# Foundations of Software Engineering

Tetsuo Tamai

# Contents

# Preface

The first edition of ”The Fundamentals of Software Engineering” was published in March 2004 with the tagline ”How to make products that are invisible and weightless”. Fortunately, it was well received, winning the Okawa Publishing Award that year, and was published five times. Since then, a certain number of copies have been shipped every year since on-demand publishing began in 2016, indicating that there is still strong demand. However, the world of software is changing rapidly, and it cannot be denied that some of the content is somewhat outdated.

During this time, trendy words such as digital transformation (DX) have been born, and the government will launch the Digital Agency in September 2021, but it is clear that software is at the core of both movements. As a technology, the application of artificial intelligence (AI) has spread and is continuing its third boom, but compared to the second boom in the 1980s, the term AI has become completely common. In parallel with this, and with overlapping methods and application fields, data science has been in the spotlight. Both AI and data science are realized through software, and there has been renewed interest in software engineering, which answers the questions of how to create and use such software.

As a result, we have decided to completely revise the 2004 edition of ”Fundamentals of Software Engineering” to keep up with the changing times. We have retained the characteristic of the previous edition, which systematically covered software engineering in general while keeping the overall number of pages down and adopting a compact structure, and have aimed to create a text that is concise and easy to read compared to similar English books.

# Chapter 1

# Software and Software Engineering

Software is an abstract description written in a programming language, but it has the mysterious property of acting directly on the real world through a computer. Software engineering, the technology that designs and develops such software as an industrial product, has some commonalities with traditional engineering such as mechanical engineering and electrical engineering, but also has some quite different aspects.

## 1.1 What is software?

### 1.1.1 The origin of the word software

The term software has become widely used, but it is often used to refer not only to computer software, but also to broadcast programs, recorded music and video, etc. In Japanese, the abbreviated term ”soft” is also widely used. This book deals with computer software, of course. However, with the development of the Internet and the digitization of music and video recordings, it is true that music and video ”software” and computer software are becoming very close. In the first place, one of the major characteristics of modern so-called von Neumann-type computers is that they handle the data to be operated on and the program that describes the operation in the same form, so it is natural that the boundary between the two is becoming unclear.

Who first used the word ”software” and when did it become widespread? It is clear that the word ”hardware” came first, and then the word ”software” was created.

According to the dictionary, ”ware” means ”crafted item” or ”product,” and is often used in conjunction with other roots, such as ”kitchenware” or ”ironware.” The word ”hardware” has long been used to refer to metal objects such as knives and weapons.

As noted in the preface to the first edition, the use of the word ”software” can be traced back to Charles Dickens in the 19th century, but of course it does not refer to computer software. Until relatively recently, the Oxford English Dictionary listed the first use of ”computer software” in 1960. Yale Law School librarian Fred R. Shapiro found an earlier usage in 1958 of the word ”software” in an article written by Princeton University professor John W. Tukey, published in the January issue of American Mathematical Monthly. Tukey is also known as the developer of the fast Fourier transform (FFT), but there is a theory that he is the creator of the word “bit”, and there is a fairly solid basis for this, so it would not be strange if Tukey created the word software. I have written in detail elsewhere about the origins of this word [158, 159].

Computer software is written in an artificial language called a programming language. Although it is an abstract description, it can directly affect the real world through a computer. In other words, software has a strange nature in that it is similar to works such as novels and papers in the sense that it is the result of creation through linguistic expression, but is produced as a practical industrial product because of its function of direct action on the real world. Therefore, the process of producing it combines the aspect of an act of expression through language and the aspect of designing and developing an artifact.

The result of a description in a programming language that runs on a computer is usually called a program. The difference between the word ”program” and ”software” is that the former refers to a set of symbolic strings that are to be executed by a computer, while the latter is used as a general term, such as when discussing hardware. Therefore, the

English word ”software” is an abstract noun and does not have a plural form, ”softwares.” Programs, on the other hand, can be counted, and there is a plural form, ”programs.” Even people in software-related fields who are not native English speakers often make this mistake, so be careful.[1].

### 1.1.2 Characteristics of Software

Software is an abstract description, but its basic nature of being produced as an industrial product gives rise to various characteristics.

**Software is invisible** As a result, no matter how much software is around us, people are not aware of its existence unless a malfunction occurs and the software does not work. Software has no physical substance, so it has neither weight nor shape. It does not decay like the word ”software” in Dickens’s time, nor does it rust or wear out like hardware.

The lack of physical constraints means that there are no external conditions that bind its creation, and it is essentially free. This allows software to become as large and complex as it wants. The only constraints are the processing power and memory capacity of the computer that runs the software, but these are only increasing. There is a phenomenon called ”Moore’s Law” regarding the increase in semiconductor integration. It was originally proposed in 1965 by Gordon Moore, one of the founders of Intel, and it meant that integration would double every year for the next 10 years. It was later spread to mean that integration would double every year and a half, but what is surprising is that this tendency to increase in a geometric series rather than an arithmetic series, regardless of the coefficient, has continued to the present day (Figure 1.1). Thanks to this, even if software becomes larger or heavier (here ”heavy” is a metaphor for the amount of work required to calculate), advances in hardware absorb it. In that case, the constraints on software development are the limits of the human ability to design, develop, and maintain it. In fact, as we will discuss again and again, this is the most serious problem.

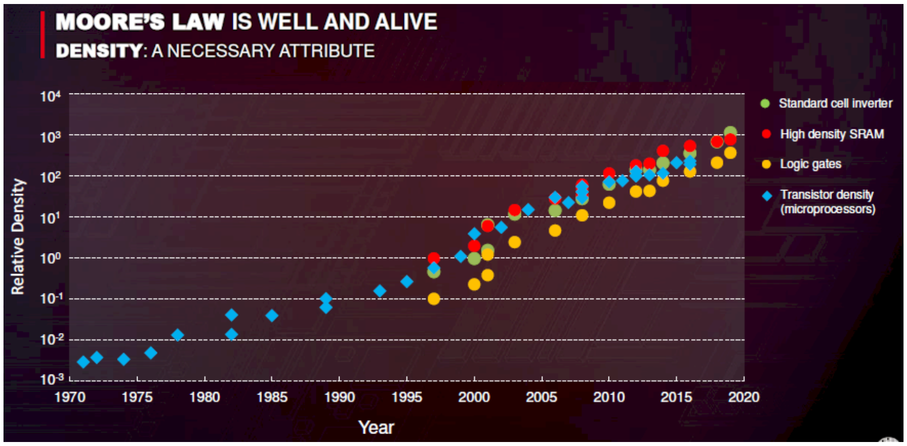


Figure 1.1: Moore’s Law (Source: [46])

**Easy to change** Software does not decay or wear out or get damaged, but it is easy to modify and change after it is created. In other words, it does not change naturally, but it is easy to change it artificially, and changes are often made after it is actually put into operation. Nowadays, when all devices are connected to the Internet, changes can be easily made via the Internet, and it is even possible to set it to automatically perform version upgrades.

**Differences in the meaning of maintenance** As for the task of software maintenance, we will discuss it again in Chapter 14, but its meaning is different from that of hardware maintenance, and this is due to the inherent characteristics of software. Since hardware deteriorates over time, maintenance of hardware means maintaining the initial functional level, but as mentioned above, software does not change due to wear and tear, so its maintenance can be corrective

[1] Similarly, the word ”information” does not have a plural form, ”informations.” This mistake is also sometimes seen, so be careful.

maintenance to fix defects that were latent at the time of shipment, adaptive maintenance to respond to changes in the surrounding hardware and networks, or functional expansion maintenance to meet user requests. In either case, these tasks are not usually called maintenance for hardware.

**All fields are covered** Software is used in all industries and is incorporated into all products. However, bank online systems, On the other hand, banking systems have been created independently by the systems departments of banks and engineers at manufacturers that supply computers to banks, engine control software by automobile manufacturers and engineers specializing in embedded software for automobiles, and game software by engineers and designers at game manufacturers, so there has been little interaction between these industries. As a result, they have tended to be created without much consideration of the perspective of general software engineering technology.

In this way, software has the advantage of flexibility that differs from hardware, so things that were previously created as hardware are now being realized with software one after another. For example, when making a phone call, you used to turn the dial with your fingers, but now you press a button or an icon on the screen. When changing channels on a TV, you used to turn a rotary lever by hand, but now you press a button on a remote control. The mechanical parts are being switched to control by digital signals. The operation and movement of cars have also become increasingly digitally controlled as they have become EVs. It is software that processes the digital signals and generates control signals as output based on them. This does not mean that the mechanical parts completely disappear, and even in digital cameras, opening and closing the shutter and moving the lens position are mechanical operations. However, software is responsible for processing the digital data obtained from the sensor that captures the optical signal and sending digital control signals to the actuator, which controls the operation.

The term firmware well represents the softening of hardware. Firmware means that it is not as hard as hardware, but not as soft as software. Firmware is software built into devices to control hardware, but it is written into LSI such as ROM and cannot be easily rewritten like normal software. However, it is essentially software, and is fixed in LSI for specific purposes and to achieve high processing speed. Furthermore, by using flash memory, it is possible to make changes after shipment, and the gap with software is narrowing further.

### 1.1.3 Types of software

Since software has permeated various fields, there are many different types of it.

#### Information systems and embedded systems

The most representative types of modern software would be information systems and embedded systems. The former refers to systems used by companies, government agencies, and service organizations to provide information services to the outside, as well as internal information management and decision-making support. Traditionally, these have been called data processing, clerical work, or business applications, but these days many people prefer the term enterprise systems. Many such information systems are now being created as web-based applications.

On the other hand, as we have already mentioned, embedded systems are systems found in cars, home appliances, measurement and control equipment, and so on. Recently, systems have been embedded as chips into all kinds of ”things,” and they are connected to the Internet, creating a situation known as the “Internet of Things (IoT).”

There is also software that does not fall into these two categories. For example, the control systems that run steel plants, nuclear plants, and heavy chemical industry plants cannot be called embedded systems.

The ALPAC Committee report changed the focus of AI. The ALPAC was a committee that aimed to evaluate machine translation in particular. At the time, machine translation was attracting attention as an AI technology because of the high demand for translating Russian documents into English during the Cold War between the United States and the Soviet Union. The report concluded that machine translation was more expensive, inaccurate, and time-consuming than human translation, which led to the National Research Council (NRC) halting its budget. This ushered in the so-called ”AI winter.”

After that, AI booms occurred in 30-year cycles. The second boom occurred in the 1980s. Edward Feigenbaum’s cleverly named ”knowledge engineering” came into the spotlight, and it was emphasized that AI should be applied to practical problems. To achieve this, it was necessary to collect the knowledge of experts, create a knowledge base, and develop an expert system that utilizes this knowledge. The industrial sector also competed to join the trend. In Japan, a national project to develop a ”fifth generation computer” began in 1982 as a 10-year plan, with the goal of creating hardware and software suitable for AI. This trend in Japan also inspired the United States and Europe, and competing projects were launched.

A little later than this second AI boom, interest in neural networks was revived. The history of research into artificial neural networks can be traced back even further than AI. The neural network model proposed by McCulloch-Pitts was proposed in 1943, and modern models still inherit the basic concept from it. In the late 1950s, Frank Rosenblatt proposed the perceptron model, which attracted attention, but in 1969, Marvin Minsky and Seymour Papert published a book called *Perceptrons*[92], in which they pointed out the theoretical limitations of the perceptron, which resulted in a slowdown in research into neural networks.

In the 1980s, interest in neural networks was revived by backpropagation, but this was essentially a rediscovery of backpropagation, which had in fact been invented and published in a paper by Shunichi Amari in the 1960s.

The third boom in AI in the 2010s came from neural networks, which were a separate movement from AI in the 1980s. The backpropagation method of the 1980s overcame the limitations of perceptrons by making the discriminant function nonlinear and stacking the network layers in multiple stages, but faced the problem that the accuracy could not be achieved because calculation errors accumulated when the network layers were multi-layered. By solving this problem with various ingenuity, machine learning was applied to benchmark test problems for image recognition, and it attracted public attention by achieving remarkable success. The method is called ”deep learning,” and there is also discussion that it is similar to the learning of actual neural networks that takes place in the human brain.

Its applications have expanded, and it has achieved results such as beating the world’s top professionals in the game of Go, which has been considered difficult because the search space is much larger than that of chess or shogi, and has produced results that are easy to understand for the general public, such as face recognition, cancer detection from MRI images, and natural language processing (question answering and translation).

**AI software development languages** The programming languages used in AI software development have had a major impact on programming and software engineering.

**Lisp** John McCarthy created the language Lisp in 1958, and it is still one of the programming languages born in the 1950s, along with Fortran and Cobol. Since then, it has continued to be used mainly in the AI field, while producing successor languages such as Common Lisp and Scheme.

**Prolog** Prolog is the language that symbolizes the second AI boom in the 1980s. While Lisp is a functional language, Prolog is a logical language. In Japan’s Fifth Generation Computer Project, languages such as GHC and KL1 were developed based on Prolog, which allow the description of parallel processes.

**Python** Python is not a language made for AI, but since neural networks are at the center of the third AI boom and Python is suitable for science and technology applications such as matrix calculations, a large number of Python-based libraries such as Scikit-Learn and PyTorch for machine learning were created. As a result, it became the language that symbolizes the third AI boom.

**Characteristics of AI software** As for the characteristics of AI software, it is natural that it has many things in common with other software. On that basis, the following points can be mentioned.

- It is difficult to judge whether behavior is ”correct”. In the first place, the definition of AI was once thought to be a technology that realizes intellectual abilities on a computer that humans are good at but machines have difficulty with. For example, mathematical processing was once considered a branch of AI, but as superior mathematical processing systems based on clear algorithms were developed, it fell out of the scope of AI. Due to the nature of AI, the behavior of AI software is often not explicitly predicted from the specifications.
- Among AI, systems that use machine learning consist of two stages: the learning stage and the application stage using the learning results. If there is a problem with the operation, it is necessary to determine whether the problem is in the learning process or the application process. The learning process also depends on the nature and size of the training data, and the number of epochs (number of repetitions) and the size of the mini-batch (a unit into which data is divided), which are hyperparameters, must be determined through experimentation and experience.

- Related to the above two points, the reproducibility of software execution is low. In a sense, this may be something to aim for if it is to be similar to human intellectual behavior, but it poses difficult problems in terms of software verification. At the same time, it is pointed out that there is a problem in that explanations of why actions or decisions were made are not given to people, or if there are, they are vague.

**AI and Data Science** The keyword during the second AI boom in the 1980s was ”knowledge.” During the third boom, the keyword became ”data.” In this era, the terms ”big data” and ”data science” were widely discussed.

What is the difference between data and knowledge? Tamito Yoshida gives the following somewhat difficult definition[149].

- Data = A set of non-cognitive symbols in a transmission system
- Information = Single-purpose data that determines the premise for a decision
- Knowledge = Durable data that can determine the premise for a decision

The relationship shown in the figure1.2 by Makoto Nagao is perhaps easier to understand[168]. In short, information is data abstracted, and knowledge is information that can be shared more widely and used repeatedly.

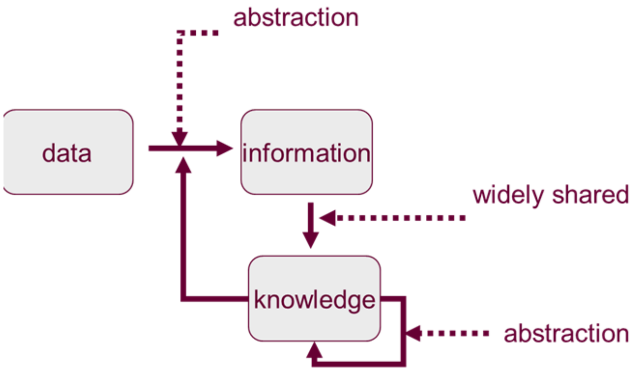


Figure 1.2: Data, Information, and Knowledge (Source: [168])

In the 1980s, AI attempted to realize intellectual capabilities such as judgment, diagnosis, and design using a knowledge base integrated in a rule-like format. The problem was how to acquire that knowledge, and one effective method was to infer it inductively from data. However, in the 2010s, the era of data science, it can be said that a more effective method is to use raw data itself for machine learning, rather than aggregating data and converting it into knowledge.

## 1.1.4 Who makes it?

### Information industry

Software is often made by individuals, but much of the software used daily around the world is developed by organizations. A large part of the information industry is made up of manufacturers that produce and sell computer hardware, related devices, and communication equipment, and the software industry that exclusively develops and provides software. The annual sales of Japan’s software industry in 2019 was approximately 22.3 trillion yen. This figure is captured in the Economic Structure Survey (Survey B) conducted annually by the Ministry of Economy, Trade and Industry (MITI), and is the total of the main business sales of approximately 38,000 businesses in the software industry, information processing and provision service industry, and Internet-related service industry that were surveyed[1].

However, software specialists are not the only companies that develop software. Computer manufacturers also develop large amounts of software as products and for customers. In addition, in industries other than computers, such as manufacturing, finance, distribution, and transportation, there are cases where software development is not outsourced but is developed in-house. Furthermore, software is produced to a greater or lesser extent in a variety of organizations, including universities, research institutes, government agencies, foundations, associations, and non-profit organizations. However, in Japan, most companies and government agencies tend to outsource development rather than creating in-house systems development organizations. In comparison, in the United States, when a need arises for a large-scale system development, U.S. companies and government agencies will organize a team to carry out the development, gathering personnel from outside if necessary[165]. When the development project is completed, the in-house team may be disbanded, but since society is highly mobile, engineers will move on to look for another project. In Japan, in many cases, the organization seeking outsourcing selects a prime contractor to take on the entire system development. Such companies are called system integrators or SI businesses (the Japanese-English term SIer is also often used). A hierarchical structure is created in which the prime contractor divides the work and gives it to a subcontractor, who in turn gives it to a sub-subcontractor. This structure follows the traditional structure of the IT industry and its original model probably came from the civil engineering industry.

### Software Engineers

Software developers are also called programmers, but in Japan, in particular, there is a tendency to call those who are given detailed design specifications and engage in the task of writing program code programmers, and to distinguish them from engineers who perform higher-level analysis and design. For this reason, people who are engaged in software development are often called software engineers, avoiding the term programmer. One clue to estimating the total number of software engineers in Japan is the figures captured in the Economic Structure Survey (Survey B), mentioned above. The simple total number of employees in the software industry, information processing and provision service industry, and Internet-related service industry surveyed in the survey comes to 1.16 million. This does not include software engineers who are present in large numbers in other industries such as manufacturing and finance. Estimates that take this into account have been made, and according to these, the number is 1.25 million as of 2020[162].

What kind of knowledge and skills are required of software engineers? In the United States, the body of knowledge for software engineering, abbreviated as SWEBOK (Software Engineering Body of Knowledge), has been created by the Computer Society of the IEEE. The latest version is V4.0[140].

Furthermore, software engineering has been taken up as a major field in the activity of creating a standard curriculum for computer science. In the United States, the IEEE Computer Society and ACM have jointly created a standard curriculum for the information field. Previously, IEEE and ACM developed it independently, but the first joint result of the two parties was CC'91, published in 1991. It was subsequently revised to CC2001 and CC2005, and the latest version as of 2021 is CC2020. However, since CC2001, in order to cover a wider range of this field, it has been divided into five fields: computer science (CS), information systems (IS), software engineering (SE), computer engineering (CE), and information technology (IT), and standard curricula for each field have been formulated, and the timing of revisions has not necessarily been synchronized. On the other hand, the CCxxxx report is intended to set an integrated policy for the whole. As of 2022, the latest edition in the software engineering field is SE2014[63].

In addition, in English-speaking countries, texts such as Ian Sommerville[122] and Roger Pressman[109] are relatively popular as standard textbooks for software engineering and have been reprinted many times. The topics and arrangements covered in them are more or less the same between the two, providing an overview of the knowledge and skills that software engineers should have. Another encyclopedic work is the Handbook of software engineering edited by Sungdeok Cha, Richard Taylor, and Kyochul Kang. I am also responsible for the Software Paradigm chapter in this book.

In Japan, the Information Processing Society of Japan has created a standard curriculum based on the US CC series, and the results have been published as reports. The latest version is J17, which includes software engineering as an independent volume.

The Information Processing Engineers Examination, a national examination conducted by MITI, Trade and Industry and administered by the Information-Technology Promotion Agency (IPA), has 12 categories, including Information Technology Engineer, Applied Information Technology Engineer, and System Architect, and evaluates the abilities that should be possessed in each technical category.

MITI has also created and published the IT Skill Standard (ITSS), an index that clarifies and systematizes the skills required to provide IT-related services. As shown in Table1.1, the job types are classified into 11 categories, such as "consultant," "project management," and "IT specialist." While the kanji compound "job type" is an everyday

Table 1.1: ITSS Jobs

| |
|---:|
| Marketing |
| Sales |
| Consultant |
| IT Architect |
| Project Management |
| IT Specialist |
| Application Specialist |
| Software Development |
| Customer Service |

word, the individual names are all impressively written in katakana, indicating they are foreign origin words..

Each job type has a specialization, and there are a total of 35 specializations. For example, the specializations of IT specialists are ”platform,” ”network,” ”database,” ”application common platform,” ”system management,” and ”security.” The names of the specializations are also mostly written in katakana, but not all of them are.

Seven levels, from level 1 to level 7, are defined for each specialization as an indicator of achievement. At level 3, the person is expected to ”complete all required tasks independently,” and at level 4, the person is expected to ”establish a specialized skill field as a professional and independently lead the discovery and resolution of work-related problems by utilizing their own skills.” Level 5 is a ”high-end player within a company,” level 6 is a ”high-end player within Japan,” and level 7 is a ”player with global competency.”

### Certification of IT Engineer Qualifications

For ITSS levels 1 to 4, there are corresponding IT Engineer Examination subjects, and by obtaining them, the technical level of the engineer is certified. However, since level 4 requires a track record of work and professional activities, there are some aspects that cannot be determined by the exam alone. The system that certifies the qualifications of advanced IT engineers from levels 4 to 7 is called the ”Certified Information Technology Professional (CITP) system,” and has been established and operated by the Information Processing Society of Japan. [154]. CITP is an abbreviation for “Certified IT Professional.”

It is aimed at ITSS levels 4 and above, but since there is currently no distinction between levels 4 to 7, it is recognized that the qualification is ”level 4 or above.” Although the certification is for individuals, since many companies have their own in-house certification systems, the system itself is reviewed, and if it is certified to meet the established standards, the company’s certified personnel are automatically certified. This method has contributed to increasing the number of CITP holders, and as of April 2021, the cumulative number of holders has exceeded 10,000.

Each country in the world has a similar certification system, and in today’s world where industrial activity is globalized, it is desirable to recognize the mutual equivalence of these qualifications, if possible, and promote international division of labor and human resource exchange. To this end, the International Federation for Information Processing (IFIP) established an organization called IP3 (International Professional Practice Partnership) to promote international mutual certification of qualifications. The Information Processing Society of Japan is also a member of IP3, and CITP is certified there, so it can be said to be a qualification that is valid worldwide.

As a qualification, the nationally recognized professional engineer system has a longer history. Professional engineers are divided into several specialized fields, one of which is the ”information engineering field.” CITP has had a system for updating its certification since its inception, and the certification is valid for three years. It is necessary to take the exam again when renewing it. Originally, there was no system for Professional Engineers, but in recently, a system for updating was newly introduced. In this trend, professional engineers in the field of information engineering can apply for CITP certification, and for those who have CITP some of the Professional Engineers exams are exempted, so that both can get benefit from each other.

### Software production system

Some of the software that is widely used around the world is developed by genius programmers who are called hackers, either alone or in small groups. Examples include Unix, WWW, Google, and Emacs. Visicalc, that initiated a genre of spreadsheet, was created in 1979 by Dan Bricklin and Bob Frankston, students at Harvard Business School, and the functions realized by Visicalc are almost the same as those of Microsoft Excel, the mainstream spreadsheet software today.

In Japan, there are people who deserve to be called hackers, such as Yukihiro Matsumoto, who created the programming language Ruby. Ikuo Takeuchi, a hacker himself, has been promoting the ”MITOU (hidden or unexplored) Program" since 2000, which aims to discover genius programmers. This project has been carried out under the Information-Technology Promotion Agency (IPA), and since 2008 has been renamed the "MITOU IT Talent Discovery and Development Project". Every year, the project accepts applications from the public and selects and grants outstanding individual software development proposals. From these, about 10-30 people are recognized and awarded as super creators. This has led to the "discovery" of many genius programmers. On the other hand, software produced by companies and other organizations is usually developed using a set of tools prepared in a standardized development environment under a clear work process, with standardized document formats for requirements specifications, design specifications, test cases, etc. A typical system for this is called a "software factory," and in Japan, Hitachi, Toshiba, and other companies have been known to use this name since the 1980s. The term "software factory" itself had been used in the United States before, but after the publication of a book by Michael Cusumano of MIT's Sloan School of Management[38], which introduced Japanese software factories in the United States, Japan's production system became known as a success story. In the early 1990s, the United States was swept by a tone that, although unthinkable now, would be beaten by Japan in the software industry, following steel, automobiles, and semiconductors. This was before the appearance of Amazon, Google, and Facebook.

