular4 was developed by Google, and uses the JavaScript framework for maintenance. Angular4 is used for single-page application programs and responds to user requests quickly by extending the HTML DOM.
Angular is a single page application (SPA) framework, which means that a view page can be retrieved and used dynamically without having to change the address of the page when executing the web page.

• Key features of Angular4
- Created using TypeScript 2.1, which is helpful when making a small application.

- Supports program logic and the backend binding function, and is not a simple static web page.

② React.js

React is a JavaScript library created by Facebook to develop the Facebook web. Like Backbone.js and Angular, React provides a web front framework needed for developing an app. React is useful when building a large-scale application that changes web pages frequently.

• Key features of React

- Supports the writing of the markup code in JavaScript using extended JavaScript syntax, which enables the existing XML called JSX (JavaScript XML).
- Supports components in individual view units that constitute the user interface.
- Allows data processing using the Virtual DOM.

③ Express.js

Express is the web application framework of Node.js, which is small and flexible and provides optimal features needed to develop a web or application. Express supports fast web development for Node.js.
Node.js is a back-end server development language based on events, which uses the JavaScript engine of Google's Chrome V8.

• Key features of Express

- Enables processing of the dynamic view of HTML pages depending on the parameters passed to the template.
- Combines necessary functions only by supporting a middleware structure written in JavaScript code.

B) Technology trends of software architecture

① Microservice architecture

The microservice architecture allows independent development, build, test, deployment, monitoring, routing, and extension for each service, by dividing one large application into several small services and creating an individual data repository, and by creating an independent execution environment.
Prompt response to rapid business changes, deployment within a short period of time, easy improvement and modification are all possible.

• Key features of the microservice architecture

- Development productivity is high as a service is divided into small units and developed independently.
- Flexible deployment in units rather than deployment of the whole set.
- A system can be expanded elaborately by expanding the units independently.

② Docker

Docker is an open-source platform that automates the Linux container (LXC) technology, which provides an independent and isolated space like a virtual machine, for ease of use. A company called Dot Cloud (now Docker) has started providing technical support for the first time, while Google is preparing Docker configuration for enterprises by implementing an open source project called "Kubernetes".

Docker can create a virtualization environment that is much lighter than hypervisor-based virtualization, supports server virtualization, and enables the user to develop an application regardless of the development language or tool.

- Key features of Docker
  - Makes it easy to package, distribute, and manage an application developed using a container.
  - Files and set values needed for container execution are included using an image.
  - Allows simple and fast development configuration regardless of the infrastructure requirements.
  - Supports multiple cloud platforms.