Even though it is called a software factory, it is different in style from, for example, manufacturing work on an automobile assembly line. The software production process is closer to the design work of automobiles as products and the process of manufacturing them, rather than corresponding to the manufacturing process of automobile production. The production of the final software product is merely a matter of copying onto a medium such as a CD, and even this copying work is no longer necessary now that download distribution via networks has become mainstream.

However, the image of software development sites, especially in large Japanese companies, may be that of desks lined up in a large room, with terminals placed on each desk and each software engineer working in parallel while staring at the terminals (however, for better or worse, this image is changing due to the promotion of telecommuting caused by the COVID-19 pandemic). There, the development process is planned, and the work content at each step of the process and the format and content of the output deliverables are also clearly defined. This type of production system is efficient when the target system is large-scale and complex, but there is experience in developing similar systems in the past, or a detailed design has been achieved and the overall picture of the system to be developed is clear. However, it is not very suitable for developing software that is innovative and has never been done before, or that requires exploring various possibilities through trial and error.

As a kind of intermediate between this type of factory-style development and hacker-style development, there exists the agile process to be described in Chapter 2, "Software Process."

## 1.2 The Significance and History of Software Engineering

### 1.2.1 What is Software Engineering?

How to create such software is a major challenge for modern society. The purpose of this book is to explore the process and methods of creating software while focusing on the unique properties of the software. In particular, software engineering is a perspective that emphasizes that software is an artificial structure and regards it as engineering for constructing it.

The standard definition of software engineering is the following description from IEEE Std 610-1990.

> "Software engineering.
> (1) A systematic, controlled, and quantifiable method for developing, operating, and maintaining software.
> In other words, the application of engineering to software.
> (2) Research into methods such as (1)."

The term "software engineering" gives the strong impression of being a field of study that is researched and taught in the engineering departments of universities, just like mechanical engineering or electrical engineering. That is correct, but the English word engineering is a verb and has the connotation of production or manufacturing. So reengineering means rebuilding, and reverse engineering means deriving a design from a product in the opposite direction to the normal manufacturing process. Therefore, it is important to be aware that the concept of software engineering also has the connotation of software manufacturing.

Of course, software engineering is a latecomer in engineering. Compared to mechanical engineering and electrical engineering, which are representative of traditional engineering fields, the naming of software engineering, which directly indicates the software that is the subject of engineering development, may be closer to mechanical engineering than electrical engineering. Just as mechanical engineering is the technology of building and operating and maintaining machines, software engineering is the technology of building and operating and maintaining software. On the other hand, it is not appropriate to define electrical engineering as the technology of building and operating and maintaining electricity.

On the other hand, the fact that the subject of software is abstract and does not have the physical properties of electricity, let alone machines, is a major feature that distinguishes it from general engineering. Due to its abstraction, software engineering is close to computer science and mathematics as sciences. Furthermore, software development is similar to linguistic expression in that it involves writing in a programming language and describing requirements and design, and therefore has a close relationship with the humanities. In fact, books that explain how to write good programs are very similar to ”how-to-write” books [77].

Software engineering has a common nature with other engineering fields, and this can be seen from the fact that it shares the following tasks.

1. Modeling
   The problem and the type of system to be created are not clearly given from the beginning. It is important to have the skills to analyze the target domain, discover the problem, accurately grasp the user’s requirements, and model them.

2. Specification
   In engineering, writing proper specifications is a major prerequisite for development work.

3. Design
   The core of engineering is design technology.

4. Verification
   It must be verified that the system that satisfies the specifications has been created.

5. Maintenance
   A system is useless if it is created as is. Efforts are needed to maintain the system and adapt it to changes in the environment.

6. Organization
   A system is usually not small enough to be created by one person, so management techniques are needed to develop it by a team.

In fact, software engineering borrows many terms and concepts from existing engineering, including the above-mentioned ”model,” ”specification,” ”design,” and ”maintenance.”

The most important characteristic of engineering is the creation of artifacts as products. If one person creates software and uses it oneself, there is no need for ”engineering.” However, software that is provided to users and the market as a product is extremely difficult to create by one person, and must be created by an organization. However, due to its high level of abstraction, software is similar to mathematical theory, music, literature, art, and other works. It can be said that the task of developing such a thing by a large organization is something that humanity has never experienced before.

### 1.2.2 History of Software Engineering

#### The Birth of Software Engineering

The development of software began almost simultaneously with the birth of modern electronic computers. Nevertheless, in the case of the first electronic computer, ENIAC (1946), programming was done by a primitive method of rearranging wires. There is some disagreement as to whether the credit for inventing the stored-program computer design goes entirely to John von Neumann, but as the name von Neumann computer is still in use today, the impact of the ”First Draft of the EDVAC Report” written by von Neumann in 1945 on subsequent computer development is great. The first program written for a modern computer is included in the appendix of the first draft.

The first commercial computer, UNIVAC, was delivered to the U.S. Census Bureau in 1951, and since then computers have come to be used in a variety of fields, and many programs have been developed. IBM's System/360, released in 1964, increased the usefulness of software through its versatility and compatibility from high-end to low-end models. Its operating system, OS/360, was already a huge piece of software even by today's standards, with 5 million lines of assembler source code.

Thus, the social demand for software increased, but the production technology and systems for supplying it could not keep up with the demand, so development schedules were always delayed and the quality of the software produced was low. This situation became evident in the latter half of the 1960s.

To address this problem, the term software engineering was born, and it is possible to pinpoint the time when it became recognized as a field. It was October 7, 1968, the day the first conference named Software Engineering was held in Garmisch, Germany. The history of software engineering can be said to have begun from this date. Other engineering fields, such as mechanical engineering and electrical engineering, have long histories, so it is difficult to pinpoint birthdays in this way. This conference was organized by NATO's Scientific Committee, and the approximately 50 participants were all from Europe and North America. The same NATO conference was held in Rome, Italy the following October, 1969, with approximately 60 participants from Western European countries.

Considering the historical context in which it was born in 1968, it is significant that it was four years after IBM's System/360 and its operating system, OS/360, were announced in 1964, as mentioned above. With the arrival of the System/360, the demand for software development for application systems increased dramatically. Programming languages such as COBOL, FORTRAN, LISP, and ALGOL had been created by around 1960, and COBOL and FORTRAN in particular were widely used as languages for software running on general-purpose computers.

The organizers of the 1968 NATO conference stated that its objectives were to discuss how to ensure the reliability of information systems, which are increasingly linked to the core activities of modern society, how to deal with the difficulties of meeting deadlines and specifications in large-scale software development projects, and how to educate software engineers. These concerns are still relevant today.

This conference loudly proclaimed the birth of software engineering, and was also a forum for discussing how to deal with the dark reality of the gap between what was expected of software and the actual development situation, which weighed heavily on society. The term ”software crisis,” which was later used as a keyword derived from this conference, symbolizes this.

### The 1970s: The Age of Structuring

Although the awareness of the need for software engineering was shared at the NATO conference and the basic concepts that make up software engineering were largely established, it was not until the 1970s that they were properly fleshed out. The guiding principle that drove this was ”structuring.” First, structured programming was proposed, followed by structured design and structured analysis, which had a major impact by tracing the software process backward.

The idea behind structured programming was, first and foremost, to simplify and clarify the control structure of a program. The catchphrase was ”no goto statements.” The use of goto statements causes control to jump around, which makes the behavior of a program difficult to understand and makes it more prone to errors. So what should be used instead of goto statements? The answer was given theoretically by Bohm-Jacopini's ”Structuring Theorem,” which states that any control structure can be written as a combination of concatenation, branching, and looping. By combining these three control structures, programs can be written neatly. Branching and looping can be nested with smaller branches and loops inside each other. Nested structures can also be made more visually appealing by using indentation when writing a program.

The second pillar of structured programming is the abstraction of modules. The concept of modules as a semantic unit of a program has been provided as a component of programming languages called subroutines or procedures. However, abstract modules proposed by David Parnas in the mid-1970s and abstract data types proposed for incorporation into new programming languages such as Alphard, Clu, and Euclid were revolutionary in that they raised the level of abstraction of modules and encapsulated richer semantic content to make them easier to use. The concept of abstract modules was prepared in Simula in the 1960s, but theoretical research such as formal semantics based on algebraic specifications flourished during this period. In addition, structuring based on procedural processing units results in a top-down hierarchical structure, while structuring based on abstract modules has the characteristic of forming a more flexible structure.

Structuring was not only promoted as a proposal for theories, languages, and methods, but also actively implemented in the industrial world. For example, at IBM in the United States, the structured programming movement was promoted by Harlan Mills and others. In Japan, a little later, many structuring diagrams were proposed to replace flowcharts. For example, Hitachi's PAD, NEC's SPD, and NTT's HCP.

Although not directly related to structuring, the concept of software life cycle model and its representative waterfall model were introduced during this period and had a major impact on subsequent software development.

In this way, the 1970s can be said to be a time when research and practice worked like two wheels of a car, with "structuring" as the watchword, and software engineering made great progress.

### 1980s: Shift to management technology

In the course of development, any technical field will be subdivided into specialties. Software engineering is no exception. First, the fields of programming languages and programming methodology left software engineering. However, these fields were long-established before the birth of software engineering, and it may be better to say that they returned to their original place rather than being separated and independent from software engineering.

The scale of software developed became enormous. It is no longer unusual for source programs to have several million lines, and there are cases where the number is tens of millions. In the development and maintenance of such software, management and human aspects become major issues. For this reason, areas such as programming methodologies such as stepwise refinement, modularization, and abstract data types, which were developed within the framework of software engineering in the 1970s, gradually fell out of the scope of software engineering. Instead, management technology was emphasized. The preface frequently used to explain the need for software engineering was that it was necessary to deal with the development of "large-scale, complex software," but the recognition spread that programming-level technology was insufficient for this purpose. In order to develop a large-scale system over a long period of time in a large organization, "management" is more important than anything else. There are many objects of management. Words such as project management, personnel management, budget management, process management, quality management, configuration management, and computer resource management overlap with each other and indicate a variety of management targets. From a practical point of view, it is meaningful to emphasize the management aspect, but the result is that the difference from general project management is not clear, and the characteristics of software engineering tend to fade. It is inevitable to some extent that it approaches business administration and industrial psychology, and meaningful results may come from there, but if nothing remarkable appears in the core of software engineering, its vitality as a science and technology field will decline.

One of the results close to practice is a technique called prototyping. One of the most difficult problems in software development is when user requirements are vague or insufficient. To solve this, a simple prototype is quickly created that allows the user to understand the operation of the system, and the user is asked to try it out to clarify and enrich the requirements. This can be said to be a new model of the software development process, and it influenced the proposal of various process models afterwards. In fact, in the latter half of the 1980s, software processes attracted a lot of attention. Processes were somewhat successful in attracting the interest of practitioners through CMM (Capability Maturity Model) and ISO 9000.

Another practical example is the method known as inspection or review. This is a formalization of the previously widely used method of having people read and inspect specifications and program code, as well as the structure of the team that performs the inspection, the roles of the members, how records are kept, and how corrections are made based on the results. Since IBM reported on its implementation, it has been consciously implemented in many organizations.

These can be said to be solid achievements, but they are hardly a dramatic technological innovation like the structuring of the 1970s. On the other hand, speaking of technological innovation, the 1980s was also a time of an unexpected AI boom. Research into artificial intelligence (AI) began almost at the same time as the birth of the computer, but in the 1980s, expectations for practical application rose greatly with the call for "knowledge engineering" proposed by Edward Feigenbaum. In Japan, a 10-year project to develop the fifth generation computer began in 1982, which became a trigger for similar projects to be launched in the United States and Europe.

Since AI is realized as software, it is natural that it is closely related to software engineering. Many attempts have been made based on the idea that knowledge engineering methods can be used for the complex intellectual work of software development. The logical expressions used in AI inference directly contributed to the development of formal methods in software development. Furthermore, the influence of AI can be seen in the creation and use of various automation tools in software development, including automatic program generation.

Speaking of tools, CASE (Computer Aided Software Engineering) also came into the spotlight from the late 1980s to the early 1990s, a little later than the AI boom. From the name CASE, it seems that the target is all tools that can be used in software engineering, but what particularly attracted attention at that time were tools for the analysis and design phase that combined drawing functions such as data flow diagrams with databases to manage items such as data and processes that appear in the diagrams. Although CASE was only popular in the media for a short period of time, the situation in which such tools are commercialized and used has continued steadily to the present day.

To summarize these trends, it is no exaggeration to say that software engineering in the 1980s was in a period of decline compared to the vibrant 1970s.

### The 1990s: Object-oriented

The 1990s was the era of object-oriented programming. Simula, which is considered the original object-oriented language, was created in the 1960s, and the main concepts of object-oriented programming, such as encapsulation, inheritance, and polymorphism, were all established in the 1970s. However, it was with the release of Smalltalk-80 in 1980 that the name and concept of object-oriented programming became widely known.

Just as structured design and structured analysis were born from the idea of structured programming in the 1970s, object-oriented design and object-oriented analysis were born as methods from object-oriented programming. Around 1990, object-oriented analysis and design caused a great stir with the publication of several books. It seems that the ”object-oriented paradigm” comparable to the ”structured paradigm” of the 1970s has taken the software engineering field by storm.

In the early 1990s, there was a profusion of object-oriented methodologies and a sense of a hundred schools of thought, but UML (Universal Modeling Language) emerged as an attempt to at least unify the notation. This was the origin of the movement to unify methodologies that began when Ivar Jacobson joined the union of Grady Booch of the Booch method and James Rumbaugh of OMT. Later, an alliance called OMG (Object Management Group) took over and focused on unifying the notation, and UML 1.1 was released in November 1997.

In the 1980s, software engineering focused on how to use management technology to tackle the development of large-scale, complex software. At the same time, personal computers were being commercialized and widely used, and connected to the Internet, bringing with them waves of miniaturization, openness, and decentralization. For a time, software engineering lagged behind this movement completely, and neglected the issue of how to respond to the development of small-scale, multi-variety software with rapidly changing needs.

Object-oriented technology was also useful in this regard. The concept of objects was extremely effective as a unit of distribution, modularization, and reuse, and design patterns and frameworks made up of objects were useful for efficiently assembling software needed in the Internet age. For example, Java, which was released in the mid-1990s, was a programming language designed from the beginning with the goals of platform independence, web-based application system development, and security in mind, and responded to the needs of the times.

In general, the 1990s can be seen as a period in which software engineering itself was re-engineered based on such object-oriented technology. If the 1980s was a period of stagnation for software engineering, the 1990s could be seen as a period of recovery when software engineering began to regain its centripetal force. On the other hand, software in the 1990s was far more diverse than that of the 1970s.

Not everything can be solved with a single framework such as object-orientation. However, it can be acknowledged that this helped software engineering to regain a certain degree of centripetal force. After that, object-oriented technology developed new themes such as reuse techniques for units larger than objects, such as patterns and frameworks, component-based software development, and web service technology.

### The 2000s: Advances in analysis techniques

A notable movement in software engineering since the beginning of the 21st century is the advancement of analysis techniques for programs and documents. Static analysis techniques for programs have a long history, but the techniques have become increasingly sophisticated. Furthermore, there has been remarkable progress in the direction of model-level analysis, as represented by model checking, and analysis of program behavior rather than structure. With the spread of UML, the scope of application of analysis of models described in UML has also expanded greatly.

There are at least two factors that have promoted the development of such analytical techniques. One is the release of various open source software, which provides a huge amount of data that can be analyzed. Not only the original code but also design documents, email correspondence, bug reports, and many other data are released. This is a treasure trove of desire for researchers who are trying to apply quantitative data analysis.

The other is the extraordinary improvement in hardware performance. For example, it is now easy to analyze large amounts of log data, which was difficult to do just a short time ago. The fact that it has become possible to use a brute force method such as reducing a model analysis problem to a satisfiability problem (SAT) and solving it with a general-purpose SAT solver is a gift not only from ingenuity in algorithms but also from the speed of hardware.

In this context, empirical software engineering has come to be emphasized, and there has been a growing movement to evaluate software engineering processes and methods using quantitative data. On the other hand, the approach of precise analysis of programs has also become an opportunity to bring software engineering closer to the field of programming languages again.

### 2010s and Beyond

In the 2010s, agile development, which originally started with open source development and development by hacker-type teams, began to be widely accepted in the industrial world. Representative examples of this are Scrum as a development method and DevOps as a methodology for integrating development and operations. In parallel with this, companies have begun to actively participate in open source development. For example, IBM's open source development of Eclipse and Microsoft's acquisition of Github.

As for technology and its applications, AI and data science, centered on machine learning, have been in the spotlight, but both have a two-way influence on software engineering. Both machine learning and data science are realized as software, so the results of software engineering are utilized there, and for example, the advancement of natural language processing technology using deep learning is widely used to analyze documents such as software specifications and emails created by development teams.

In Japan, programming education has started in elementary school, and the subject of ”information” will be added to the university entrance exam subjects from 2022. Many people will have more opportunities to come into contact with software, which was previously invisible to the general public, in the role of creating it. People from many countries and regions, including not only developed countries but also developing countries, are involved in the research and practice of software development around the world. It is expected that the knowledge and methods of software engineering accumulated so far will be utilized in a wider range.

### Software Engineering Research Activities

Inheriting the spirit of the NATO conference in 1968, the International Conference on Software Engineering (ICSE) began in 1975. Since then, it has been positioned as the flagship conference in this field, and is usually held around May. ICSE is organized by IEEE-CS and ACM's SIGSOFT (Study Group on Software Engineering), which also cooperated in the creation of SE2004. ICSE has been held twice in Japan, in Tokyo in 1982 and in Kyoto in 1998. Many researchers have been active at ICSE. Some names not mentioned in other articles include Robert Balzer and David Notkin from the United States, Carlo Ghezzi and Antonia Bertolino from Italy, and Bashar Nuseibeh from the United Kingdom. They have served as executive and program chairs, and many young researchers have presented papers from their groups, which has helped to liven up the event. At ICSE in Kyoto in 1998, the executive chair was Torii Koji, and the program chairs were Niki Kokichi and Richard Kemmerer.

There are many other international conferences on the theme of software engineering. For example, there are regional conferences such as ESEC in Europe and APSEC in the Asia-Pacific region. There are also various thematic conferences such as FSE, which focuses on theory, ASE, which focuses on automation, RE, which focuses on requirements, ICST, which focuses on testing, and ICSM, which focuses on maintenance.

Software engineering research in Japan has been centered around the Software Engineering Study Group of the Information Processing Society of Japan, which was established in 1976. The aforementioned APSEC was started in 1994 as an international conference with academic societies from six countries/regions: Japan, Korea, Australia, Hong Kong, Taiwan, and Singapore as founding groups. The Japanese Software Engineering Study Group played a central role in its launch, with contributions from Mikio Aoyama, Genji Saeki, Yoshiaki Fukazawa, and others. Since then, India, mainland

China, Thailand, Malaysia, Vietnam, and have joined APSEC, bringing the number of member countries/regions to 11. As of 2021, the chairman of APSEC's standing steering committee is Katsuhisa Maruyama of Japan.

Although there is a certain amount of participation from companies in international conferences such as those listed above, the number of participants from universities and other research institutions is overwhelmingly higher in terms of ratio. In the industrial sector, there are many other seminar-style conferences that are more specialized for practice.

# Chapter 2

# Software Process

The basic components of software engineering are products and processes. A product refers to all outputs, including intermediate products and documents, that are generated during the engineering process. A process is a way that produces a product. Therefore, for software engineering, it is important to consider the software development process itself as a methodological subject. The software development process as an engineering subject is usually simply called the software process.

## 2.1 Products and Processes

Looking at the history of software, there seems to be a tendency for periods of increased interest in the process and periods of increased interest in the product to alternate. In particular, from the late 1980s to the early 1990s, software processes attracted a keen attention as a field of software engineering, and activities focusing on the theme of process flourished in both industry and research. One of the things that triggered this was Leon Osterweil's keynote speech entitled "Software Process is Software Too" at ICSE (International Conference on Software Engineering) held in Monterey, California, USA in 1987. Another was the Capability Maturity Model (CMM) by the Carnegie Mellon University Institute for Software Engineering, which was first proposed in the same year as a process evaluation model. This is a five-level evaluation of the development process carried out by software development organizations.

In the 1990s, a series of international workshops and conferences on the theme of software process were held one after another. In particular, the 6th International Software Process Workshop (6th ISPW), held in Hakodate in October 1990, was chaired by Takuya Katayama, with Koichi Kishida and Koichiro Ochimizu as committee members from Japan, and attracted about 30 participants from overseas, where lively discussions took place.

However, the enthusiasm for software process cooled off rapidly after the first International Conference on Software Process (ICSP) was held in Redondo Beach, California, USA in October 1991. In the mid-1990s, interest in process was replaced by a period in which software architecture became popular. Since architecture means the structure of a product, it can be seen as a shift in attention from process to product, and this trend may be explained by the trend of such a shift in attention. The shift in attention between products and processes seems to apply even if we extend it beyond this period. For example, before the mid-1990s, when interest in process increased, interest in object-oriented products was at its peak. After a shift from interest in process to interest in architecture, agile processes began to attract attention from the late 2000s to the early 2010s, indicating return of interest to processes. On the other hand, after the peak of interest in architecture, interest in product lines gradually increased, which can also be seen as a return to interest in products. Figure 2.1 shows this in a schematic form.

## 2.2 Plan-driven process vs. iterative evolutionary process

In the early days of software engineering, the concept of a life cycle was proposed. This is a standard model of the process of software development, from planning to design and development, operation, maintenance, and finally disposal. In other words, it comprehensively shows the entire software process, and standard processes in software development

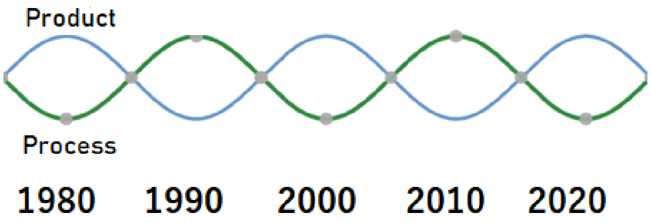


Figure 2.1: Alternating focus on product and process

organizations such as companies are created in accordance with some kind of life cycle model, and project management is carried out based on it. This is a model that is composed of the largest unit as a process model.

The term life cycle was originally used in biology to refer to the process of development and growth that is repeated for each generation (it is sometimes translated as life cycle). However, it is generally used to refer to the process of a person's life or a product from its release to its disappearance from the market. The term probably entered software engineering not through biology, but through existing engineering, and was adopted to mean the process from when an industrial product is released to the market until it is gone.

A life cycle model can be seen as a representation of how software development should be, and in that sense it presents a normative model of the software process. The purposes of such a model are as follows:

1. To define standard software development procedures and guide the work of actual developers.
2. To use as a management model that conforms to the management of development projects.
3. To serve as a basis for defining standard development methodologies, tools and development environments, standard documentation systems, etc.

Such a software process model must be designed for each development organization and each project, based on the standard. In fact, in large organizations, departments are created that specialize in designing software processes, and are given the role of designing standard processes used within the organization and processes for each project, as well as observing operation and providing appropriate feedback.

When the life cycle model was proposed in this way, emphasis was placed on a ”plan-driven” process, which requires planning the process properly in advance and implementing it according to the plan. However, in reality, development cannot be planned in such a way, and even if planning is done, it is inevitable that changes will occur during implementation. This is especially true for innovative software that is newly created, and so ”iterative evolutionary” processes have been devised and implemented based on the recognition that such planned processes are not suitable.

## 2.3 Plan-driven process

### 2.3.1 Waterfall model

The typical example of a plan-driven model is the waterfall model, which is the oldest of all life cycle models and is still the basis for the standard processes of many companies today. Here, waterfall does not refer to a vertical waterfall like Nachi Falls or Kegon Falls, but rather to something that falls in steps, as in Figure2.2.

Figure2.3 shows a typical waterfall life cycle model. Here, each process unit such as analysis and design is called a phase.

Winston Royce is said to have first proposed the waterfall type life cycle model in a paper[114]. This paper does not use the word waterfall, or even the term life cycle, but it

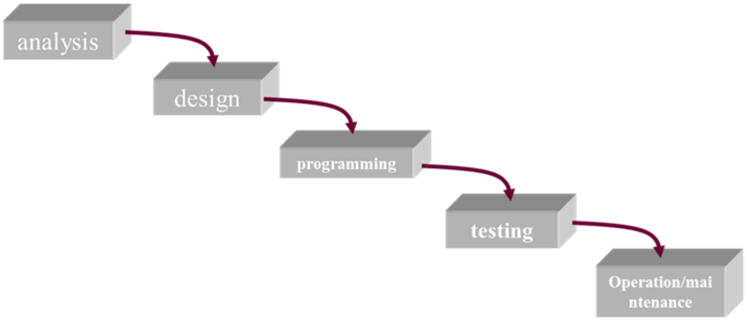


Figure 2.3: Waterfall type life cycle model

not only draws a picture equivalent to Figure 2.3, but also presents several variations and
develops a detailed discussion. Although the process shown in Figure 2.3 is criticized, the paper is still thought-provoking even when read today.

The model, which was based on Royce's model and has since come to be generally known as the waterfall type, emphasizes the following two points.

1. A clear division is made between phases, and well-formatted documents are handed over between them.

2. Rework between phases is minimized.

However, a survey by Akito Ito and the author has shown that it is not realistic to eliminate rework when observing actual projects, and many reworks occur. [129, 148]. There is a considerable amount of rework not only in one phase, but also in two or more phases, which greatly increases development costs. The reason why the waterfall model has been the mainstream in the field for a long time despite various criticisms is that it is convenient for project management.

### 2.3.2 V-shaped model

The waterfall model also breaks down the test phases, dividing them into unit tests for programs, integration tests for designs, and acceptance tests for requirements, and sometimes depicts them as a V-shaped life cycle model, as shown in figure 2.4.

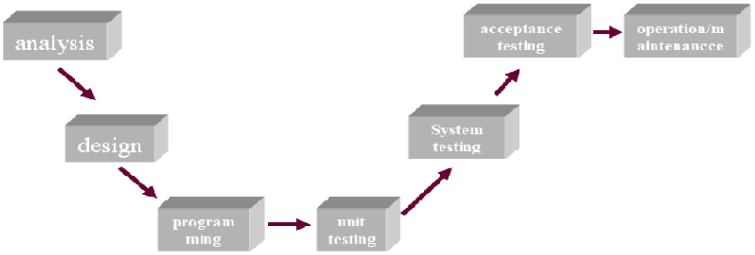


Figure 2.4: V-shaped life cycle model

Based on a proposal from Japan, the International Organization for Standardization (ISO) has adopted this type of model in its standard life cycle model ISO/IEC/IEEE 12207. The first edition was published in 1995, and after revisions, the current edition was published in 2017. The Japanese translation of this is the "Common Frame," and as of December 2021, the latest edition is "Common Frame 2013."[146]

### 2.3.3 Prototyping Model

The limitations of the waterfall model were pointed out early on.

As already mentioned, reworking between phases is common in the real software development process. Everyone can understand the goal of minimizing this, but a process model that ignores the fact that reworking actually occurs is not realistic. In particular, requirements analysis in the early phases is problematic, and requirements are often not clearly defined, remain ambiguous, or change as the process progresses. However, if the design phase cannot be advanced unless the requirements are completely determined and properly documented as specifications, this will hinder the progress of the project.

In addition, in development of a certain scale, it is natural that the degree of progress will vary depending on the part. It is almost impossible to switch between phases or lower steps at the same time as the entire project. In fact, even from the perspective of the concept of concurrent engineering, which is practiced as an efficient method of manufacturing processes and has been effective, the rigid waterfall model does not fit reality.

Even if the results of requirements analysis are difficult to determine, the further into the design, implementation, and verification phases that follow, the greater the losses if reworking the requirements analysis is required. This is where the ”prototyping model” comes in.

Prototyping is the process of creating an experimental but working system (prototype) and evaluating it before creating the final software system that will be used. Usually, the purpose of prototyping is considered to be limited to clarifying user requirements. A development model that incorporates the prototyping process in this sense into the requirements analysis phase is called a prototyping model. The prototype is repeatedly revised to determine the requirements specifications, but the specifications are fixed at an appropriate point, and the traditional waterfall development model is followed thereafter.

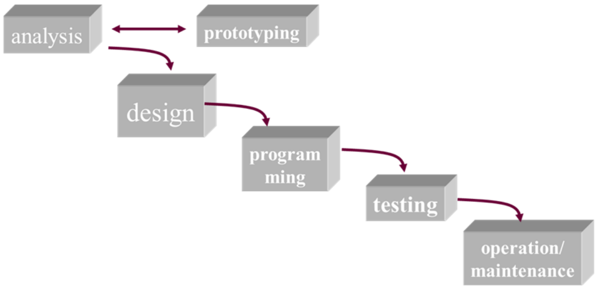


Figure 2.5: Prototyping-based life cycle model

Prototyping involves the creation of a prototype with the appearance of a user interface that allows users to understand the operation of the system. By touching and moving the prototype, users can get an image of the system and become aware of misunderstandings or deficiencies in requirements. Developers can deepen their understanding by interacting with the user through the prototype and make improvements to the prototype. Prototypes are usually disposable, and the actual system is rebuilt from scratch.

The general rule was to throw away prototypes, but when a usable one was created, users continued to use it, and there were cases where there was no need to create a new actual system. However, since prototypes are originally created with the aim of creating them quickly, the quality of the product is secondary. Therefore, if a prototype is used because it is usable, unexpected operational problems or maintenance problems may occur later.

## 2.4 Iterative Evolutionary Process

Instead of building an operational system by executing the development process from requirements analysis to implementation only once, the iterative evolutionary model is a development process in which a system with a small range of functions is first realized and then the process of improving it is repeated, and the system’s completeness is flexibly improved while responding to the expansion and changes in the range of user requirements.

It has some similarities to the prototyping model in that it provides users with a system that actually works quickly. However, the difference between the prototyping model and the iterative model is in the judgment of which functions or partial systems to prioritize development.

In prototyping, the parts of the user's requirements that are ambiguous are prioritized, a prototype is created, and the requirements specifications are refined. In contrast, the incremental improvement model takes the approach of prioritizing the development of products with clear functions, clear development effects, and few uncertainties in terms of design technology, and gradually adding new and challenging functions.

### 2.4.1 Spiral model

Among the iterative evolution types, the spiral model by Barry Boehm, shown in Figure 2.6, was proposed relatively early and is well known. [23]

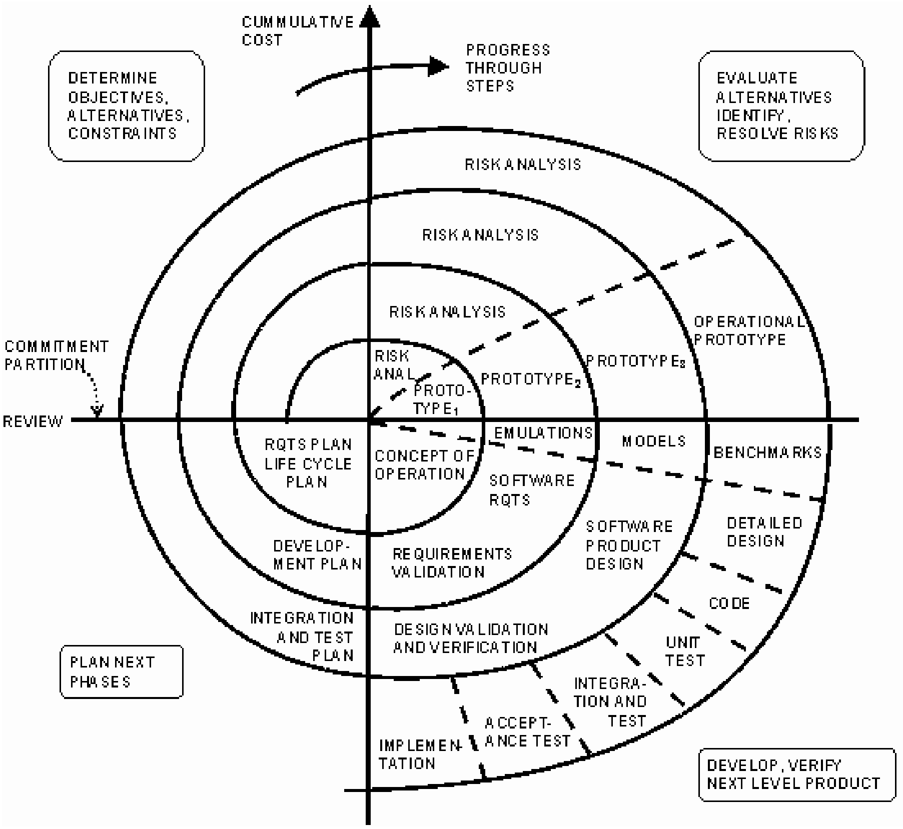


Figure 2.6: Spiral model by Boehm [23]

In this figure, the planning step of considering the objectives, alternatives, and constraints, the analysis step of alternative evaluation, risk analysis, and solution, the development step of design and verification, and the next phase consideration step of planning the next phase are repeated four times. This is a process of gradually evolving the system while controlling risk, but this cycle does not have to be four times, and it can be more or less. Boehm claims that it is judged based on the size of the risk remaining in one cycle.

### 2.4.2 Extreme Programming

Extreme programming (abbreviated as XP) is a development process that takes iterative evolution to the extreme. This method was proposed by Kent Beck and others, and is mainly used in object-oriented development projects. It is also typically used in open source projects. [10]. The following methods, for example, are proposed as components of XP:

- First, decide on a plan for software release, and then determine a development plan based on that.
- Divide development into a repetition of small units of work.
- Software is released frequently for each small unit that has been divided.
- When developing, first create tests that can be executed automatically, and then write code that satisfies the tests (test-driven development). The tests essentially play the role of specifications.
- Development is carried out in pairs (pair programming). One person writes the code, and the other checks it, working together.
- If small units of development are accumulated, the structure of the entire system will deteriorate. Therefore, frequent refactoring is required to reconstruct the system so that its internal structure is improved and it is more efficient while its functions remain unchanged.

### 2.4.3 Scrum and Agile Development

As the merits of extreme programming are gradually recognized, there is a movement to apply it not only to small-scale research and development projects but also to larger-scale development in the industrial sector. Scrum is a representative development methodology. Seventeen people who promote extreme programming and Scrum came together and collectively called this movement agile development, and in 2001 they published its core concept as the ”agile manifesto.” It goes like this:

- Emphasis on individuals and their interactions over processes and tools
- Emphasis on programs over documents
- Emphasis on collaboration with customers over contract negotiations
- Responding to change over adhering to plans

Scrum is a process methodology proposed and practiced by Ken Schwaber and Jeff Sutherland, who also participated in the Agile Manifesto. The name Scrum comes from rugby, but it was first used in a paper called “The new product development game” by Hirotaka Takeuchi and Ikujiro Nonaka[126]. Nonaka and Takeuchi proposed the idea of ”knowledge-creating companies” in the field of business administration, and are particularly well-known in the United States[99, 166]. The spiral model also had the idea of evolving by repeating the development cycle. The basis of Scrum is to shorten the cycle, for example from two to four weeks, and provide users with a new version of the software each time the cycle is completed. This life cycle model is common to agile processes.

The roles of members in a scrum team are only three: product owner, developers, and scrum master. The product owner is responsible for maximizing the value of the product being developed. He/she decides what functions should be realized in the product from the customer’s perspective, prioritizes them, and manages them as a backlog. There is only one product owner in a team. The developers are the people in charge of the actual development, and can have various specialties such as architects, designers, data analysts, and testers, but there is no distinction between titles. The scrum master is the coordinator of the entire team, and his/her role is to remove various obstacles that arise during development. He/she is not a traditional team leader, but a supporter and coordinator.

Development progresses by repeating a process called a sprint, which is a one-month unit. Before each sprint, the scrum team meets with the customer to prioritize the work to be done, and based on that, decides the development content to be implemented in the next sprint. During the sprint, the development team holds a short scrum meeting every day to understand the current situation. At the end of a sprint, the increment in product functionality is completed in a form that can be shipped. In addition, a review is always conducted after the end of a sprint.

Scrum is said to be suitable for products with easily changing requirements, such as web services and product development in new fields, but there are reported cases where large-scale, long-term projects that have experienced repeated setbacks in development have been successfully switched to the Scrum method midway through[125].

### 2.4.4 Integration of development and operations

DevOps is a practical methodology that has emerged from agile development. It is a proposal based on the idea of integrating development and operations both as a method and as an organization, and is particularly attracting attention and being practiced in the industrial sector[112, 66]. Tools have been created to integrate the two and speed up shipping, and their use is spreading.

## 2.5 Software Process Evaluation

A well-known method for evaluating software processes is the Capability Maturity Model (CMM), a process maturity evaluation model developed by the Carnegie-Mellon University Software Engineering Institute (CMU-SEI). Originally, it was developed to evaluate development organizations when the US Department of Defense procured software products from outside. It has come to be used as a tool to evaluate the processes implemented in software development organizations in general and lead to their improvement. In particular, the evaluation part is a relatively simple five-stage evaluation, which has had a great impact.

The five stages are defined as follows:

**Level 1** Initial

The process is barely defined, and even if software development is going well, it is a state where it depends on chance or individual ability.

**Level 2** Repeatable

Basic management processes for cost, schedule, and function management are established. A state in which the system has been established to repeat the process by making use of past experience in developing similar systems.

**Level 3** Defined

The software process is documented, standardized, and incorporated into the organization from both the management and development perspectives. All projects use the defined standard process with modifications.

**Level 4** Managed

Detailed quantitative data is collected about the software process and product, and analysis and management are carried out based on that data.

**Level 5** Optimizing

Process improvements are constantly being made based on feedback from quantitative analysis.

For this stage-based evaluation, a large number of evaluation items are defined, categorized into several process areas, and it is also indicated which process areas and items should be improved in order to move up to the next level.

In fact, the purpose of such process evaluation is not to rank, but rather to lead to process improvement. Therefore, Software Process Improvement Networks, abbreviated as SPIN, are being implemented at the regional and organizational levels. CMM itself has also developed, and CMMI (Capability Maturity Model Integrated), an integrated model, was formulated. Version 2.0 of CMMI was released in 2018[36].

Evaluating a process also means assuming a normative process model. The oldest normative process model is the waterfall model. ISO/IEC/IEEE 12207 SLCP[156] by ISO/IEC JTC1/SC7/WG7 can also be considered a normative process model. SLCP stands for Software Lifecycle Process. The first edition was published in 1995, and the current edition was published in 2008 after revisions. The Japanese translation of this is the ”Common Frame,” and as of December 2021, the latest version is ”Common Frame 2013.” [146] These show the overall flow of development in a process, but the above-mentioned CMMI, ISO9000, SPICE, etc. consider the internal structure of the process a little more and put the evaluation perspective at the forefront. **ISO 9000**

The ISO 9000 series defines standards for quality control and quality assurance systems. In Europe, a system has begun to take root in which a third-party organization examines and certifies whether the quality system of the applicant company conforms to the system specified in the ISO 9000 series, and this has spread to the United States and Japan. The standard document for this system is ISO 9001. 9001 specifies the quality system requirements that suppliers should have

for general products, especially when they are manufactured and supplied under a two-party contract. In particular, ISO 9000-3 is a standard created as a guide for applying this to software development/design, supply, and maintenance.
**SPICE**
SPICE has been developed with the goal of integrating CMM and several similar process evaluation models to create a global standard. It feels like Europe has taken over the CMM, which was born in the United States, and in that sense it is a fusion with ISO 9000. SPICE is the nickname given to it based on the project that developed it, and the standard was ISO/IEC 15504, but in 2015 the ISO33k series, including ISO33001, was launched and replaced it[65].

## 2.6 Process Programming

As mentioned in the "Product and Process" section at the beginning of this chapter, L. Osterweil proposed process programming in 1987 with the catchphrase "Software processes are software too." This was the trigger for the rise of process research. Osterweil's idea was to describe the process of developing software as a procedural program and "execute" it. Since then, many proposals have been made for formal languages and models to describe processes, as well as process support environments to run them. The purposes of these have also varied widely, from empirical analysis of processes to evaluation, improvement, and the definition of standard processes, but Osterweil and others, the original proponents, emphasized the aspect of executing the described process, and ultimately aimed to automate the development process.

However, there is also an argument that the process can only be described as a program in a very limited part of the entire development process. In addition, although humans are one of the subjects that execute processes, some mechanistic positions that try to treat the uncertain behavior of humans as deterministic as possible have been criticized.

Software processes are actually design processes, not manufacturing processes.

It is important to note that they are fundamentally different from processes in chemical engineering and the like. Until now, design processes have not been given much attention in other engineering fields either. For that reason, formalization is certainly a difficult problem.

Process programming as a research project has been carried out with the following goals:

1. Observe and analyze actual processes, especially development processes in which human behavior is a major element, and explore ways to improve them.
2. Research into formal systems and languages suitable for the formal description of processes, and conduct actual description experiments.
3. Develop a software development environment centered on the concept of process.

The idea of process programming did not succeed as originally intended, but the following results influenced software engineering thereafter.

**Formal description/modeling of processes** The process is described formally using, for example, a functional or logic programming language, and then clearly described and analyzed.

**Process-centric software development environment** There are many integrated software development environments that are currently in practical use, but in addition to repository management, process management such as scheduling is also important. Therefore, various attempts have been made and implemented based on the idea of building and operating the entire development environment with a process-centric approach.

**Process design and execution** To apply the concept of process programming to actual software development, two steps are required: process design and execution.

There are two concepts for process design:

- **prescriptive** The process is shown as a procedure. It is concrete and easy to link to execution, but it lacks flexibility.
- **proscriptive** Describe the conditions that the process must satisfy. It is easy to respond to dynamic changes in the process, but there is a gap with execution.

In any case, it is difficult to imagine a completely new design for each software development project. It is practical to customize the process for each organization and project, based on a basic general process model.

What does it mean to execute a designed process? If we think about it from a human perspective, the expression ”execution” would be appropriate. If we use the expression ”enaction” by a computer, we first assume that the developer is guided (navigated) and given guidance. In addition, a function to verify whether the process follows the specified constraints or is steadily progressing in the direction of the goal can also be considered. For this purpose, it is necessary to automatically measure various characteristics of the process and perform numerical management. Furthermore, if we take the position of ”strong” process programming, we aim not only to support the human developer but also to execute the process automatically as much as possible.

However, guidance and warnings are possible only when the process is plan-driven, and are not very effective when the process is not known how it will change, as in the case of agile. Agile emphasizes that people should consult with each other when an unexpected situation occurs, and this is where the difference between the two positions emerges.

**Description of Scheduling** Process models have several aspects, but most models deal with the area of process scheduling.

As with traditional PERT, one natural way to construct a schedule is to explicitly specify the relationship between work units. Another possible method is to indirectly determine the schedule by describing the events that trigger each work unit and the events that occur from the work. An even more indirect method is to specify the input and output conditions of the work, rather than the events that trigger the work unit, and to build a scheduler that satisfies those conditions. There are two types of scheduler methods: a forward-looking method that extracts tasks that satisfy input conditions in the current state, changes the state based on the output conditions after execution, and searches for the next task that can be executed; and a backward-looking method that generates conditions backwards from the goal and determines the execution order.

**Object management** Alongside scheduling, another important aspect is the management of objects (products, artifacts) that are generated and used in the process. If the process model is interpreted narrowly, object management can be considered to be outside of it, but it is an important subsystem in a process-centric development environment.

Technically, it is based on the technology used in traditional configuration management and so-called repository systems. On top of that, more diverse consistency maintenance techniques are required in relation to processes. In this case, the techniques accumulated in databases are applied.

# Chapter 3

# Requirements Engineering

Software engineering is often thought of as the pursuit of how to make software, but before we can answer ”how,” we need to clarify ”what” to make and ”why.” This is clarified through ”requirements engineering,” a technique for determining the ”requirements” for the product to be developed.

## 3.1 What to Make

How does software development begin? Or, to use the terminology introduced in the previous chapter, what is the first task in the software process? The answer to the latter question seems obvious when we look at life cycle models such as the Ochimizu type, where the first phase is ”analysis.” However, this has simply been replaced by the question of what is included in the analysis task, and even if this is clear, the question arises as to whether there is anything that comes before the ”analysis.”

To start software development, it is first necessary to decide what to make. ”Analysis” is usually thought to mean ”requirements analysis.” Here, ”requirements” refers to what users and other stakeholders have, either explicitly or implicitly, about the system to be built. The whole process of eliciting such requirements, defining them as requirements specifications, evaluating their accuracy, non-ambiguity, and sufficiency, and then tracking changes and evolution of the requirements specifications is called ”requirements engineering.” This term implies that the target product is not limited to software, but can be applied to the development of any product. However, the term ”requirements engineering” was historically first used in the software world. To decide what kind of software should (or should not) be developed, it is necessary to predict what value the software will bring and how much it will cost to develop it. Once software has been created as a product, its value can be measured by how much it is used, and especially in the case of commercial software, by how much it sells. However, software that is functionally excellent and appears to have high commercial value does not necessarily sell well. Even software developed individually is not necessarily used well just because it is well made. On the other hand, development costs can be divided into two categories: labor costs and equipment costs, which can be simply converted into monetary terms, and development time and other necessary resources, which are indirect.

When deciding ”what to make,” a decision must be made as to whether development should be started or at least whether consideration of development should begin. To make that decision, the type of system must also be assumed. In other words, consideration of ”what to make” and ”whether to make” must be carried out in parallel.

In short, this type of decision-making issue is very strongly influenced by human factors, and is a complex field where business administration, psychology, market analysis, general problem discovery and solving methods, etc. intersect in various ways. It is difficult to know to what extent software engineering, a methodology that basically focuses on ”how to make,” should be involved. Mechanical engineering and electrical engineering textbooks will not directly address ”what to make.” Software engineering has a great degree of freedom in what to create, and requires continuous communication between users and developers, so the task of requirements engineering is particularly important, and it is somewhat unavoidable to deal with problems that involve value judgments.

## 3.2 Requirements Engineering and Idea Generation

Requirements engineering is aimed at creating something new, which is the field of innovation. Of course, there are systems for which many similar systems already exist, and the development team has experience of repeatedly creating similar systems in the past, so there are cases where no special creativity is required. Even in such cases, some judgment is required, such as what combination of functions to use and what the user interface should be. On the other hand, when pursuing the development of a completely unprecedented system, the key is the ability to create something new and to discover and solve problems. Various idea generation methods and creative development methods have been tried to apply to software as well[85, 117]. Below, we will give an overview of some representative ideas.

### 3.2.1 KJ Method

The KJ Method is a card-based method developed by cultural anthropologist Kawakita Jiro[152, 153]. Kawakita was a member of the Imanishi Kinji group at Kyoto University, and the KJ Method was born from the need for field science. The key points of this method are how to organize field observation notes and how to generate ideas in a team. The method involves the following steps.

**1. Determine the topic** Clarify what the problem is and decide on the topic to which the KJ Method will be applied.

**2. Make notes** Make many notes with one-line headings and turn them into cards or pieces of paper.

**3. Exploration** There are two types of exploration: internal exploration and external exploration. Internal exploration is exploring your own mind to find problems, related matters, and ideas. External exploration is going out into the outside world, not inside your own mind, and finding related information through observation.

**4. Brainstorming** Brainstorming has a longer history than the KJ method. It was proposed by Alex Faickney Osborn of the United States in a book called “Applied Imagination” published in 1953[103]. The following four principles are emphasized in the book.

- Do not criticize other people’s opinions and strictly prohibit quick conclusions.
- Speak freely about what comes to mind.
- Generate as many ideas as possible, focusing on quantity over quality.
- Improve by combining ideas with other people’s opinions.

The KJ method also uses this method to generate ideas and take notes.

**5. Group organization** Spread out notes written on cards or sticky notes on a large surface and look at them, and group similar ideas into groups. Give each group a heading. If you find a unity between the groups, further organize them into medium-sized groups.

**6. Type A Diagram Method** Look at the pieces of paper in a group and arrange them on a flat surface according to their relationships. Then look at the headings attached to the groups and arrange the groups on the flat surface. Depending on whether the relationships are mutual, antagonistic, or one-way, use arrows or other methods to connect the groups. You can also use a method of surrounding similar items with a closed line.

Figure 3.1 shows an example of a Type A diagram. This diagram itself represents the KJ method.

**7. Type B Writing** Decide on a strategic starting point on the Type A diagram, and use that as the starting point for writing. From then on, choose a point near the diagram you have already written and write. A group of ideas that are neither too complex nor too simple should be grouped together in Type A decomposition, so gradually gather them together and build them up. Then, write as various ideas and data intertwine and build up. A general rule of thumb when writing is to write in a way that makes it clear the difference between the description of the facts and your own interpretation.

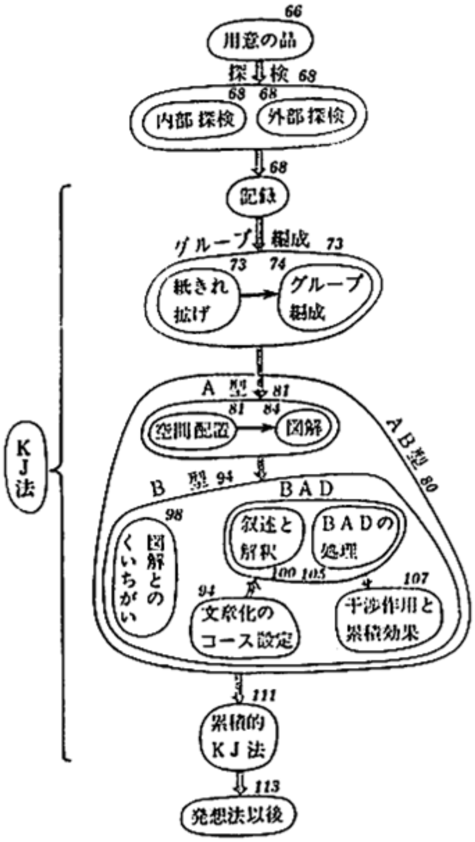


Figure 3.1: KJ Method Type A Diagram

### Applications of the KJ Method

The KJ Method was born from the need for field science, but Kawakita says it can also be used in the following areas:

- Creativity development
- Meetings
- Understanding books
- Persuasion
- Counseling

### Tools used in the KJ Method

Initially, the KJ Method recommended using tools such as cards, construction paper, paper clips, scissors, glue, colored pencils, and rubber bands. Of the products that came out later, Post-it notes would be effective, and currently, various tools such as electronic whiteboards and remote conference systems that allow file sharing can be used. Also, some electronic versions of the KJ Method were prototyped at universities in the 1990s, but there are currently products available on the market, such as Lucidchart.

### 3.2.2 Soft Systems Methodology

Soft Systems Methodology (SSM) is a methodology that has been advocated since the 1980s by Peter Checkland and others in the UK[31]. It aims to be a methodology that can deal with problems that arise from systems that do not necessarily have physical entities or clear boundaries, and is said to be effective when the nature of the problem changes depending on the perspective from which the subject is viewed.

#### Seven Principles

The key concept in SSM is the ”problem situation.” A situation is something that people face on a daily basis, and the goal of SSM is to show how to deal with the problems that arise there. In fact, it would be more appropriate to call it a ”problematical situation” rather than a ”problem situation.” This is because the problems to be solved are not always clearly defined. SSM was born from systems engineering, but it does not deal with rigid ”systems” that have clear boundaries and can be handled logically, as systems engineering targets. This is where the term ”soft” systems comes from.

When dealing with problematic situations in the real world, people make judgments about what is good and bad, but it is their worldview that determines this. In SSM, worldview plays an important role. The SSM methodology can be summarized in the following seven principles:

1. A real-world problem can be understood as a problematic situation in the broader real world, one that calls attention to something and motivates action on the part of the person in it.
2. All thinking and discussion of problematic situations is influenced by the worldview of the person who thinks about it. The worldview is internalized, and assumptions are not conscious of the person, but are taken for granted.
3. Problematic situations in the real world involve people who act purposefully and with will. Models of purposeful action are therefore useful tools for building system models that represent a particular worldview. Such system models are then used to explore the characteristics of problematic human situations.
4. Through repeated discussion and debate on such system models, a set of questions for the situation can be structured and compiled.
5. In the process of discussion and debate to improve real-world situations, a reconciliation between different worldviews can be found. This means finding an alternative situation in which different people with different worldviews can coexist.
6. The exploration based on these five principles is a never-ending learning process. It never-ending because when you take action to improve the situation, the characteristics of the situation change. Repeat steps (3) to (5) for the new situation.
7. Repeat conscious reflection and reconsideration of the above process. This means reconsidering the situation itself and your perspective on the situation.

Based on these seven principles, the following five steps of action emerge.

1. Understanding the situation
2. Exploration using a purposeful action model based on a different worldview
3. Discussion and debate of the situation
4. Definition and execution of actions to improve the situation
5. Reflection and reconsideration of the above actions

The core of this is the construction of a ”purposeful action model.” In SSM, the method of constructing this model is described using a large number of mnemonic abbreviations.

1. RD (Root Definition): A statement that describes the core of the behavioral model, and serves as the starting point for constructing the model.

   [Example] ”A system for painting garden hedges.”

2. PT (Primary Task), IB (Issue Based): Distinguish RD as PT or IB. For IB, it is important to think beyond the boundaries of existing organizations.

3. P, Q, R: An extension of RD as ”do P, by Q, in order to achieve R.”

   [Example] In the case of ”A system for painting garden hedges,” P and R are self-evident, but Q could be, for example, ”with your own hands.”

4. CATWOE: The most well-known abbreviation in SSM, referring to the following six items.

   Customers, Actors, Transformation process, Worldview, Owners, Environmental constraints

   A purposeful action is initiated by an actor A who executes a transformation process T based on a worldview W under environmental constraints E. The action benefits or affects a customer C. However, the owner O of the system can stop or change the action.

5. Evaluation criteria $E_1, E_2, E_3$: efficacy, efficiency, effectiveness, respectively

   Efficacy $E_1$ is the criterion of whether transformation T produces the intended results. Efficacy $E_2$ is the criterion of whether transformation T uses the minimum amount of resources. Effectiveness $E_3$ is the criterion of whether transformation T contributes to achieving higher-level or long-term goals.

Another tool used is the rich picture.

**RichPicture** A problematic situation is composed of various elements. A rich picture allows you to intuitively grasp the whole situation. A rich picture is used metaphorically to describe the overall appearance of the situation, but it can also be drawn as a picture. Figure 3.2 shows an example.

### 3.2.3 Hackathon

The word hackathon is a combination of hack and marathon. In other words, it is an event that brings together good hackers, makes them compete with each other in teams, and devote a few hours or days to developing new software that works properly. As the relatively early and well-known example of an event organized by Sun Microsystems shows, even though hackathons were born from the hacker community, they are still often held today as a way for companies to solicit ideas from outside. The concept is open innovation, that is, it is considered an effective method of creating open products and technologies through the participation of people beyond organizational boundaries.

Since then, more and more events have been held in Japan as a tool to get new ideas not only for software development, but also for the development of new businesses, new products, and services. Unlike the KJ method and soft system methodology mentioned above, it is characterized by the fact that ideas are squeezed out in a tight situation, and is suitable for fields where the creativity that is born under such circumstances is valuable. On the other hand, it is often held for the purpose of education and building communities through human interaction.

### 3.2.4 Mathematical analysis methods

In this way, various methodologies are used in the analysis stage of requirements engineering, and interview techniques, meeting techniques, and camp-style discussions are also important. More mathematical methods also use algorithms using graphs and matrices. Here are some examples that have been devised long ago and are still in use today.

#### Graph-based analysis methods

ISM and Dematel, both of which were developed in the 1970s, extract hierarchical structures between concepts from graphs that represent the relationships between concepts.

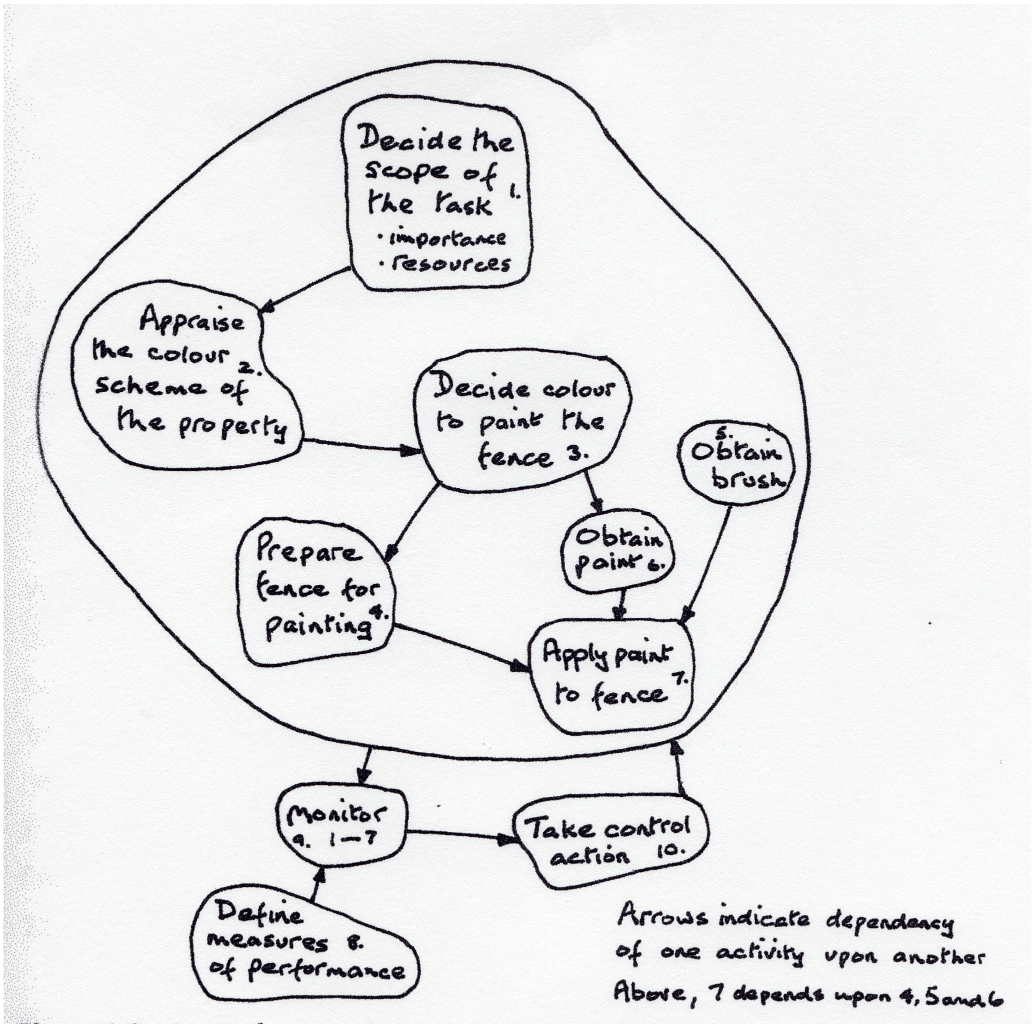

Decide the scope of the task 1.
· importance
· resources
Appraise the colour scheme of the property 2.
Decide colour to paint the fence 3.
Prepare fence for painting 4.
5. Obtain brush
Obtain paint 6.
Apply paint to fence 7.
Monitor 9. 1—7
Take control action 10.
Define measures 8. of performance
Arrows indicate dependency of one activity upon another
Above, 7 depends upon 4, 5 and 6

Figure 3.2: Example of a rich picture: ”A system for painting garden fences” from [31]

**ISM (Interpretive Structural Modeling)** ISM was developed by J. Warfield of Battelle Columbus in the United States around 1973. The method uses a tree structure extraction method on an acyclic graph, but if there is a cycle, it also includes removing the cycle using a heuristic method.

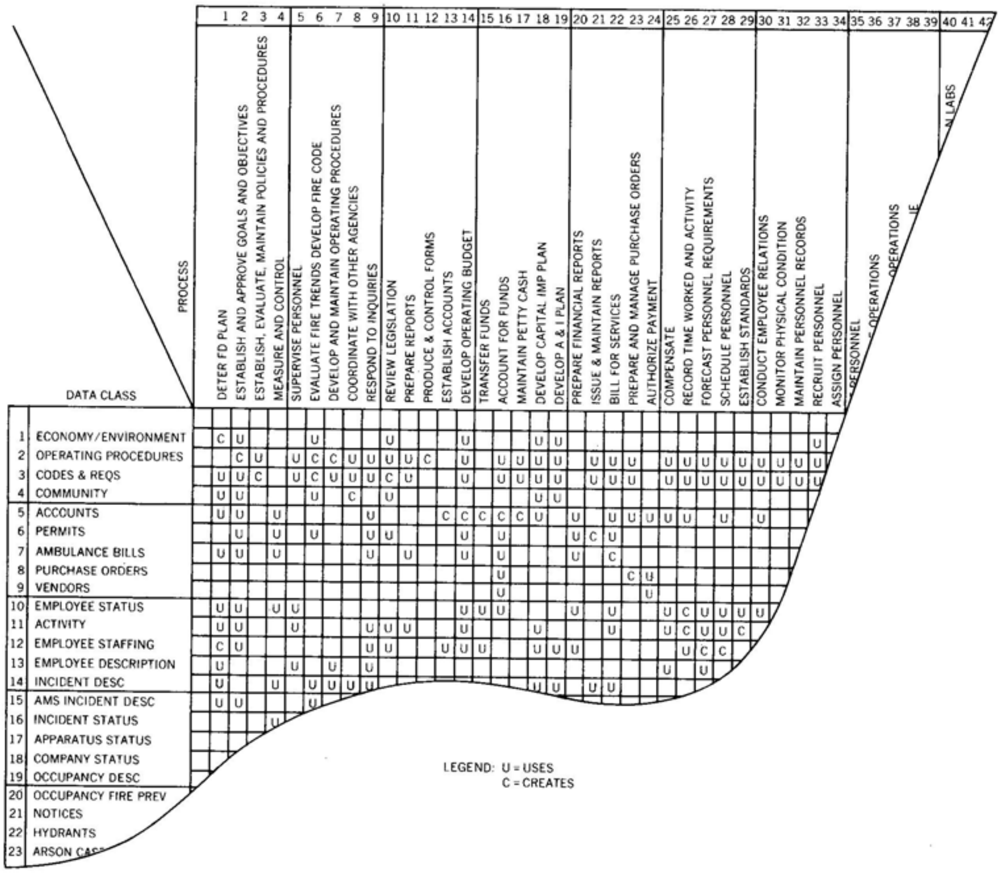
**Dematel (Decision Making Trial and Evaluation Laboratory)** Dematel was developed around 1973 by A. Gabus & E. Fontela of Battelle Geneva in Europe. The method is very similar to ISM, and it also extracts hierarchical structures between concepts.

**Analysis methods using matrices**

A representative method using matrices is BSP (Business Systems Planning), developed by IBM in the 1970s. It was initially developed for internal use by IBM, but has been made publicly available to the public since the 1980s. Its purpose is to analyze and design information structures within organizations.

**BSP procedure**

1. Identify the types of data and processes (tasks/organizational units) within the organization, and create a matrix with these as the vertical and horizontal axes.
2. In the matrix cells, enter C if the data is created in that process, U if it is used, and leave it blank if neither.
3. Rows and columns are appropriately swapped so that non-blank elements are aligned near the diagonal.
4. Blocks of rows and columns along the diagonal represent sets of highly related data and processes, so they can be used as clues for database design and reorganization.



| DATA CLASS \ PROCESS | 1 DETER FD PLAN | 2 ESTABLISH AND APPROVE GOALS AND OBJECTIVES | 3 ESTABLISH, EVALUATE, MAINTAIN POLICIES AND PROCEDURES | 4 MEASURE AND CONTROL | 5 SUPERVISE PERSONNEL | 6 EVALUATE FIRE TRENDS DEVELOP FIRE CODE | 7 DEVELOP AND MAINTAIN OPERATING PROCEDURES | 8 COORDINATE WITH OTHER AGENCIES | 9 RESPOND TO INQUIRIES | 10 REVIEW LEGISLATION | 11 PREPARE REPORTS | 12 PRODUCE & CONTROL FORMS | 13 ESTABLISH ACCOUNTS | 14 DEVELOP OPERATING BUDGET | 15 TRANSFER FUNDS | 16 ACCOUNT FOR FUNDS | 17 MAINTAIN PETTY CASH |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 ECONOMY/ENVIRONMENT | C | U | | | | U | | | | U | | | | U | | | |
| 2 OPERATING PROCEDURES | | C | U | | U | C | C | U | U | U | U | C | | U | | U | U |
| 3 CODES & REQS | U | U | C | | U | C | U | U | U | C | U | | | U | | U | U |
| 4 COMMUNITY | U | U | | | | U | | C | | U | | | | | | | |
| 5 ACCOUNTS | U | U | | U | | | | | U | | | | C | C | C | C | C |
| 6 PERMITS | | U | | U | | U | | | U | U | | | | U | | U | |
| 7 AMBULANCE BILLS | U | U | | U | | | | | U | | U | | | U | | U | |
| 8 PURCHASE ORDERS | | | | | | | | | | | | | | | | U | |
| 9 VENDORS | | | | | | | | | | | | | | | | U | |
| 10 EMPLOYEE STATUS | U | U | | U | U | | | | | | | | | U | U | U | |
| 11 ACTIVITY | U | U | | | U | | | | U | U | U | | | U | | | |
| 12 EMPLOYEE STAFFING | C | U | | | | | | | U | U | | | U | U | U | | |
| 13 EMPLOYEE DESCRIPTION | U | | | | U | | U | | U | | | | | | | | |
| 14 INCIDENT DESC | U | | | U | | U | U | U | U | | | | | | | | |
| 15 AMS INCIDENT DESC | U | U | | | | U | | | | | | | | | | | |
| 16 INCIDENT STATUS | | | | U | | | | | | | | | | | | | |
| 17 APPARATUS STATUS | | | | | | | | | | | | | | | | | |
| 18 COMPANY STATUS | | | | | | | | | | | | | | | | | |
| 19 OCCUPANCY DESC | | | | | | | | | | | | | | | | | |
| 20 OCCUPANCY FIRE PREV | | | | | | | | | | | | | | | | | |
| 21 NOTICES | | | | | | | | | | | | | | | | | |
| 22 HYDRANTS | | | | | | | | | | | | | | | | | |
| 23 ARSON CAS | | | | | | | | | | | | | | | | | |

| DATA CLASS \ PROCESS | 18 DEVELOP CAPITAL IMP PLAN | 19 DEVELOP A & I PLAN | 20 PREPARE FINANCIAL REPORTS | 21 ISSUE & MAINTAIN REPORTS | 22 BILL FOR SERVICES | 23 PREPARE AND MANAGE PURCHASE ORDERS | 24 AUTHORIZE PAYMENT | 25 COMPENSATE | 26 RECORD TIME WORKED AND ACTIVITY | 27 FORECAST PERSONNEL REQUIREMENTS | 28 SCHEDULE PERSONNEL | 29 ESTABLISH STANDARDS | 30 CONDUCT EMPLOYEE RELATIONS | 31 MONITOR PHYSICAL CONDITION | 32 MAINTAIN PERSONNEL RECORDS | 33 RECRUIT PERSONNEL | 34 ASSIGN PERSONNEL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 ECONOMY/ENVIRONMENT | U | U | | | | | | | | | | | | | | U | |
| 2 OPERATING PROCEDURES | U | U | | U | U | U | | U | U | U | U | U | U | U | U | U | |
| 3 CODES & REQS | U | U | | U | U | U | | U | U | U | U | U | U | U | U | U | |
| 4 COMMUNITY | U | U | | | | | | | | | | | | | | | |
| 5 ACCOUNTS | U | | U | | U | U | U | U | U | | U | | U | | | | |
| 6 PERMITS | | | U | C | U | | | | | | | | | | | | |
| 7 AMBULANCE BILLS | | | U | | C | | | | | | | | | | | | |
| 8 PURCHASE ORDERS | | | | | | C | U | | | | | | | | | | |
| 9 VENDORS | | | | | | | U | | | | | | | | | | |
| 10 EMPLOYEE STATUS | | | U | | U | | | U | C | U | U | U | U | | | | |
| 11 ACTIVITY | U | | | | U | | | U | C | U | U | C | | | | | |
| 12 EMPLOYEE STAFFING | U | U | U | | | | | | U | C | C | | | | | | |
| 13 EMPLOYEE DESCRIPTION | | | | | | | | U | | U | | | | | | | |
| 14 INCIDENT DESC | U | U | | U | U | | | | | | | | | | | | |
| 15 AMS INCIDENT DESC | | | | | | | | | | | | | | | | | |
| 16 INCIDENT STATUS | | | | | | | | | | | | | | | | | |
| 17 APPARATUS STATUS | | | | | | | | | | | | | | | | | |
| 18 COMPANY STATUS | | | | | | | | | | | | | | | | | |
| 19 OCCUPANCY DESC | | | | | | | | | | | | | | | | | |
| 20 OCCUPANCY FIRE PREV | | | | | | | | | | | | | | | | | |
| 21 NOTICES | | | | | | | | | | | | | | | | | |
| 22 HYDRANTS | | | | | | | | | | | | | | | | | |
| 23 ARSON CAS | | | | | | | | | | | | | | | | | |

LEGEND: U = USES
C = CREATES

Figure 3.3: Example of BSP

Step 3 in the procedure is not a clear criterion. Therefore, this method includes a heuristic element. Block triangulation, which divides a matrix into blocks in which rows and columns of non-blank elements do not overlap, is a well-known algorithm, but the BSP method can also be seen as a heuristic method that allows some overlap between blocks while minimizing the range of overlap as much as possible.

## 3.3 The Significance of Requirements Engineering

### 3.3.1 The Importance of Requirements Engineering

It has been repeatedly said that development projects will fail if the requirements engineering process is not properly implemented and appropriate requirements specifications are not created, and many examples have been reported to illustrate this. One report that has been frequently cited is a report published in 1994 by the American research company Standish, although the data is a little old[124]. This report was greeted with surprise and has been frequently cited since then because it showed a low project success rate of 16.2%. The survey was conducted as a questionnaire targeting American companies, and was based on data from 8,380 system development projects provided by 365 respondents. 52.7% of projects were recognized as failures due to budget or deadline overruns, and a further 31.1% were discontinued midway. Respondents were also asked about the factors behind the success and failure of the projects. According to the survey, the third most common success factor was "clearly stated requirements," accounting for 13.0

The Standish Report is essentially a report for customers, and only a small portion of its contents is made public. Even in the latest 2014 edition, which was published as of May 2021, the figures presented only cite the results of a survey conducted in 1994. The Standish Report presents data suggesting the importance of requirements, but this is based on a survey of project managers, and can be considered subjective data.

There is also a study by Mayumi Kamata and the author that analyzes the relationship between the quality of requirements specifications and the success or failure of projects based on more objective data (Kamata & Tamai [74]). In this study, 32 development projects carried out by a certain company over a certain period of time were analyzed, and the results of the evaluation of the requirements specifications performed by the company's quality control group during the development period and the judgment of the project's success or failure performed by another evaluation group after the project was completed were statistically evaluated. The success or failure of the project was determined based on whether the budget and delivery date exceeded the plan, as in the Standish report. However, the quality of the requirements specifications was based on standards that had been cultivated by the company over many years, and a checklist consisting of more than 100 items for evaluating the quality of requirements was used, and it was recognized that there was a sufficiently high level of objectivity. The evaluation was done on a six-point scale from 0 to 5. The results were:

1. A relatively small number of requirements specification items have a large impact on the success of a project.
2. The requirements specification description of normal projects is well-balanced, with no variation in the details of the description items.
3. Chapter 1 of the requirements specification, which describes the purpose and overview of the system, is well written in normal projects, but poorly written in projects that are overcosted or overtime.
4. Projects with well-described "References" and "Objectives" in Chapter 1 tend to be completed on time.
5. Projects with poor descriptions in Chapter 1 and many descriptions in the "Functions" and "Product Overview" sections tend to run overtime.

These are some of the findings we gained. Here, "Chapter 1" refers to the corresponding check items in the IEEE standard for writing software requirements specifications, “Recommended Practice for Software Requirements Specifications” [62], and the structure of the recommended requirements specification (SRS) is three chapters as shown in Figure 3.4.

As the importance of requirements engineering is gradually recognized, agile processes as described in the previous chapter have attracted attention. The aim of agile processes is to prevent project development time and costs from exceeding the schedule, and to avoid project failures such as those exposed by the Standish report by shortening and repeating the development cycle. Therefore, the requirements analysis process is also shortened. Methods such as emphasizing test-driven instead of requirements specifications and describing requirements as user stories are also used, but the importance of capturing what users actually want as accurately as possible remains unchanged. Regarding the fact that agile processes and requirements engineering are not necessarily in opposition, Axel van Lamsweerde states the following in his book *Requirements Engineering*[137], which systematically develops requirements engineering from a theoretically clear and practically useful perspective:

"Agility is not a binary concept. Depending on the extent to which assumptions (such as the fact that one person can represent the project, the size of the project is small enough, etc.) suitable for agile development are met, more or less agility can be introduced into each phase of the requirements engineering cycle (elicitation, evaluation, specification, and synthesis), and the duration of each phase can be extended or shortened."

1. Introduction
    1.1 Purpose
    1.2 Scope
    1.3 Definitions, Acronyms, and Abbreviations
    1.4 References
    1.5 Overview
2. Overall
    2.1 Product Overview
    2.2 Product Features
    2.3 User Characteristics
    2.4 Constraints
    2.5 Assumptions and Dependencies
    2.6 Priority of Requirements
3. Specific Requirements
    3.1 External Interfaces
    3.2 Individual Functions
    3.3 Performance Requirements
    3.4 Logical Database Requirements
    3.5 Design Constraints
    3.6 System Attributes of Software
    3.7 Organizing Specific Requirements

Figure 3.4: Recommended SRS Structure (IEEE Std. 830-1998)

### 3.3.2 History of Requirements Engineering

Although requirements engineering is included as a part of software engineering, it has a long history since it was proposed, and its practical importance is widely recognized, so it has a firm position. Back in the mid-1970s, when structural programming was in vogue, several techniques were proposed under the name of requirements engineering. The most representative ones are as follows:

1. PSL/PSA
   This is a method that combines the PSL language and the PSA analysis tool, developed in the ISDOS project by Daniel Teichrow (University of Michigan) and others.

2. SREM
   This is a method developed at TRW, and uses the RSL requirements definition language, the R-Net process flow representation, and the REVS analysis tool. It is particularly suited to real-time systems.

3. SADT
   This is a method developed by D. T. Ross and others at Softech, and represents tasks or actions as boxes, and the input, output, control data, and resources used are represented by arrows from the left, right, top, and bottom. SADT was later published in 1993 by the National Institute of Standards and Technology (NIST) as the IDEF0 (Integration Definition for Functional Modeling) standard.

These papers were published in the special issue on requirements analysis and definition in the IEEE Transactions on Software Engineering, Vol.SE-2, No.1, 1977l

These were then aggregated into structured analysis and design techniques, but with the rise of CASE tools in the late 1980s, they were brought back into the spotlight.

In addition, the International Conference on Requirements Engineering began to be held regularly in the 1990s, and the field of research has flourished once again, to this day.

### 3.3.3 The concept of requirements engineering

The first step in requirements engineering is the process of gradually clarifying what you or the software purchaser intends and requires from a vague and chaotic state. This process is often called requirements analysis, problem analysis, or simply analysis. There are various ways of thinking about requirements analysis, but let's consider them below by classifying them into three types.

1. Requirement elicitation type
   First, assume that there are user needs and intentions, and the center of requirements analysis is the task of extracting them as requirements. In other words, it is assumed that there are people who have requirements, even if only latent. One of the basic methods is to interview users, and in this case, a method using scenarios is often used. The user and the system developer jointly create a scenario for using the system to be built. [108]. A more specific description of this is also called a use case, and is used as the basis for defining the system specifications. This method is characterized by its ability to respond to users in a detailed manner, and also makes it possible to clarify the image of how the system will be used at an early stage. The weakness is that it depends heavily on the person from whom the requirements are extracted, and the requirements may change significantly when the person changes. Therefore, it is recommended to extract requirements from as many parties as possible, but this can lead to scattered requirements and, in some cases, conflicts between requirements, making it difficult to adjust them. It tends to be difficult to summarize requirements that are consistent overall.

needs
intension
elicit
Req's

Figure 3.5: Requirements elicitation-based requirements analysis

2. Goal-oriented
   The idea is that the system is developed to achieve some goal, so if the goal is structurally clarified, the requirements for the system will become clear. Since it is the user who consciously has goals, interviewing users is a natural method. However, it is also possible to decompose and integrate goals in a fairly systematic and objective way. That is, a goal is decomposed into smaller goals to satisfy it. An and connection is possible, in which the original goal is satisfied if all of the decomposed goals are satisfied, and an or connection is possible, in which the original goal is satisfied if any one of the decomposed goals is satisfied. Conversely, a hierarchical structure between goals is created as a hierarchical directed graph by asking what the goal was created to satisfy and combining it into a larger goal. The goal at the bottom of the goal hierarchical graph created in this way becomes the requirement for the system[136, 137].

   In fact, such methods have long been devised as a general problem discovery and problem solving method in fields such as operations research and management science, and in the field of requirements engineering, it has been further refined and tools to support it have been created.

   The feature of this method is that it can obtain a set of requirements that is more consistent overall than the requirements elicitation type. However, its weakness is that systems created based on this method tend to have a structure that is specialized for a set goal, and while it is extremely effective in solving that goal, it tends to lack flexibility when the problem changes slightly.

3. Domain model type
   First, there is the target domain, which can be called the problem domain, application field, business, target world, etc., and this is mapped onto some kind of model. The view is that requirements and goals are added to the model later.

   Object-oriented models and other models are used as modeling techniques.

   This method is effective when performing requirements analysis for many slightly different systems in a series of product lines at once. In addition, since the target domain is relatively stable compared to functional requirements, a

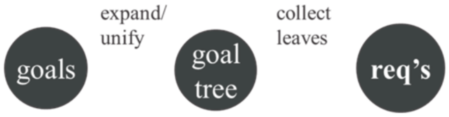


Figure 3.6: Goal-oriented requirements analysis

system that reflects the model of the target domain has the characteristic of being able to flexibly respond to changes in requirements. On the other hand, the weakness of this method is that it is difficult to clearly image the system, and the boundary between the inside and outside of the system to be created within the problem domain tends to be unclear.

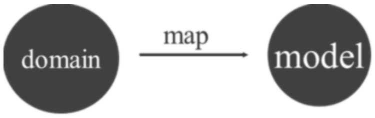


Figure 3.7: Domain model-based requirements analysis

These methods are not necessarily carried out independently, and it is often more effective to use them in combination. However, it will be necessary to select and combine the appropriate ones depending on the characteristics of the target system and the development organization.

We will explain scenarios and goal deployment using examples.

**Drink vending machine scenario** Let's imagine that we are developing a new drink vending machine like the ones we see around town.

The following is one possible scenario for a user to use a vending machine.

1. The user puts in money with bills or coins, or both.
2. The amount put in is displayed, and the lamp corresponding to the drink whose price is equal to or exceeds that amount lights up.
3. When the user presses the button of the lighted lamp, one drink is dispensed into the dispenser, and the price of that drink is deducted from the displayed amount.
4. If there are drinks whose lamps are still lit, the user can press that button to continue taking out drinks.
5. The user can return to step 1, put in more money, and continue with steps 2 and onward.
6. If the user has any money left, they press the return button to receive their change.

This scenario is for normal operation, and it is necessary to prepare separate scenarios for special situations such as ending without dispensing any drinks, forgetting to take the change, or running out of drinks. Also, as can be seen from the way the scenario is described, a scenario consists of a series of action events, and the action can be seen as an interaction between several subjects. In this example, the action subjects are the user and the vending machine.

**Goal development of sake store warehouse management** Imagine a problem domain of inventory management in a sake store warehouse. Each square box in Figure 3.8 represents a goal. Vertical lines connect the higher-level goal to the decomposed goal. In the figure, arrows with a minus sign indicate that the effect is negative and there is a conflict between the goals. In this example, there is no or expansion, but all and expansion.

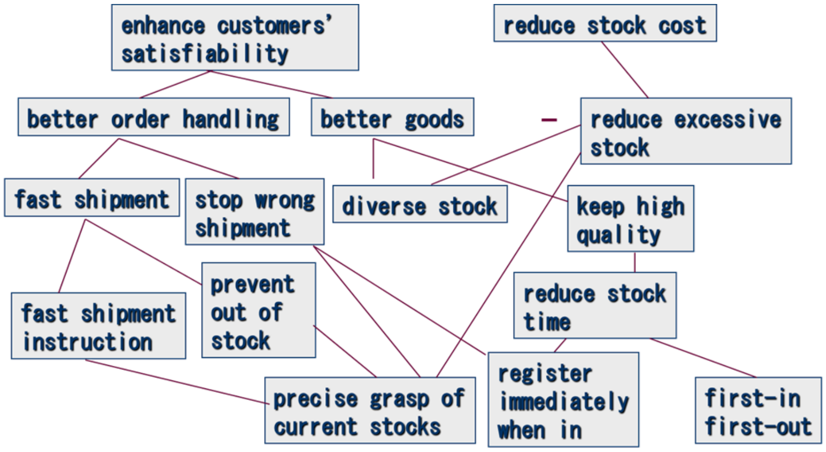


Figure 3.8: Goal development of sake store warehouse management

## 3.4 Requirements Engineering Process

The requirements engineering process is generally said to be a repeated cycle of four phases, as shown in Figure 3.9.

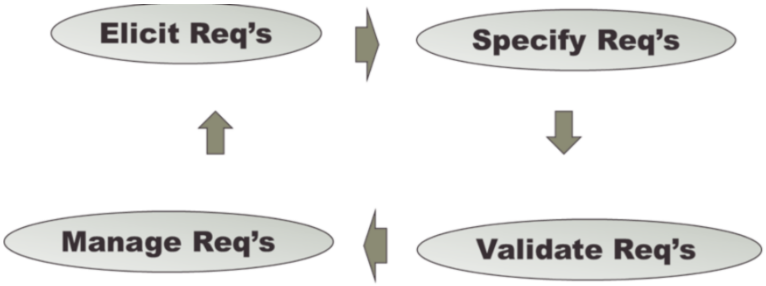


Figure 3.9: Requirements Engineering Process

An overview of each phase is as follows.

1. Requirements Analysis
   Requirements for the newly developed system are extracted through interviews with stakeholders, research into laws and regulations and related documents, and research into past similar systems and current related systems. If there are multiple stakeholders and there are conflicts of interest, they are reconciled through negotiation.

2. Requirement description
   The extracted requirements are compiled and organized, contradictions are removed, and described as requirement specifications. Before defining the requirement specifications, the requirements may be modeled.

3. Requirement verification
   The requirement specifications are verified from the viewpoints of whether they are described correctly, whether there are omissions, whether there are mutual contradictions, and whether they are feasible. Verification methods can be both mechanical and manual. If necessary, a prototype can be created and the user can test it to verify the requirements.

4. Requirement management

Manages the addition and change of requirements. Also, tracks the correspondence between the implemented program and the requirements.

**Stakeholders** The stakeholders from whom requirements are extracted in the requirements analysis phase refer to people who have various interests in the system to be developed, and examples include the following people:

1. Client: The person who pays for the development
2. Customer: The person who buys the developed product
3. User
4. Development engineers
5. Managers
6. Marketing staff
7. Testers

And so on.

Identifying these stakeholders, eliciting as many requirements as possible from them, and prioritizing those requirements is an important role of requirements analysis.

## 3.5 Types of requirements

### 3.5.1 Functional requirements and non-functional requirements

There are functional and non-functional requirements. Functional requirements refer to the functions that the system to be developed should have, and are the most natural intuitive association with the word ”requirements.” On the other hand, non-functional requirements include various properties that the system should have, constraints that must be observed in satisfying functional requirements (including those required by organizational culture and laws), conditions imposed on the system development method, etc. However, there is criticism that the definition using the negative term ”non-functional requirements” is vague in its scope. Therefore, it has been proposed to use the term ”quality requirements” instead[20].

From another perspective, some say that quality requirements are part of non-functional requirements[137].

To more clearly express the difference between functional and non-functional requirements, functional requirements have a binary nature of being either satisfied or not satisfied, whereas non-functional requirements have a binary nature of being either satisfied or not satisfied. It is easier to understand non-functional requirements if you explain them as a continuous scale of satisfaction, ranging from sufficient to insufficient. Therefore, non-functional requirements are essentially linked to quantification.

Another aspect of non-functional requirements is that, while functional requirements are realized by a specific module, non-functional requirements are cross-module, and several modules are involved in their realization. One concept that emerged as a programming methodology is ”aspect-oriented” (see Section 9.2.2). Aspects are considered to correspond to ”cross-cutting concerns,” and the idea is to take properties that cross modules that are arranged one-dimensionally in normal programs, and realize a mechanism within a programming language to explicitly describe them. Although the concept originally came from the programming level, it is also meaningful in the analysis and design phase, and it is recognized that it has something in common with non-functional requirements[111].

Even among functional requirements, conflicts and incompatibility often occur, and how to reconcile them is one of the important issues in requirements engineering. Non-functional requirements are even more likely to cause such conflicts. For example, the pursuit of performance comes at the expense of safety. It is important to recognize and clarify such conflicts as early as possible during the requirements analysis stage.

### 3.5.2 Quality Requirement Model

When non-functional requirements are considered as quality requirements, the quality required for a system includes various things such as performance-related things like processing speed, ease of use, security, availability, and maintainability. Attempts to organize and systematize these have been made for a long time, but the ISO/IEC standard 25000 series is currently widely known. The series is divided into the ”quality management section,” ”quality model section,” ”quality metric section,” ”quality requirements section,” and ”quality evaluation section.” In particular, the 25010 standard for the quality model section defines the standard for software and data quality, and is also called SQuaRE (System and Software Quality Requirements and Evaluation).

Figure 3.10 shows the software product quality model in ISO/IEC 25010.

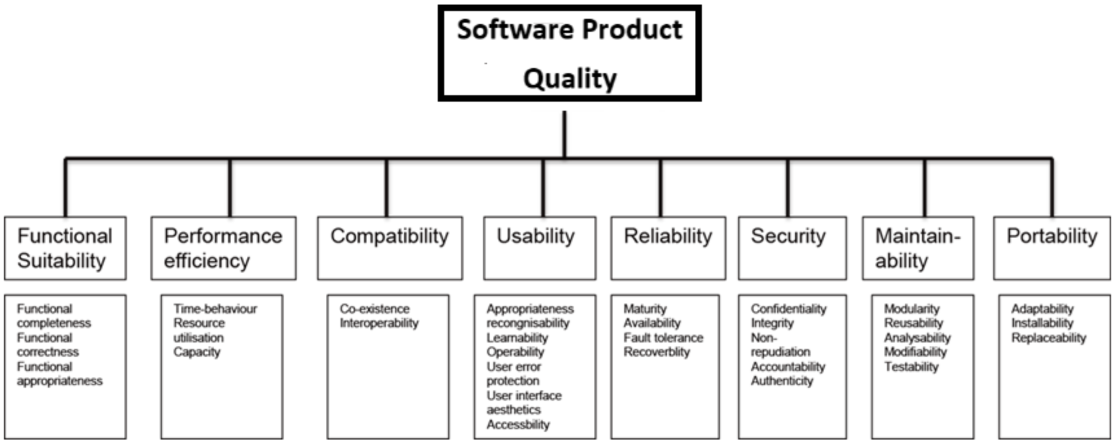


Figure 3.10: JSO/IEC 25010 ”system/software product quality” model

## 3.6 Specification

Requirements specification is a rigorous definition and description of requirements. While diagrammatic models are often used in the requirements analysis stage, specifications are usually written in natural language or textual formats such as formal specification languages.

### 3.6.1 The relationship between specifications and requirements

If specifications are strictly described as requirements, then specifications and requirements are essentially the same, but there are also schools of thought that make a clearer distinction between them. For example, Michael Jackson says that requirements describe phenomena in the environment outside the system, while programs describe phenomena within the system, as diagrammed in Figure 3.11. He then says that specifications describe phenomena (events and states) that exist on the boundary between the inside and outside of the system and are shared by both.

[69]

Then, the requirements are satisfied when the properties of the specifications and the external world are assumed, and the specifications are fulfilled when the properties of the software and the machine on which it runs are assumed. This can be written logically as follows:

$$\begin{aligned} Specifications, propertiesoftheenvironment &\vdash Requirements \\ Program, propertiesofthemachine &\vdash Specifications \end{aligned}$$

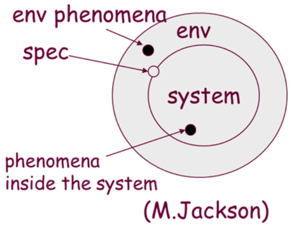


Figure 3.11: One way of looking at requirements and specifications

In other words, it must be possible to show that **requirements** are satisfied when the properties of the environment surrounding the system and **specifications** are assumed, and that **specifications** are satisfied when the properties of the machine (computer) on which the program runs and **program** are assumed.

### 3.6.2 What to express as a specification

In response to functional and non-functional requirements, specifications can also be

- Functional specification
- Performance specification
- Reliability specification
- Interface specification

and so on. Even when we limit ourselves to functional specifications, it is not easy to write them precisely.

For example, it may be relatively easy to clearly write functional specifications for mathematical software. Consider a requirement to find the greatest common divisor of two given positive integers. If the person receiving this requirement (for the time being, let's call them the software designer/developer) knows the concept of the greatest common divisor, then the requirement definition is already complete. If not, then a definition of the greatest common divisor (the maximum value that is a divisor of both of the two numbers) can be given. If you are a fastidious person who thinks that this expression is still not precise enough, you can use an expression based on the axioms of number theory.

Unfortunately, in many cases, specification description is not this simple. The reasons for this are as follows:

1. The problem domain in question does not have a clear logic like the world of mathematics.
2. The ”user” himself is not clearly aware of what he wants done, or even if he is aware, he cannot express it clearly.
3. The user's intentions change. Or, the intentions are inevitably changed by changes in external conditions.
4. The problem in question is so large and complex that it is difficult to know how to describe it.

In addition, even for problems with the greatest common denominator, the requirements mentioned above are insufficient. For example,

1. The environmental conditions under which the software will be used (what kind of computer it is, where it is located, who will use it and for what purpose, etc.) are not stated.
2. The form of the input and output interfaces is not stated.
3. It is not stated how it will operate in abnormal situations (for example, when a negative number is input, etc.).

### 3.6.3 Who writes specifications and who uses them?

There are two types of people who feel the need to write requirements as specifications. One is the client who communicates the requirements to others and requests the production of software. The other is the person who produces the software, who writes specifications to confirm the client's intentions or, in the case of market-oriented products, to clearly show the functions and operations of the product. The written specifications act as requirements that the output (design documents and programs) must satisfy in the subsequent design and development process.

Specifications also serve as a contract between the client and the contractor, which often makes specification writing even more difficult. This is because, since it is a contract, it is thought that if even a single word is neglected, it may cause a painful experience later. In some cases, the client and the contractor are the same person. Even in that case, it is essential to properly describe the intentions as specifications in order to create a high-quality program.

Specifications can be read by users, designers and developers, and inspectors. Depending on the subject, it may be possible to change the content of the description (description of user needs, content to be designed, inspection/verification criteria) and format, but there are few examples of this being put into practice.

The inspector here refers to the person who inspects whether the finished product meets the specifications. The user himself may also be the inspector, but in most cases, an independent staff member dedicated to inspection is employed.

### 3.6.4 Conditions for a good specification

A good specification must satisfy the following conditions.

- Validity
  This includes consistency (consistency) in that the content is self-consistent, and completeness (that everything is described without omissions).

- Strictness
  There is no ambiguity. The content is uniquely defined.

- Readability
  Easy to read and understand.

- Verifiability
  Consistency can be verified, or completeness can be checked in some way. It is even better if this can be done automatically or mechanically. It is also important that a system developed based on a specification can be verified to see whether it satisfies the specification.

- Feasibility
  Specific implementation is possible. No matter how valid and strict a specification is, if it is not feasible, it is meaningless as a specification. On the other hand, it is usually undesirable to write a specification that depends on a specific implementation method and restricts the design.

The problem, however, is that there is often a trade-off between these conditions. For example, the pursuit of strictness generally makes the text harder to read.

# Chapter 4

# Modeling and UML

In the chapter on software processes, we used the term process model.

In requirements analysis, we introduced a method of modeling the target domain, but even in the approaches of the requirements elicitation-type and in the goal-oriented requirements analysis type, introduced as contrasing types, it is oonsidered that requirements models are created according to each method.

In this chapter, we will not limit ourselves to requirements models, but will describe the general characteristics of modeling techniques used in a wide range of software development.

## 4.1 What is a model?

The discussion of "models" by M. Jackson in his book [69] is very thought-provoking. First, Jackson points out that the relationship between the real world and the abstract world is completely reversed between models in mathematics and logic and models in software engineering. Indeed, in mathematics and logical systems, a model is something that is the realization of a theory. For example, for the theory of abstract algebra called groups, integers with addition defined are called one of its models. Conversely, in the world of software, a model is usually something that abstracts a real-world problem domain and expresses it in some kind of descriptive system. This point is important. Moreover, formal methods, which are a major part of software engineering, overlap with formal logic, so there is a high possibility of confusion.

Jackson argues that software engineers should not use the word model in the same sense as it is used in logic, but he also questions whether it is appropriate to call an abstract description a model. He argues that abstract descriptions should simply be called descriptions.

Jackson's position is that a model should be something that has a meaningful physical existence in itself and behaves similarly to a part of the real world in question. For example, if the behavior of a fluid flowing through a network of pipes is modeled by an electrical circuit, then the electrical circuit is a physical entity and a model that is similar to the domain of the fluid. A model created in a computer is a physical entity that exists and at the same time behaves in a way that imitates the real world, so it is exactly what Jackson means by a model.

Jackson is right, but in software engineering, abstract descriptions themselves have always been called models. Data flow models, ER models, and state transition models, which will be discussed in this book later, are all abstract descriptions. Here, following the conventions of traditional software engineering, I will call these abstract descriptions models. In other words, a model is an abstraction of a structured object in order to understand its properties and behavior. Of course, it is always important to keep in mind that the model will eventually be realized as software that runs on a physical entity called a computer.

## 4.2 Models in Software

### 4.2.1 How the Term Model is Used

The word model is loved in the world of software engineering. For example, there is a survey by Torii Koji that looked at the frequency of words that appeared in the titles of papers from the first International Conference on Software Engineering (ICSE) held in 1975 and in the successive first 20 year conferences[135]. According to this, the frequency of the word software was by far the highest, but excluding this, the order was as follows:

1. system, 2. design, 3. specification, 4. model, 5. analysis

The word model is not only ranked 4th, but also has the characteristic of being used continuously for 20 years from 1975 to 1994. For example, design, ranked 2nd, appeared more frequently in the first half of the 20 years and decreased in the latter half, but model hardly showed any such fluctuation.

This is probably because ”model” has been used in conjunction with other terms. The terms that it can be combined with are numerous, such as:

> life cycle, design, system, analysis, process, product, quality, domain, object, computation, data, data flow, entity-relationship, function, application, state transition, ...

While these combinations have gone in and out of fashion, model has remained popular throughout.

The entry for ”model” in Jackson's ”Lexicon” [69] lists how the word model is used not only in the world of software but also in other areas as a nice-sounding word.

> Model as an example (the Shogun's model)
> Model of a new car
> Economic model, ship model, architectural model, mathematical/logical model, plastic model

On the other hand, when you look at the entry for ”model” in the Kojien dictionary,

> type, model, prototype, example
> A person who is the subject of an artist's work
> A real person who is the subject of a novel, play, etc.
> Abbreviation for fashion model

It seems that, unusually for a foreign word, there is almost no difference from the original usage.

### 4.2.2 Models and Abstraction

In engineering, modeling can be done either by constructing a model of the domain itself to understand the target problem domain, or by constructing a model before actually developing the system, structure, or product that will be realized on that domain. The two are closely related, and it is sometimes possible to express them simultaneously within the same framework, but in many cases it is better to distinguish between them.

As already mentioned, the key to modeling is abstraction of the target. Abstraction means discarding non-essential things. In Japanese, there is a convenient word to express this: ”shashou.” Models are generally expressed as components, the relationships between them, and the mechanisms of their operation, but the appropriate model expression is determined by what is focused on in the target, and what components, relationships, and operations are selected.

Below is a classification of modeling techniques commonly used in system analysis and design, based on what structure or properties of the target are focused on in the abstraction.

1. Abstraction of events (behavior)

2. Abstraction of things and relationships

3. Abstraction of information flow

4. Abstraction of functions

These techniques focus on very different objects, but when expressing them as diagrams, they share a common format called a graph structure. In the next section, we will consider the graph representation of models from a general perspective, paying attention to their similarities and differences.

## 4.3 Graph-based modeling

### 4.3.1 Graphical representation of various models

Many modeling techniques use some kind of diagrammatic representation. Examples of diagrams include the following. Here, we include not only models used during analysis, but also diagrams of models used in system design and program design.

1. Those that show control flow (flowcharts, PAD, HCP, SPD, etc.)
2. Those that show data flow (data flow diagrams, SADT, HIPO, etc.)
3. Those that show data structures (Jackson structure diagrams, etc.)
4. Those that show data-related structures (ER diagrams, semantic networks, object model diagrams, etc.)
5. Those that show state transitions (state transition diagrams, Statecharts, Petri nets, etc.)

All of these have the common feature of being composed of a group of closed figures, such as circles, squares, or boxes, and a group of lines, such as broken lines, dotted lines, or arrows, that connect them.

Mathematically speaking, they have in common the fact that they have a graph structure consisting of a set of vertices and a set of edges.

The following factors are thought to be the reasons why graph structures are so widely used.

1. By visualizing it as a diagram, it looks like a ”model,” and is easy to draw and intuitively understand.
2. The world is naturally expressed by a graph, with vertices representing the ”things” in the target world and edges representing the relationships between the ”things.”
3. There are often transitive laws or some kind of transitive relationship between ”things,” which can be understood by tracing paths on a graph, and path-finding algorithms have also been well studied.

However, the fact that it is intuitively easy to understand also encourages arbitrary and ambiguous use. In fact, when the concepts of state transition diagrams and data flow diagrams are explained to students, they seem to understand relatively easily, but when they try to draw a diagram, they often end up creating something that does not represent state transitions or data flows at all. In fact, this is not only the case with students, but also with quite experienced software engineers.

The use of graph structures as diagrams is not limited to the software world. Many diagrams appearing in newspapers, magazines, textbooks, reports, proposals, papers, presentation slides, etc. show graph structures for some representations. However, the meaning of these diagrams is often ambiguous. For example, we often see diagrams in which it is unclear what the arrows represent, whether they represent causal relationships, time relationships, material flows, or control flows. The same is true for boxes, where it is common to see a mixture of processing subjects such as processes and processing targets such as data represented by the same box or circle.

Because graphs are so intuitively easy to understand, people tend to interpret their semantics ambiguously. Therefore, it is important to consciously consider the commonality of the same graph structure and the differences in meaning.

### 4.3.2 Characteristics of modeling using graph representation

As already mentioned, in graph representation, ”things” (concepts, objects, processes, data, etc.) are usually connected to vertices, and relationships between things are connected to edges. Depending on the role of the edges, models can be classified into static models that represent structure and dynamic models that represent behavior.

1. Static model

   Edge connecting vertices A and B ⇒ Relationship between A and B
   If the edge is undirected, it indicates that ”A and B have a certain relationship,” and if it is directed, it indicates that ”A has a certain relationship with B.” Representative examples include ER (entity-relationship) models, class diagrams, and semantic networks.

2. Dynamic model

   Edge from vertex A to B ⇒ Movement from A to B
   Edge is directed. There are cases where the viewpoint of control, etc., moves from A to B (control flow, state transition, etc.), and cases where data or objects flow from A to B (data flow, workflow, logistics, traffic flow, etc.).

Static models and dynamic models are rarely confused, but it is common to see confusion among dynamic models, such as data flow and control flow, or state transition and flowchart.

Table 4.1 shows the combinations of vertices and edges in typical model diagrams.

Table 4.1: Graph structure of a typical model

| Model | vertices | edges |
|---|---|---|
| Data flow | processes | data flow |
| ER | entities | relationships |
| State transitions | states | transitions |
| JSD (Jackson method) | processes | data stream connections<br>data vector connections |
| Flowchart | execution units | control flow |
| Petri net | places, transitions | firings and token flow |

### 4.3.3 Graphs and physical quantities on models

Graphs represent the topology of the relationship between vertices and edges. However, when modeling uses graphs, it is common to associate some physical quantity with the vertices and edges. I say physical quantity, but since we are dealing with ”information” here, it is difficult to call them physical quantities in many cases. However, graph models that target typical physics such as electric circuits provide many suggestions. In the case of electric circuits, potential is associated with vertices and current with edges. Voltage, which is the potential difference, also corresponds to edges. Kirchhoff’s Law is a law that relates this physical quantity to topology.

1. Kirchhoff’s Current Law: The total amount of current flowing into a vertex (measured as inflow positive and outflow negative) is zero. Alternatively, it can be said that the current flows continuously along each edge without stagnation or interruption.

2. Kirchhoff’s Voltage Law: The sum of the voltages along a closed circuit is zero.

In this context, there are two types of edge connections: series and parallel. The physical properties of each connection can be described as follows (see Figure 4.2). This is derived from Kirchhoff’s Law.

1. Series: Current is constant. Voltage is additive

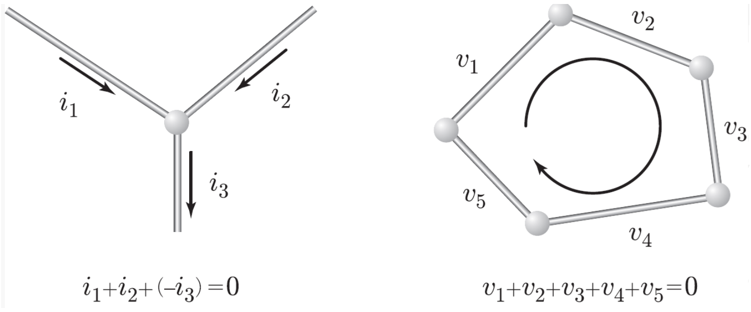
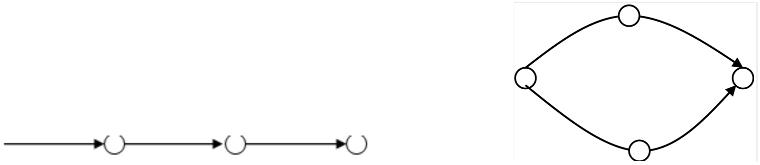


Figure 4.1: Kirchhoff's Law



(a) Series (b) Parallel

Figure 4.2: Series and Parallel

2. Parallel: Voltage is constant. Current is additive

There are many models in which ”flowing things” and the pressure that drives them are connected to edges and vertices, respectively, but not all models fit this description. However, like Kirchhoff's laws, it is fundamental to

1. Consider the properties of all the edges entering and leaving each vertex;
2. Consider the properties that hold along the paths that the edges connect;

For example, in the case of a data flow model (chapter 5), you can see the data conversion function of a process by matching the data entering the vertex, which is the process, with the data leaving it. You can also understand the flow of a series of data processing by tracing the flow of data along the path.

### 4.3.4 Application of Basic Concepts in Graph Theory

You don't need something as grand as graph theory, but basic concepts like those found at the beginning of a textbook are useful.

First, there are several types of graphs that can be considered. Depending on which type, the properties of the model will differ.

- Directed graphs/undirected graphs
- Trees, Directed Acyclic Graphs (DAGs), General Graphs

  Tree structures correspond to hierarchical structures, and acyclic structures correspond to hierarchical structures with a common substructure. General graphs correspond to so-called network structures that do not have a hierarchical structure.
- Partition of vertex sets (e.g. Petri nets), Partition of edge sets (e.g. PAD)

A graph that is part of a graph and has a subset of the edge sets of the original graph as its edge set and both ends of those edges as its vertex set is called a subgraph of the original graph. If a subgraph is reduced to a single vertex, it becomes an encapsulated and abstracted graph (Figure 4.3). This represents a recursive structure in which the vertices are further refined and still have a graph structure. This provides a natural framework for dealing with hierarchies of abstraction. In fact, this method is used in data flow diagrams and statecharts.

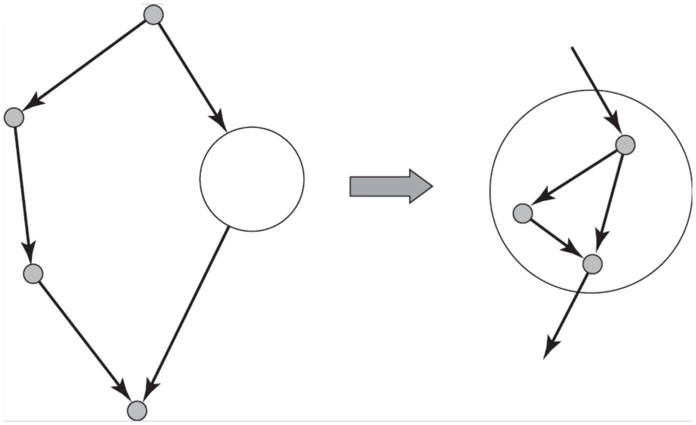


Figure 4.3: subgraph

### 4.3.5 Weaknesses of graphs and countermeasures

Even if in principle any complex model can be illustrated as a graph, the number of vertices and edges that can be drawn on one diagram is actually limited. One reason is physical constraints: paper and display screens have a limited area, so no matter how much you pack in, you can only show a limited amount. There are also cognitive constraints. As George Miller's often-quoted "magic number 7 ± 2"[89] asserts, the number of things that humans can naturally recognize is surprisingly small. Even if the number of vertices is kept to a certain level, diagrams with a large number of edges that look like a spider's web are very difficult to understand.

This is a weakness of model representation using graphs, but there are several ways to deal with it, as follows.

- Layout techniques: Various algorithms have been proposed to make graphs easier to read, such as arranging the edges to minimize crossings, and other criteria.
- Screen operation techniques: Zooming and scrolling are used.
- Hypergraphs: As a generalization of graphs, hypergraphs, which treat edges as relationships between multiple points rather than between two points, can be used to simplify diagrams in some cases.
- Hierarchy: As already mentioned, by introducing a hierarchical structure in which the vertices of a higher-level graph are expanded to become lower-level graphs, diagrams at the first level can be simplified.

  It is also easy to support this hierarchical structure with tools.

  Relatedly, a commonly seen technique is to make it possible to draw nested structures in the boxes of vertices.

## 4.4 UML notation

UML (Unified Modeling Language) is a "language" designed to unify many diagram-based notations used in various object-oriented analysis/design methodologies and provide a standard syntax and semantics. Although it is a language, its main components are graph-like figures rather than text. The latest UML language specification is available from the publisher OMG [102].

### 4.4.1 History

Many object-oriented methodologies were published from the late 1980s to the mid-1990s, and the two most widely accepted were the Booch method by Grady Booch and the OMT method by James Rumbaugh and others. When Rumbaugh moved to Rational Software, where Booch was, efforts began to merge the two and create an integrated methodology, and a provisional version was published in October 1995 as the Unified Method. Around the same time, Ivar Jacobson, who had also popularized his own methodology OOSE, joined Rational with his own company Objectory, and OOSE was incorporated into this integration. As a result, UML, a compilation of notations used in object-oriented methodologies, was announced as UML 0.9 in October 1996. The OMG (Object Modeling Group), which evaluated the influence of UML, called for proposals for a standard modeling language. In response, many companies, including IBM, HP, TI, and Microsoft, participated as partners in addition to Rational, and compiled and proposed a specification called UML 1.1, which was adopted as a standard by OMG in November 1997. Since then, revisions to UML have been carried out at OMG, and the 2.5.1 version was announced in May 2017.

### 4.4.2 Overview of UML

UML uses a notation centered on diagrams to represent various aspects of object-based models. Diagrams can be broadly divided into those that represent the static structure of objects and those that represent their dynamic behavior. Table 4.2 shows the 14 types of diagrams defined in UML 2.5. Those classified as structural diagrams are static models, and those classified as behavioral diagrams are dynamic models.

Table 4.2: UML diagrams

| | | |
|---|---|---|
| Structure Diagram | Class Diagram<br>Object Diagram<br>Component Diagram<br>Package Diagram<br>Deployment Diagram<br>Composite Structure Diagram<br>Profile Diagram | |
| Behavior Diagram | Activity Diagram<br>Use Case Diagram<br>State Machine Diagram | |
| | Interaction Diagram | Sequence Diagram<br>Communication Diagram<br>Interaction Overview Diagram<br>Timing Diagram |

UML defines 14 diagrams, and it is not necessary to use all of them. In this book, UML is considered to be a compilation of past analysis and design know-how centered around diagrams. From that perspective, it is sufficient to select and use the various diagrams of UML as appropriate, and it is not necessary to use all the notations defined in UML for a single diagram. This book also focuses on the graph structure of model representations, and considers UML as a catalog of model representations with graph structures, paying special attention to the commonalities and differences among those graph representations. The graph structure of a typical UML diagram looks like a table.

Table 4.3: Graph structure of UML diagrams

| | vertices | edges |
|---|---|---|
| Class diagram | classes | generalization, aggregation, association |
| State machine diagram | states | transitions |
| Activity diagram | activity states | control flow |
| Communication diagram | objects | message flow |
| Series diagram | message sending and receiving points | message flow |

From Chapter 5 onwards, we will explain various modeling techniques from the perspective of what aspect of the subject they focus on. In the process, we will take up UML diagrams that are suitable for each technique.

## 4.5 Common Example

We will use common examples to explain and compare the modeling techniques we will discuss below. One is a typical warehouse management problem for business systems, and the other is a typical vending machine problem for reactive systems.

### 4.5.1 Sake Store Problem

In 1984, the Software Engineering Study Group of the Information Processing Society of Japan created an example called the ”Sake Store Inventory Problem” to compare various design techniques (created by Toshiharu Yamazaki (then Nihon Unisys)). Since then, this example has been used in many papers and research presentations, but the most useful literature for comparing various design techniques side by side is the special feature that was compiled in three issues of the academic journal ”Information Processing” (”Information Processing” Vol. 25 (1984) No. 9, No. 11, Vol. 26 (1985) No. 5).

It is valuable to be able to see nearly 20 different techniques side by side using the same example. Moreover, not only are the authors all experts in their respective techniques, but I was impressed by how seriously they all tackled the problem. The problem is well-constructed. It is typical of the field of business management but not self-evident, and is presented clearly and in short sentences, while also including a moderate amount of ambiguity.

First, let me quote the problem in its original form.

> At a certain sake sales company’s warehouse, several containers are delivered every day. They contain bottled sake, and one container can carry up to 10 brands. There are about 200 brands available. The warehouse clerk receives the container, stores it in the warehouse, and hands the shipping slip to the receptionist. The contents are then shipped out when the receptionist gives an order to ship them out. The contents are not transferred to another container or stored in another location.
>
> Empty containers are immediately removed.
>
> Shipping slip: Container number (5 digits)<br>
> Date and time of delivery<br>
> Item name and quantity (repeated)
>
> Now, the receptionist receives dozens of shipping requests every day and issues a shipping instruction to the warehouse clerk each time. Shipping requests are made by shipping request slip or telephone, and each request is limited to one brand. If there is no stock or the quantity is insufficient, the receptionist will call the requester and enter the information in the inventory shortage list. Then, when the required amount of the item is loaded, the receptionist will issue a shipping instruction for the insufficient item. The receptionist will also notify the warehouse clerk of the containers that will be emptied.
>
> Shipping request: Item name, quantity<br>
> Destination name
>
> Create a computer program for the receptionist’s work (notifying of out-of-stock items, creating shipping instructions, and creating the inventory shortage list).
>
> Shipping instructions: Order number<br>
> Destination name<br>
> Container number<br>
> Item name, quantity (Repeat)<br>
> Empty container removal mark
>
> Inventory shortage list: Destination name<br>
> Item name, quantity

- No loss of alcohol occurs during transportation or storage in the warehouse.
- Some parts of this assignment are unrealistic, so input data error processing, etc. can be handled in a simplified manner.
- Please interpret the ambiguous points above as appropriate.

The problem is written clearly, but it is not as simple as it seems. There is a part in the description that is a little difficult to understand. It is the part that says, ”Then, when the required amount of the item is loaded, an instruction to ship the missing item is issued.” If you do not read carefully, you will not understand the meaning of ”an instruction to ship the missing item.” How can it be shipped if there is a shortage?

If you read it again, you will see that this sentence, following the sentence just before it, is dealing with a case where there is a shipping request but there is a shortage of stock. It is talking about a time when there was a shortage at one point, but then the stock arrives and the shortage is met. The reason this is not so straightforward is because it does not explicitly state the case distinction between issuing an shipping instruction immediately if there is stock, and contacting the requester without issuing an shipping instruction if there is a shortage.

In fact, the problem was not originally described like this. The sentence ”And the item in question...” was not in the original problem, but was added when the revised version was released. The reason for adding this was that the people who first worked on the problem pointed out that it was unclear how to handle shortages: should they just notify the customer of the shortage, should they ship what they have in stock even if it is not enough to meet the order amount, or should they ship it later when the stock is full? This sentence was added in response to that. Those who know the background will be able to read this sentence easily, but those who first saw the revised version will find it difficult to understand. This may be said to be an unintentional indication of the problems that come with changing the requirements.

### 4.5.2 Drink Vending Machine

This is a greatly simplified and partially revised version of the ”Vending Machine Control Software Problem” (created by Masamitsu Noro), which was used as a common problem in the modeling workshop at the Object-Oriented Symposium ’97.

1. The vending machine accepts bills and coins. Only 1,000 yen notes are accepted, and coins of 10 yen or more are accepted.
2. The current balance, which is the amount inserted minus the amount already sold, is shown on the display.
3. For items that can be purchased with the current balance and are not sold out, the sales lamp corresponding to that item will light up.
4. For sold-out items, the sold-out lamp corresponding to that item will light up. If all items are sold out, money cannot be inserted.
5. When the purchase button for an item whose sales lamp is lit is pressed, the sales lamp for that item will flash, and one item will be dispensed into the dispenser. At the same time, the balance will be deducted by the price of the item. During this time, money cannot be inserted.
6. When the refund button is pressed, the remaining amount at that time will be refunded to the change dispenser. Coins cannot be inserted while the refund operation is in progress.

This problem corresponds to the ”drink vending machine scenario” listed in the 3.3.3 section. The difference from the scenario description is that it includes a description of constraints such as ”If all products are sold out, you cannot insert money.” and that it also covers exceptional cases.

# Chapter 5

# Data and Control Flow Model

From this chapter onwards, we will take up the most representative modeling techniques in order. To put it a little dramatically, modeling is about how to perceive the world. This is not a metaphysical concept, and the ”world” is also a very limited and cut-out area. However, in the following explanation of modeling techniques, we will deliberately use the phrase ”how to perceive the world” to show the characteristics of each model. The central concept we will deal with in this chapter is ”flow.” We will focus on ”data” and ”control” as things that flow.

## 5.1 Data Flow Model

In the data flow model, data is seen as flowing in the world. Flows branch and merge. Data changes shape during the flow. The thing that transforms its shape is called a process.

### 5.1.1 Data Flow Diagram

Data flow diagram (abbreviated as DFD) is a diagram that describes data flow and its transformations. In terms of the classification of graph modeling,

| Data flow diagram | |
|---|---|
| **Mode:** | Dynamic |
| **Object:** | Flow of things |
| **Vertex:** | Process |
| **Edge:** | Data flow |
| **Subgraph:** | Subprocess |

Data flow is represented by directed edges as in the figure 5.1. Data that flows is indicated by writing its name next to the edge. Data flows from a ”sender” to a ”receiver,” and these senders and receivers are called processes. A process is represented by writing the process name inside a circle.

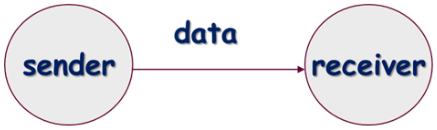


Figure 5.1: Data flow

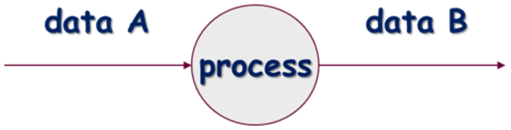


Figure 5.2: Process

Figure 5.2 shows a data flow diagram that focuses on processes. The process converts data A into data B. In other words, data A is the input to the process, and data B is the output from the process.

As shown in figure 5.3, a process may generally have multiple inputs and outputs.

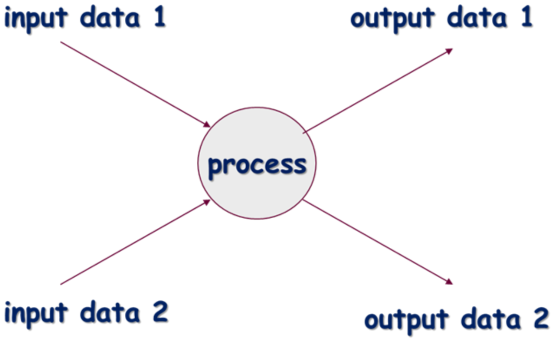


Figure 5.3: Multiple Inputs and Outputs to a Process

Note that the relationship between multiple inputs and outputs is not specified by this data flow diagram. For example, it is not known whether output 1 is made from input 1, input 2, or both. The same is true for output 2. Furthermore, it is not specified whether the outputs are mutually exclusive, such that when output 1 is made, output 2 is not made, and when output 2 is made, output 1 is not made.

Not only does a data flow diagram not specify the logical relationship between inputs and outputs, it also does not specify the timing at which data enters or leaves the process.

Input and output may always flow continuously, or data may enter and exit in a lump at a certain point. Two or more input data may enter simultaneously in a synchronized manner, or may enter randomly. Or there may be a rule that input 1 always precedes input 2. There is no rule about the order or timing of multiple outputs.

The contents of the data flow are data that have contents that are the subject of operations by processes, and do not include events that control the timing of process startup control, interrupts, or synchronous signals. Since this is inconvenient for modeling reactive systems or real-time systems, the Hatley method and Ward method, which extend DFD and introduce notations that distinguish and describe control data and control processes, have been proposed, but these modeling methods are rarely used at present, and have been replaced by state transition model methods (e.g. Statechart).

Although this property of DFD may seem inconvenient, this is exactly what abstraction by a model is. If input/output relationships and timing are important, this model should not be used. However, by ignoring these, the data flow model is simple and powerful when you focus on the structure of what data flows between processes and how it is transformed there.

### 5.1.2 Hierarchy

In the data flow model, there is a recursive relationship in which decomposing a process also results in a data flow diagram, which allows for a natural hierarchy. This relationship is a typical example of a hierarchical structure of a graph using subgraphs.

For example, consider the process of issuing an invoice, as shown in Figure 5.4. This process can be broken down

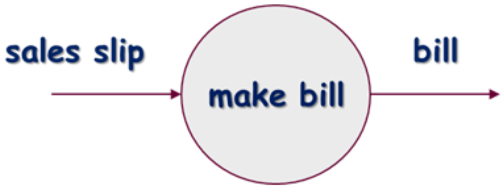


Figure 5.4: Invoice issuing process

into a data flow diagram like the one in Figure 5.5. Here, the input from outside is the sales slip, and the output to the

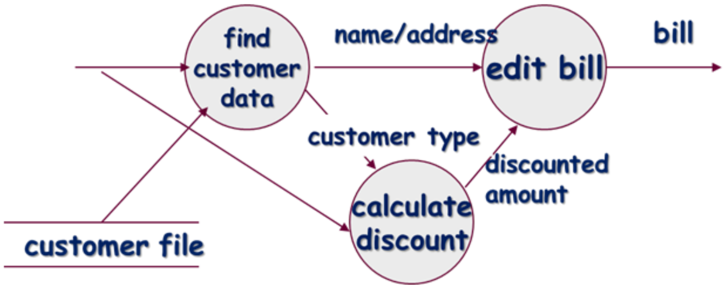


Figure 5.5: Detailed invoice issuing process

outside is the invoice. The interface is the same as in Figure 5.4. However, the internal structure has been broken down into three processes and one file. The file is where the data is stored. Even in a data flow model, data does not just flow, but sometimes it is necessary to have a place for it to accumulate. However, from the perspective of the nature of a data flow model, files should be kept to a minimum.

Furthermore, here, the data that was a ”sales slip” at the higher level is broken down into its internal components, ”customer ID” and ”sales amount.” The structure of this type of data is managed in a data dictionary, which will be described later.

### 5.1.3 Structured Analysis by DeMarco

The technique that uses data flow diagrams (DFDs) by Tom DeMarco is called Structured Analysis (SA). This is a representative example of an analysis technique that focuses on the structure of data flows.

DeMarco’s book [40] is a basic textbook and a good book.

SA is a method that predates object orientation, and since DFDs have not been adopted in UML, some consider it not as important as it used to be. However, it has the following characteristics, and is still effective if used appropriately.

1. It is intuitively easy to understand without the need for complicated theories.

2. Functional decomposition can be naturally written as a hierarchical structure.

3. The diagram helps to follow the flow of data, and functions can be understood by the combination of inputs and outputs.

4. Most commercially available CASE (Computer Aided Software Engineering) tools support DeMarco-style DFD.

In addition to DeMarco's method, there is SADT by T. Ross, which was mentioned in the 3.3.2 section on the history of requirements engineering, as a structural specification technique based on data flow. There is also the Gane-Sarson method for notating DFDs, but the essence is the same.

SA uses the following tools.

**Data Flow Diagram (DFD)** The diagram is drawn using a combination of the following four basic components:

- Process: A round frame. Also called a bubble.
- Data flow: An arrow. Connects a process to another process, entity, or file.
- Entity: An entity outside the target system.
- File: A place to store data. Also called a data store.

Among these, the process and data flow described in the previous section are particularly basic.

**Data Dictionary** This defines and records the content and composition of data. Its structure is defined by a combination of sequence, selection, and iteration. It has the same descriptive power as a regular expression, and can also be drawn using Jackson structure diagrams (section 6.5.4).

**[Example]** Here, the operators that make up sequence, selection, and iteration are represented as $+, |, and*$, respectively.

$$\begin{aligned}\text{Sales slip} &= \text{Book number} + \text{Customer number} + \text{Number of copies} + \text{Amount}\\ \text{Customer number} &= [\text{A} \mid \text{B} \mid \text{C}] + \text{Number}^*\end{aligned}$$

**Process specification description** The definition of the lowest level process in a hierarchical structure must be described in a way other than DFD. To do this, structured English, decision tables, decision trees, state transition diagrams, formal specification description languages, etc. are used.

### 5.1.4 How to write a data flow diagram

Based on the method of structured analysis, we will explain how to write a DFD using the common example problem of the sake shop problem.

First, draw a context diagram. A context diagram shows the relationship between entities and the entire system. In principle, one or more entities and one process (i.e., the entire system in question is viewed as a process) will appear there.

What appears as the subject in the problem statement is a candidate for the process or entity that will be the sender of the data flow. What appears as a predicate is likely to refer to the data transfer action between processes, and the dative of that action (to) is a candidate for the process or entity that will be the receiver of the data flow. Furthermore, what corresponds to the accusative of that action is a candidate for the data that will flow between processes.

- ”The warehouse clerk receives the container”
  ⇒ The warehouse clerk is a candidate for the receiving process. The container is a candidate for the data.

- ”(The warehouse clerk) hands the shipping slip to the receptionist”
  ⇒ The warehouse clerk is a candidate for the sending process. The receptionist is a candidate for the receiving process. The shipping slip is a candidate for the data.

- ”(The warehouse clerk) receives a shipping instruction from the receptionist”
  ⇒ The receptionist is a candidate for the sending process. The warehouse clerk is a candidate for the receiving process. The shipping instruction is a candidate for the data.

- ”(The warehouse clerk) ships the internal goods”
  ⇒ The warehouse clerk is a candidate for the sending process. The internal goods are a candidate for the data.

- ”The receptionist receives several dozen shipping requests every day (from the requesters)”
  ⇒ The requester is a candidate for the sending process. The receptionist is a candidate for the receiving process. The shipping request is a candidate for the data.
- ”(The receptionist) issues a shipping instruction to the warehouse clerk.”
  ⇒ Already mentioned
- ”(The receptionist) calls the requester if there is no stock or the quantity is insufficient.”
  ⇒ The receptionist is a candidate for the sending process. The requester is a candidate for the receiving process. The phone call is a candidate for the data.
- ”(The receptionist) fills in the inventory shortage list.”
  ⇒ The receptionist is a candidate for the sending process. The inventory shortage list is a candidate for the data.
- ”(The receptionist) informs the warehouse clerk of the containers that are about to become empty.”
  ⇒ The receptionist is a candidate for the sending process. The warehouse clerk is a candidate for the receiving process. The containers that are about to become empty are candidates for the data.

If we were to illustrate this straightforwardly, it would look like Figure 5.6.

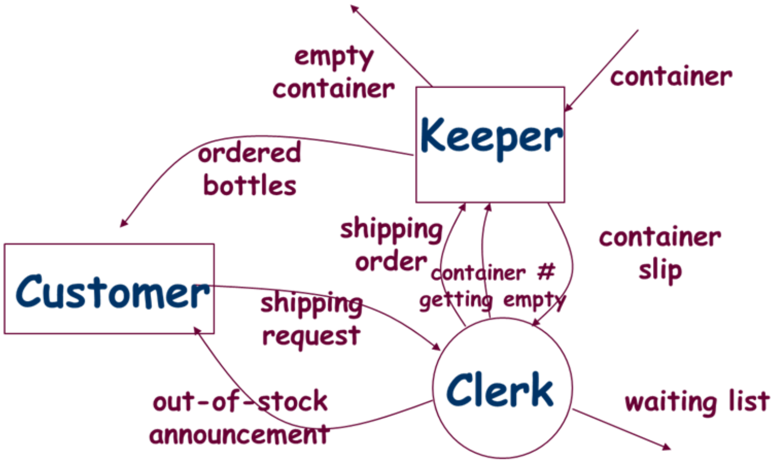


Figure 5.6: Context diagram of the sake store inventory problem

In the diagram, the boxes represent entities, and the circles represent processes that represent the scope of the system. In a context diagram, entities are positioned as components of the system’s environment, so it is usually not necessary to show information exchanges between entities. However, to understand how the environment works, it is useful to show interactions between entities to the extent necessary. Therefore, the diagram also shows containers entering and leaving the warehouse clerk, and interactions between the warehouse clerk and the client. Alternatively, the warehouse clerk can be considered to be inside the system rather than an external entity, and can be circled. In the DeMarco method, it is stipulated that only one process should be written in the context diagram, but if multiple processes within the system boundary are visible from the beginning, there is no need to stick to that rule. In addition, in terms of data flow, the flow of objects should be excluded, but in some cases it is easier to understand the structure of the problem if the flow of necessary items is also described. This is the intention behind adding the flow of containers and orders to the diagram.

In this way, if the problem statement is somewhat organized, it is quite natural to create a context diagram from it. The resulting diagram is useful for clearly showing the relationship between the target system and its surrounding environment. For this reason, context diagrams are not limited to structured analysis methods, but are used in the first stage of many development methods.

Next, the top-level diagram shows the roughest decomposition of the system. The processes that appear there are numbered sequentially, starting from 1.

In the sake shop problem, the diagram that breaks down the receptionist's work 5.7 is the top-level diagram.

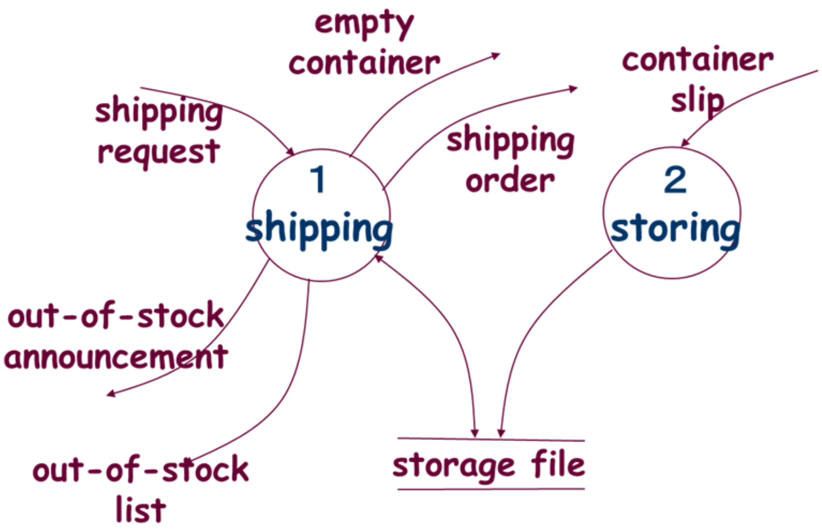


Figure 5.7: Top-level diagram of the liquor store inventory problem —Part 1—

However, this diagram corresponds to the older version of the problem. If, in the case of a shortage of stock, the order is to be shipped out when the order can be fulfilled by a later arrival of stock, the top-level diagram would change to, for example, Figure 5.8. In other words, instead of outputting the list of stock shortages to the outside, it would be made an internal file (which would require a trivial change to the context diagram accordingly), and a process would be added that would refer to it when the stock is received and perform the appropriate processing.

Next, create a decomposition diagram of process i. Its data flow with the outside world must be perfectly consistent with the data flow with which the corresponding process shown at the higher level exchanges data with the surrounding processes. Processes that appear in the decomposition diagram of process $i$ are numbered in decimal format, like $i.i_k$, which is the same as the chapter numbering system.

Figure 5.9 shows the decomposition of process 1 (shipping) at the top level of the sake store problem. This corresponds to problem 1 in Figure 5.7.

Here, pay attention to the expressions ”enough-stock case shipping request” and ”out-of-stock case shipping request”. The data content is the same ”shipping request”, but this is devised to clarify its nature by adding modifiers. This implies that the data flow after ”outgoing receipt” is a branch with two options. Of course, ambiguity remains with only such devised modifiers. To describe it precisely, we use a data dictionary. In the ”enough-stock shipping request” item in the data dictionary, we write the definition that it is a shipping request data that satisfies the condition ”enough stock” (this condition can be rigorously described with a logical formula).

If we further decompose the processes in 1.1 and 1.3, we will find that we must also decompose the structure of the file called the ”waiting order file.” Therefore, we will also decompose the file structure according to the process decomposition in 1.1 and 1.3, and register it in the data dictionary.

This structural decomposition is repeated until the process represents a single task. For processes that we have decided not to decompose further, we will create a process specification description. There is a degree of freedom in what is considered a single task. It also depends on what is used in the process specification description. For example, if a state transition diagram is used in the process specification description, the criterion is to stop when the decomposition can be made to fit on one state transition diagram.

The extent to which we decompose also depends on the purpose for which the data flow model is used. In structured analysis, the level of process specification was assumed to be the program procedure (subroutine) level, but when used as a requirements model, the decomposition will stop at a higher level of abstraction.

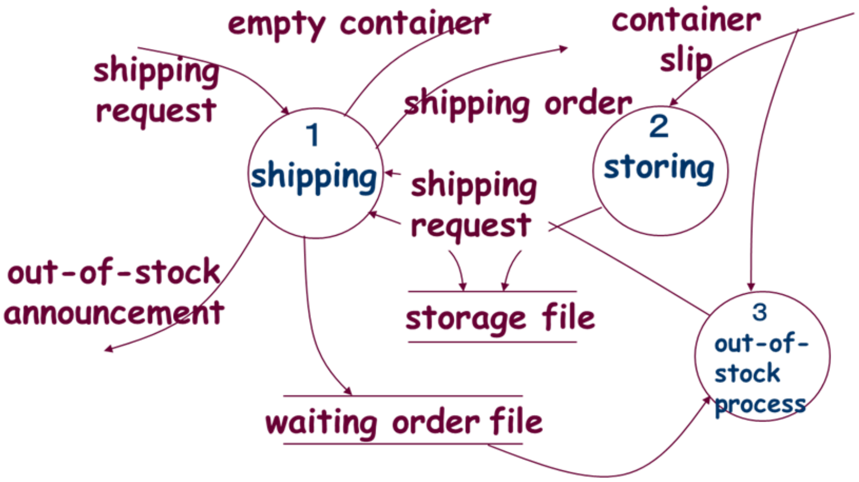


Figure 5.8: Top-level diagram of the sake store inventory problem —Part 2—

### 5.1.5 Model Verification

There are two main ways to check whether a data flow model expresses what is intended and whether the process decomposition is correct. This is related to the properties of graphs described in the 4.3.3 section.

One method is to trace the data flow path and check whether there are any problems with the series of data transformations that appear along it. For example, in Figure 5.9, there is a path along which an out-of-stock request entered from outside flows in the case of ”enough stock” and a path along which it flows in the case of ”out of stock.” This method verifies whether the series of data transformations for each are valid.

The other method focuses on the process and analyzes whether a set of output data can be correctly created from a set of input data. There is a possibility that insufficient or unnecessary input data will be found, or that output data will be missing or that should not be output here will be found.

These verifications are usually performed informally by humans, but they are still effective in improving the quality of the model.

In addition, the following points should be noted.

- The data flow only represents the relationship between processes and data, i.e., data entering or leaving a process, and does not represent the order or timing of processing. Therefore, it is essentially different from a flowchart that represents the flow of control.
- In terms of the order of thinking about modeling, DeMarco says that it is better to think about the data flow rather than the process first, and then think about the process as something that transforms it.
- The relationships between data in a process, such as which input is used to create which output, whether multiple inputs or multiple outputs are in a coexistence (and) relationship, or whether they are in a selection (or) relationship, are not shown directly. These can be seen by looking at the structure further down (ultimately the process specification).
- At one hierarchical level, the complexity should be not be high so that it is easy to understand (for example, 3-7 processes) and the conceptual consistency should be maintained.
- Verification can be done by checking the balance of data flow between layers.
- Data and processes must be given names. These names must be specific and meaningful. Data is to be named with a noun and process with a verb. Names like ”something data” or ”something processing” are not good choices.

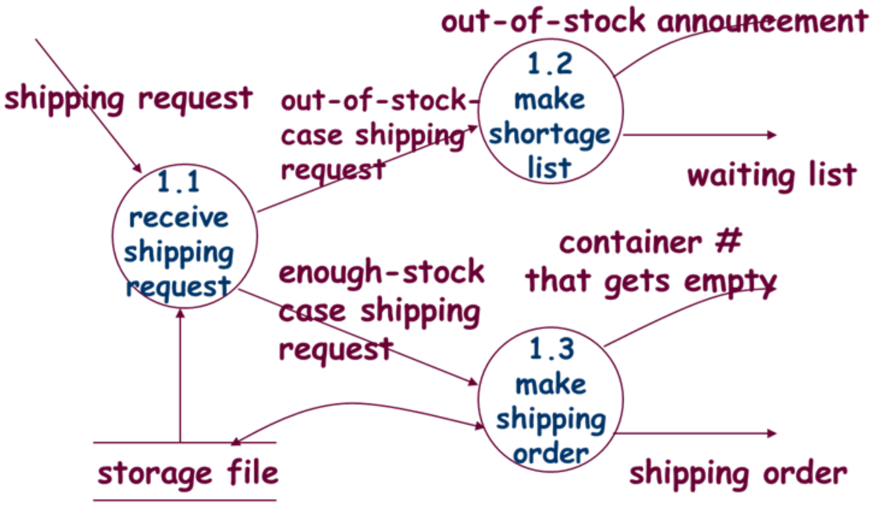

out-of-stock announcement
shipping request
out-of-stock-case shipping request
1.2 make shortage list
waiting list
1.1 receive shipping request
container # that gets empty
enough-stock case shipping request
1.3 make shipping order
storage file
shipping order

Figure 5.9: Second-level diagram of the sake store inventory problem

- Files should only contain information that is exchanged between the processes at that level. Do not include files that are only used at lower levels.
- Processes or files that allow data to flow only in or only out do not normally exist.

**Practice Problem**

Model the following problem using a data flow model.

**[Bookstore Accounts Receivable Management]** A bookstore sells books on credit to certain customers.

The salesperson hands the book along with a delivery slip to the customer and immediately sends a copy (sales slip) to the accounting clerk.

At the end of the month, the accounting clerk tallys up the sales slips for that month, creates an accounts receivable balance report, and also sends an invoice to the customer.

Customers pay the invoice amount by cash or bank transfer, but do not necessarily pay the full amount at once. The amount received is forwarded to the accounting clerk as a deposit slip from the finance clerk at any time.

- Sales slip: Customer name; Sales amount
- Deposit slip: Customer name; Deposit amount
- Accounts receivable balance report: Total outstanding balance at the end of the previous month; Total sales amount this month; Total deposit amount this month; Total outstanding balance at the end of the current month
- Invoice: Customer name; Invoice amount for the previous month; Purchase amount this month; Payment amount this month; Invoice amount for the current month

Analyze and systematize the work of the accounting clerk.
(From ”Program Design” by Toshiharu Yamasaki, [151])

## 5.2 Control flow model

The control flow model sees the world as a place where semantically coherent tasks (processes) are executed sequentially. The overall procedure of the tasks is controlled, and the control structure is determined, such as the order of which task to perform after a certain task, the selection of tasks depending on the case, the iteration of certain procedures, and the parallel execution of several tasks. The tasks here are roughly equivalent to the processes in the data flow model. However, in this model, the focus is on the order in which the processes are executed, and not on what data is passed between processes.

### 5.2.1 Flowchart

The basis of a von Neumann computer is to retrieve an instruction specified by the program counter from memory and execute it. If it is a calculation instruction, the program counter points to the next address in memory after it is executed. If the instruction is a (conditional) jump instruction, the program counter is changed to the specified address (depending on the condition) and control jumps to that address. Flowcharts have been used since the dawn of computers to illustrate the operation of programs created based on this principle. Figure 5.10 is an early flowchart from “Planning and coding of problems for an electronic computing instrument” written in 1947 by Herman Heine Goldstine and John von Neumann[53].

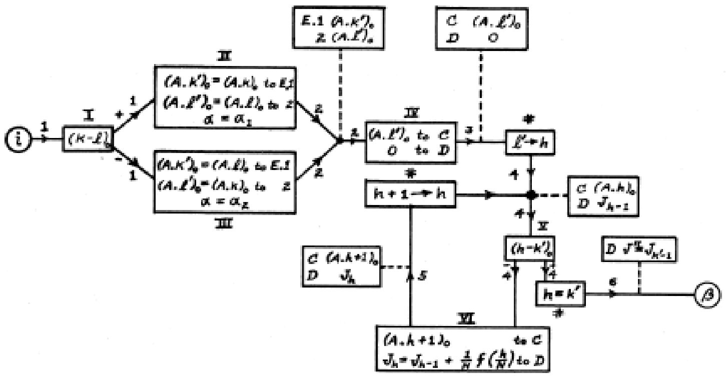


Figure 5.10: Flowchart by Goldstine & von Neumann from [53]

Following that, flowcharts were standardized to a notation that combines boxes representing ”process,” diamonds representing ”decisions,” boxes with rounded corners representing terminals, and arrows representing the flow of control, as shown in Figure 5.11.

Figure 5.12 shows a concrete example of flowchart illustrating an algorithm of finding the greatest common diviser of the given two integers.

The following are the characteristics of a graphical model of a flowchart.

| Flowchart | |
|---|---|
| **Mode:** | Dynamic |
| **Object:** | Viewpoint movement |
| **Vertex:** | Processing, decision |
| **Edge:** | Control flow |
| **Subgraph:** | Subroutine |

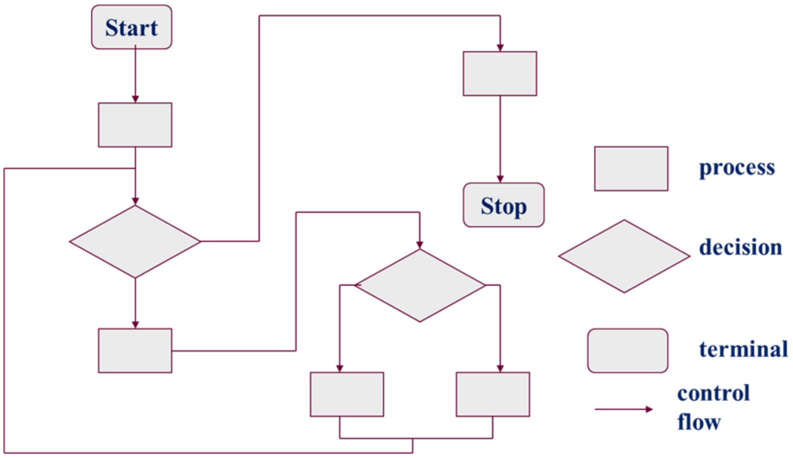


Figure 5.11: Basic constructs of flowchart

In a flowchart, vertices correspond to processes (rectangles) or decisions (diamonds), and edges correspond to control flow. In addition, terminals (rounded boxes) that represent the start and end points of processing as special vertices, and various icons are used to represent processing devices and media such as disks, which are not shown in this diagram, but they are not essential.

A flowchart is a program that describes how control of execution is transferred within computers (or other information processing machines, including humans). Execution is sequential, and therefore the vertex at which control of execution is currently located is uniquely determined. The location of the specific processing is within the ”process” depicted by a box, and the edges only indicate where execution will move next. After a ”decision,” the flow of control branches, and the vertex into which two or more edges flow corresponds to the junction of branches or the start of repeated execution.

The flow of control represented by the edges of a flowchart is a primitive equivalent of goto at the programming level. In the era of structured programming in the 1970s, programs with complicated control flows due to the use of goto were considered bad, and the corresponding flowcharts were rejected. Instead, many proposals were made to describe control flows with branching and loop structures, such as PAD, HCP, and SPD. Tools were also created to support these descriptions, and a function to generate programs from diagrams was added and used.

### 5.2.2 Activity diagrams

UML activity diagrams represent the flow of control and are similar to flowcharts.

| Activity diagrams | |
|---|---|
| **Mode:** | Dynamic |
| **Object:** | Viewpoint movement |
| **Vertex:** | Processing, decision |
| **Edge:** | Control flow |
| **Subgraph:** | Subprocedures |

However, they have the following characteristics that are different from flowcharts:

1. Unlike flowcharts, they are not intended to be used only at the programming level, but rather as conceptual-level models.

2. It allows the description of parallel activities.

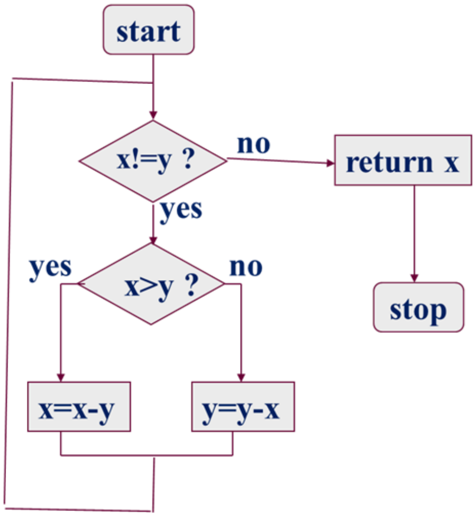


Figure 5.12: Example of a flowchart

The main components are shown in Figure 5.13.

Activity diagrams, like flowcharts, basically show procedures. Interestingly, the control flow of an activity diagram, like a flowchart, consists only of primitive control transfers equivalent to goto, and no branching or loop structures are provided. Figure 5.14 shows the common problem of the drink vending machine drawn as an activity diagram.

Compared to the scenario in Section 3.3.2, this activity diagram includes cases where the user stops midway, adds money, continues after removing the money, and other exceptional cases, and describes a more general situation. On the other hand, the control flow is more complicated and somewhat difficult to understand.

In the case of a flowchart, the subject of the activity is determined to be a single computer (or equivalent), so there is no problem in determining who the subject of the activity is. However, when using an activity diagram in an object-oriented framework, various objects can be the subject of the activity. In order to clarify the subject of the activity, a notation has been established that divides the area into sections like the lanes of a swimming pool. For example, it is drawn like Figure 5.15.

Figure 5.16 shows another common problem, the sake store problem, drawn in this way by dividing the courses according to the main activities. The activities in the sake store problem can be broadly divided into activities when containers are stored and activities when sake is shipped out according to customer orders. Figure 5.16 shows activity diagrams for each case. Comparing it with the data flow diagram in Figure 5.9, you can see the characteristics of both cases.

When viewed as a graph, the paths in an activity diagram are interpreted as paths of execution. Branches are also interpreted as ORs, and parallel branches as ANDs. Subgraphs can also be treated as activity diagrams, which allows the expression of hierarchical structures similar to data flow diagrams. At the program level, activities at lower levels correspond to procedures and subroutines.

### 5.2.3 How to Use Activity Diagrams

Activity diagrams have many uses. First, they are suitable for capturing the activities of a system as a whole. In terms of modeling activities, they have a complementary relationship with the behavior-related diagrams of UML, such as state machine diagrams, sequence diagrams, and communication diagrams. In addition, they are often useful for describing processes and procedures, such as workflows, business processes, and development processes, that are not necessarily directly related to computation.

Activity diagrams in UML actually have components that allow data flows to be overlaid. In a sense, it is natural to try to extend the model notation to deal with objects that cannot be expressed well in a model. However, this inevitably

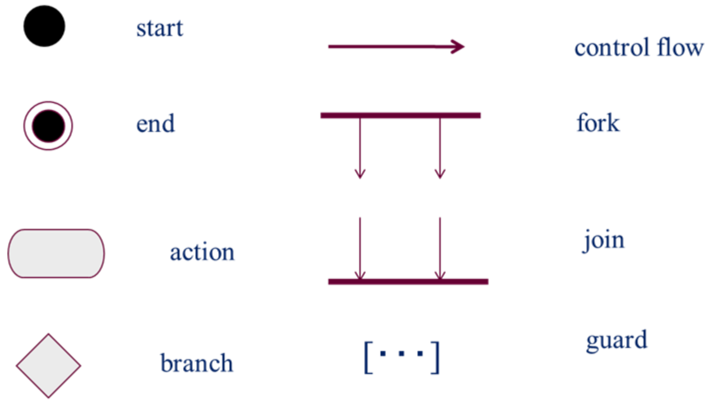


Figure 5.13: Components of an activity diagram

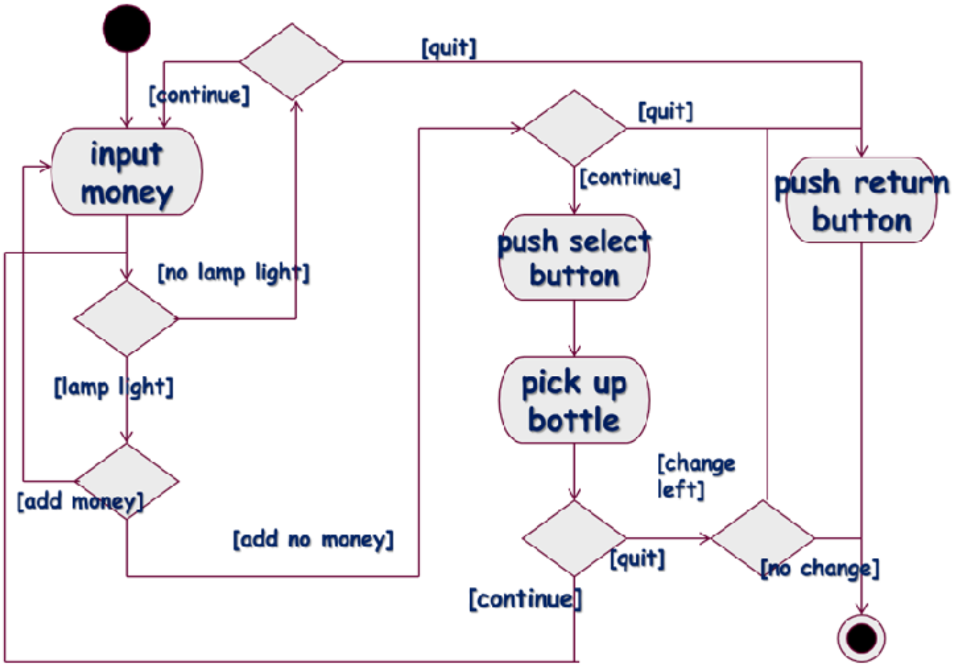


Figure 5.14: Activity diagram for a vending machine

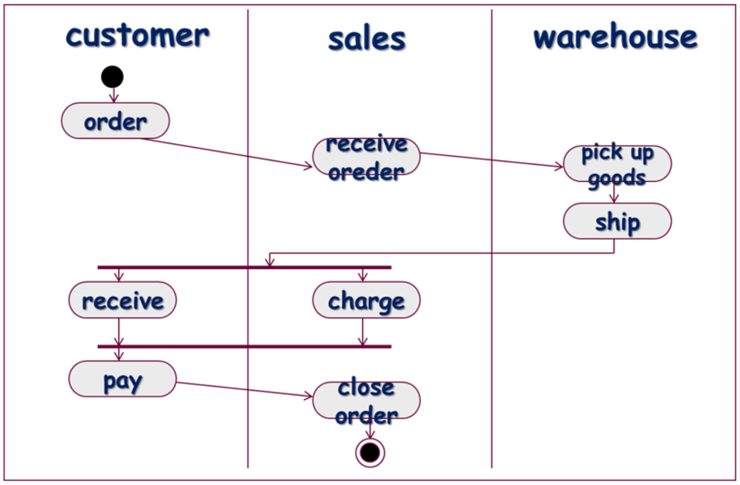


Figure 5.15: Swimlane: Swimming pool course

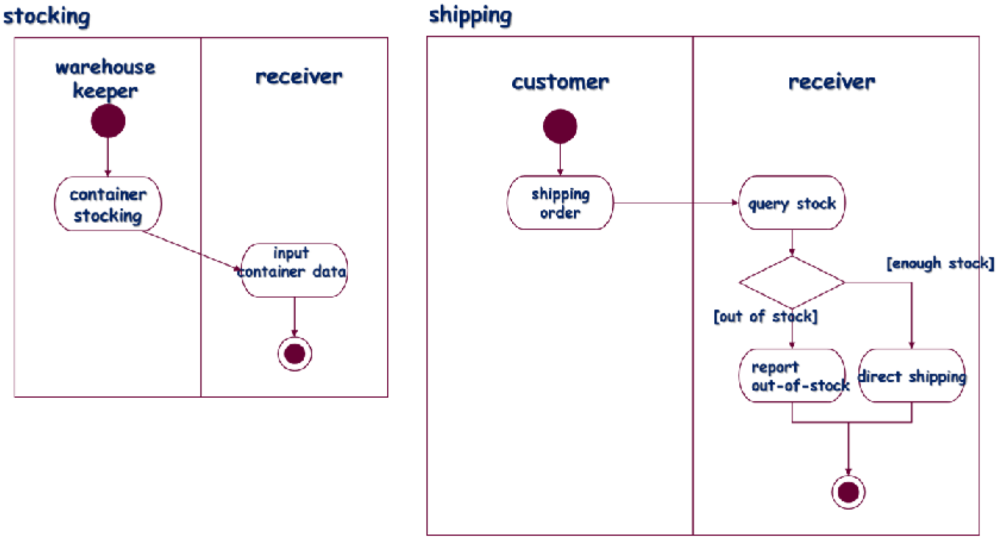


Figure 5.16: Activity diagram for the sake shop problem

distorts the essence of the abstraction that the model originally aimed for. In addition, model expressions with complex notations are difficult to use, both from the perspective of the person writing them and the reader. Therefore, this book takes the approach of focusing on model expressions that are as concise as possible and capture the essence.

Comparing activity diagrams and data flow diagrams, we can see that data flow diagrams are closer to static models, even though they deal with the dynamic nature of flow. In other words, data flow diagrams ignore anything related to control. On the other hand, flowcharts ignore anything related to data, but activity diagrams incorporate this and are able to express data flow as well. As a result, their expressiveness has increased, but their simplicity as a model has decreased.

# Chapter 6

# Dynamic Behavior Model

In the previous chapter, we dealt with ”flow.” The central concept of this chapter is ”behavior.” Behavior, by its very nature, changes dynamically. When modeling dynamic behavior, the subject that causes the behavior is assumed first. This is generally called a process or object, but in this chapter we will use the word ”object.” However, the word ”process” may also be used depending on the context, but please understand that it can be considered the same as the subject of the behavior. In addition, the important concept is ”event,” which is what triggers the behavior. An event is an input that triggers an object to cause a certain behavior, but at the same time, the behavior performed by the object causes an event to be sent as output to other objects or the environment.

## 6.1 Characteristics of Target Processes

The target processes performed by the system to be modeled have various characteristics. Before looking at dynamic behavior models, let’s consider what characteristics this model is suitable for describing and analyzing processes. Below, we characterize the characteristics of processes along three axes.

### 6.1.1 Transformational and Reactive

A transformational process is a process that converts given inputs one after another and outputs the final transformation result. In other words, the process has a beginning and an end, and the user is not aware of what happens during the process. In contrast, a reactive process is a process in which the process and the user respond to each other as the process proceeds. A process may have a beginning and an end, but what is important is the interactive exchange that takes place during the process. Bank automated teller machines and train ticket vending machines are typical examples of reactive systems that we see in our daily lives. A reactive system usually does not stop, and stopping is often synonymous with a breakdown. This is a striking difference from a transformational system, which only produces the resultant output when it stops.

In the early days of computers, transformational systems were the norm, but reactive systems also appeared relatively early on. However, with the development of the Internet, transformational systems have become more mainstream in modern times.

### 6.1.2 Sequential and Concurrentl

A sequential system is one in which processing is performed sequentially over time. On the other hand, a concurrent system is one in which multiple processes are performed simultaneously. There are two main reasons why concurrency is required. One is a natural requirement that arises from the concurrent nature of the problem domain being addressed. For example, in a system that manages automated teller machines, the machines are installed in many different locations, and concurrency arises because services are requested from them at the same time. The other is when multiple processes are run simultaneously to improve processing efficiency, and processing is parallelized within the computer. Modern operating systems usually support parallel processes in this sense. Even if there is only one internal processor (CPU), a technique called time sharing can be used to achieve apparent parallelism. This improves the overall efficiency of multiple processes.

Nowadays, it is becoming common for needless say about supercomputers but as well as personal computers, to have multiple CPUs to distribute the load and increase efficiency. The word ”parallel” is used to describe such measures to duplicate or triple systems in system design. Parallelization is not limited to CPUs, but also to peripheral devices such as disks and communication devices.

### 6.1.3 Batch Type and Online Type

In mainframe computer applications, a distinction between batch type and online type has long been made. Batch means that the necessary processing is accumulated and then processed all at once later, while online means that the system is always running and processing is done on the spot following requests from users. Therefore, it roughly corresponds to the distinction between transformation type and reactive type, but the distinction is more about how users see the system from the outside than the processing method.

In mainfreme computers, batch type processing was developed first, but online systems have also been developed for a long time. In Japan, the green window system of the Japanese National Railways (now JR) and online systems in banks are pioneers.

The control flow model and data flow model discussed in Chapter 5 were originally used in methods developed with transformation type, sequential type, and batch type systems in mind. They can also be applied to the development of reactive, concurrent, and online systems, and various proposals have been made to add reactive, concurrent, and online elements to these model notations. However, the models we are going to discuss in this chapter were originally developed for reactive, concurrent, and online processing systems.

### 6.1.4 Theory of Concurrent Systems

If the characteristics of the processing processes discussed in this chapter are represented by ”concurrency,” then various theoretical foundations have already been established by predecessors. The most influential are CSP (concurrent sequential processes) [60] by Tony Hoare and CCS (calculus of communicating systems) [91] by Robin Milner. These, along with similar methods, were later collectively referred to as ”process algebra.” Some of the models introduced below were also influenced by CSP and CCS.

## 6.2 Sequence Diagrams

UML sequence diagrams are used to describe the time course of events that take place between multiple objects, when messages are exchanged between them and a meaningful action is performed. Therefore, here, ”events” are interpreted as messages being sent and received.

A sequence is a series of messages that proceed along a time axis. The series of actions represented by this is sometimes called a scenario. A scenario can also be regarded as describing the collaborative actions of the objects that participate in it. Therefore, if requirements are described on a scenario basis (see 3.3.3 section), it is natural to describe the behavior of the system as a sequence diagram as an extension of that. Each of the participating objects plays a role in the collaborative actions.

### 6.2.1 Graphic Characteristics of Sequence Diagrams

Figure 6.1 shows one scenario assumed in the vending machine problem as a sequence diagram.

To draw this diagram, we must first identify the set of objects that will act as actors. This is a crucial step when constructing a model of dynamic behavior, not just for sequence diagrams. If we have a scenario description first, it will be fairly natural to identify the actors that appear in the scenario, determine the candidates for objects, and then select and discard them, and add, disassemble, or integrate objects as necessary to determine the set of objects. Of course, drawing a sequence diagram often makes the necessary objects clear again. In other words, it can be said that the identification of objects and the description of the sequence diagram proceed in parallel.

The objects assumed in the request scenario for the vending machine and the activity diagram in Figure 5.14 are the customer and the vending machine.

Of the actions that appear there,

- Putting money in
- Displaying the amount
- Reducing the displayed balance after a sale

are all related to the handling of money, so we can think of an object that is responsible for that. Let's call it a "cashier." Also,

- Lighting of the sales lamp
- Sales according to the item button

are matters related to product sales management, so you can think of the object responsible for that. Let's call it "sales". Furthermore, the bottle discharge action can be included in "sales", but since a physical device that arranges and discharges the bottles is assumed, we will make it a separate object called a "rack".

In this way, the objects are determined, and the interactions between them are shown in a sequence diagram.

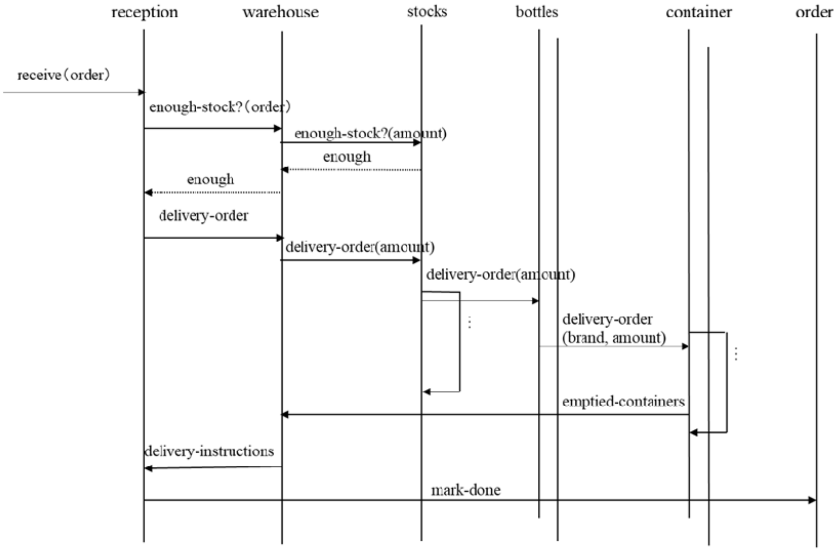


Figure 6.1: Vending machine sequence diagram

As a graph, a sequence diagram has the following characteristics.

| Series diagram | |
|---|---|
| **Mode:** | Dynamic |
| **Object:** | Information flow |
| **Vertex:** | (implicitly) the time when an object sends or receives. |
| **Edge:** | Message sending/method invocation |
| **Subgraph:** | Not applicable |

Each object is also assigned a vertical time axis. This format is similar to the activity diagram, which shows the division of the activity subject's responsibility in the course of a pool.

In this way, a sequence diagram has a structure that is slightly different from a simple graph. Also, there is no recursive structure derived from subgraphs. In addition, UML sequence diagrams have some extensions, such as allowing the condition for sending a message to be written in the notation [condition] and the description of repetition.

### 6.2.2 Representation of Concurrent Processes

Concurrent processes can be described in sequence diagrams. Figure 6.2 shows an example.

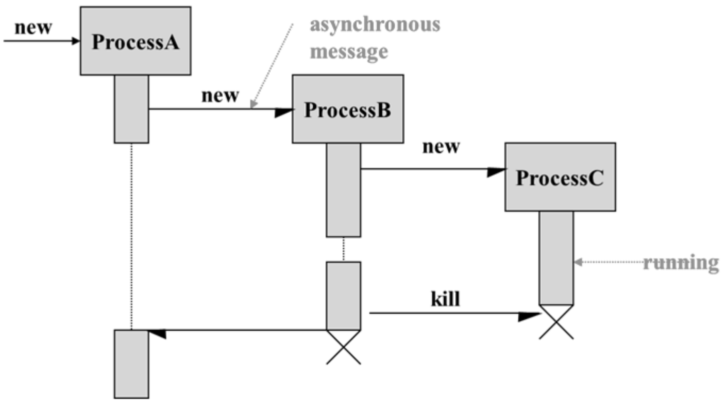


Figure 6.2: Sequence diagram with concurrent operations

Arrows with half tips indicate asynchronous message transmission. Asynchronous here means that the object that sent the message continues its own operation after sending the message without waiting for the completion of the operation of the receiving object that was started by the message. The wide band drawn from top to bottom along the time axis for a process corresponds to the period during which the process is running.

## 6.3 Communication Diagram

A communication diagram explicitly depicts the cooperative relationships between objects through the exchange of messages. Figure 6.3 shows the communication diagram of a vending machine corresponding to Figure 6.1.

The characteristics of the graph are as follows:

| Communication diagram | |
|---|---|
| **Mode:** | Dynamic |
| **Object:** | Information flow |
| **Vertex:** | Object |
| **Edge:** | Connection between objects |
| **Subgraph:** | Communication structure of sub-objects |

In a communication diagram, objects are assigned to vertices, and when messages are exchanged between objects, corresponding vertices are connected by edges. In UML, communication diagrams are semantically equivalent to sequence diagrams. Therefore, when messages are attached as labels to edges along with arrows in the sending direction, decimal numbers are attached to them to indicate the time sequence. However, communication diagrams are useful for illustrating the cooperative relationships between objects. Therefore, it seems unnecessary to forcefully make it equivalent to a sequnce diagram that corresponds to one scenario. This is because a communication diagram can depict the mutual communication relationships based on multiple scenarios at a time. As with sequenc diagrams, UML communication diagrams allow for extensions such as the description of message transmission conditions. If a communication

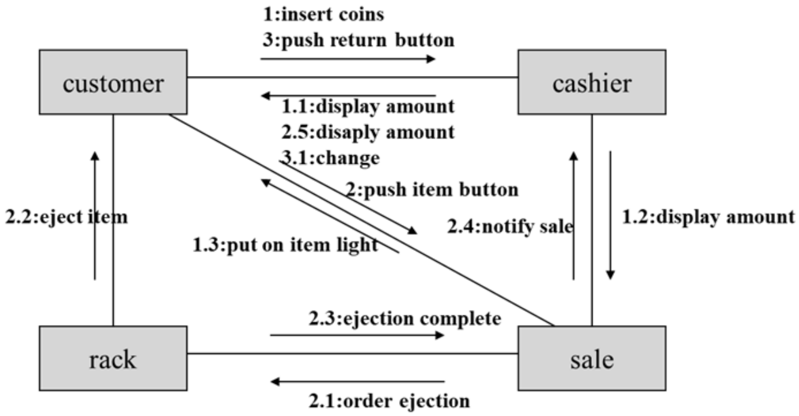


Figure 6.3: Vending machine communication diagram

diagram is used to represent the interactions between objects, the hierarchical structure of subgraphs can be assumed to be a communication structure created by the lower objects that make up a single object. There are many other diagrams with purposes similar to UML communication diagrams. For example, in David Harel's modeling technique for reactive systems using Statechart, a diagram that represents the interactions between processes equivalent to this communication diagram is called a structure diagram. In Jeff Kramer & Jeff Magee's modeling technique for concurrent systems, centered on the process description language FSP, which is an extension of CSP, also uses a similar diagram, calling it a structure diagram. As the name structural diagrams suggests, these diagrams aim to show structural relationships, i.e., interactions between objects or processes, rather than dynamic behavior.

## 6.4 State Machine Diagrams

A sequence diagram models the interactions between multiple objects, but it only gives a sample of how each object behaves based on one scenario. To describe all the behaviors of the object, many sequence diagrams must be prepared. In contrast, a state machine model can completely describe the behavior of an object with one model/diagram.

In a state machine model, an object is viewed as a machine. A machine has an internal state, and changes its state by exchanging events with the environment.

### 6.4.1 Basic Properties of the State Machine Model

**Graph Structure and Semantics**

The state machine model is a typical dynamic model. Its graph representation, the state machine diagram, can be characterized as follows:

| State machine diagram | |
|---|---|
| **Mode:** | Dynamic |
| **Object:** | Viewpoint movement |
| **Vertex:** | State |
| **Edge:** | Transition |
| **Subgraph:** | Lower state machine diagram |

The only components of the model that correspond to the topology of a graph are these ”states” and ”transitions”.

However, ”events” are an important concept that gives semantics to state machine models. Each edge, i.e., a transition, is associated with an event as a label, and it is interpreted that when that event occurs, the corresponding transition occurs.

The object described by the state machine model is the behavior of a machine with internal states. Events can be regarded as inputs given to the state machine from the outside. It is usually assumed that there is only one initial state in the set of states. It is also usually assumed that there is a final state as a non-empty subset of the states. However, when describing a reactive system using a state machine model, the final state does not exist or it refers to a singular state (deadlock). Under these assumptions, the state machine model is interpreted as describing a dynamic process as follows.

1. The process is initially in the initial state.
2. When a process is in a certain state and an event occurs, if there is a transition from that state that has that event as a label, the process moves to the destination state of that transition.
3. When the process transitions to the final state, the process ends there.

As you can see from this explanation, the state machine model assumes that the process is currently in some state. There is always one and only one current state. However, this is the case for sequential processes. In the case of concurrent processes, state machines are assumed to exist concurrently, and the ”current state” refers to the set of states of these multiple machines.

When a certain event occurs in a certain state, if there is always exactly one transition with that event as its label, the state machine model is deterministic, and if there are multiple transitions, the state machine model is nondeterministic. If there is no transition, the event is ignored, and it may be interpreted as no transition occurring or as some kind of error.

Transitions are assumed to occur instantly, and states are assumed to have a certain duration. Of course, the concepts of instantaneous and constant duration are relative, and an instant is any negligible time interval on the time axis considered in the model.

When viewed from the outside, a state machine behaves as if it had memory. In other words, the behavior of the machine is not determined solely by the current input, but depends on the history of past inputs, and appears to ”remember” them. This ”memory” is actually nothing other than a state, but this state cannot be seen from the outside. What the outside and the machine share is events. In other words, it is important to be aware of the distinction between states and events, which are the two components of a state machine model: states cannot be seen from the outside, while events are seen from the outside (or given from the outside).

### 6.4.2 Extension of the State Machine Model

The state machine model is often extended by adding further concepts and notations. In particular, we will look at four items: output events, transition conditions, concurrent systems, and state hierarchy. These were proposed in Harel’s Statechart, and are also followed in UML, which adopts Statechart as a ”state machine diagram.”

#### Output events

Up until now, events have been considered to be inputs for state machines. Isn’t it useless to call it a machine if it only has inputs and no outputs? However, in the simplest form of state transition model, a finite state automaton has no output. In this case, when an automaton transitions states according to a series of input events, if it reaches a final state, it is considered that the input sequence up to that point has been accepted, and if not, it is considered that it has not been accepted. This is interpreted as defining a language.

A sequential machine is a finite state automaton with an output added. The definition of a sequential machine will be described later, but even UML state machines based on Harel’s Statechart have been extended to allow output events to be described.

#### Transition Conditions

When you are in a certain state and an event occurs that causes a transition from that state, you may want to describe that the transition does not always occur, but only when a certain condition is met. In particular, in the case of concurrent systems, which will be described later, being able to write transition conditions often reduces the number of states.

### Concurrent Systems

Concurrent systems can be expressed as the coexistence of multiple state machines. As multiple state machines coexist, there is not just a singlle ”current state”, but as many as the number of machines. If each machine operates completely independently, it would not be interesting as a cocncurrent system, but machines interact with each other by sharing events. Sharing events means that the occurrence of one input event can cause multiple transitions with that label to occur simultaneously, or that the output event of one process can become the input event of another process.

### State Hierarchy

The biggest problem when using state machine models for model analysis for system development is that the number of states almost always explodes for problems of practical scale. There are several ways to deal with this explosion in the number of states, but one effective one is to introduce a composite state that contains several states. If we call the contained states child states, then when a process is in any child state contained in a composite state, and while transitions are occurring only between those child states, it is considered to be in one of the composite states of the containing parent. A transition from the outside of a composite state to the inside of it, or from the inside of a composite state to the outside of it, is a transition at that hierarchical level. This is an example of a common method of creating a hierarchical structure with a subgraph as a single vertex.

A state transition model that includes such an extension is called a state machine in UML, and is drawn as a state machine diagram. Its characteristics can be summarized as follows:

1. A transition is associated with an event and, in some cases, a transition condition (guard) or an output action.
2. A composite state that combines multiple states can be introduced. A composite state is considered to be an abstracted state that hides its internal structure. Conversely, a single state can be represented as a partial state machine that consists of multiple states.
3. An initial state can be specified as an implicit value for a composite state.
4. By using the ”history” of past transitions, for example, it is possible to specify that the state that was in when the most recent composite state was left should be entered.
5. It is possible to decribe and-independent states operating concurrently.
6. It is possible to describe fork and join transitions.

Below, we will borrow the example of an ”alarm watch” that Harel used to explain Statechart, and use UML notation to explain the state machine model.

## 6.4.3 Explanation of state machine model using ”alarm watch” example

Based on Harel’s paper [57], we explain the main notation and meaning of UML state machine model. Figure 6.4 shows a watch with four buttons and an alarm. We will use this as an example.

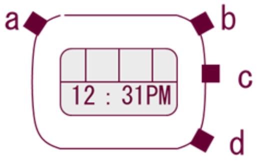


Figure 6.4: alarm watch

In a state machine diagram, states are represented by rectangles with rounded corners, and the name of the state is written in the upper left corner. Transitions are represented by arrows, and the name of the event that causes the transition

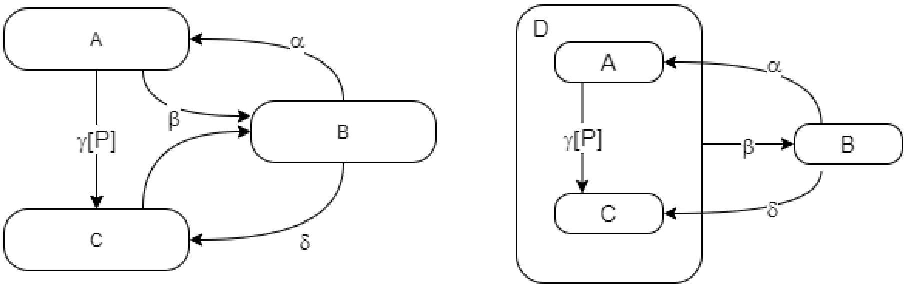


Figure 6.5: State machine model explanation diagram (1)

is given as the label. The transition from state A to state C in figure 6.5 is labeled with [P] after the event $\gamma$. This represents a transition with a transition condition (guard), and the transition occurs only when the event $\gamma$ occurs and the logical expression P is true.

In the state machine model on the right side of figure 6.5, a composite state D with A and C inside is defined. The right and left figures are equivalent. The event $\beta$ exits from the edge of composite state D, which indicates that when $\beta$ occurs, a transition to B occurs regardless of which child state of D is in, i.e., whether it is in A or C.

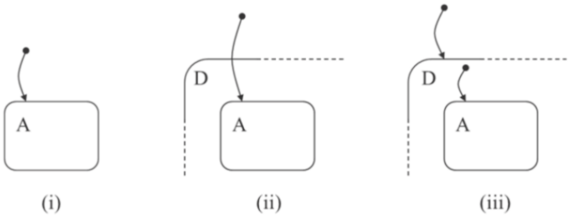


Figure 6.6: State machine model diagram (2)

When a transition is made from outside into this world consisting of states A, B, and C, the first state to be entered is A, as shown in figure 6.6. The tip of the arrow from the black circle is the initial state. In figure (iii), the initial state is D, and within D, A is also shown to be the initial state.

Now, let us actually describe the behavior of the alarm watch.

Figure 6.7 shows the transition from the ”display” state to the state in which the alarm sounds. There are two alarms, one with the alarm time set to T1 and the other to T2. T1 and T2, along with the current time T, are internal variables that can be accessed from any state in this world. The internal variables can change their values as output events accompanying transitions.

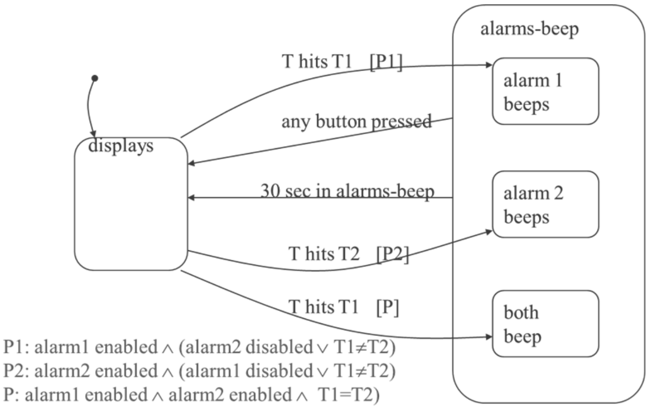
Figure 6.8 shows the internal display state expanded. The initial state is the ”hours” display, and when button d is pressed, it switches to the ”date” display. When button a is pressed, the function switches in order from alarm 1 setting, alarm 2 setting, chime setting, to stopwatch.

In the figure 6.9, a notation called history is introduced. When entering Ⓗ from the outside, as in the left figure, it means returning to the child state (”on” or ”off”) that was in the most recent past when exiting this state (alarm 1). When entering this state for the first time without history, it enters the specified initial state, i.e. ”off”. The figure on the right has the same meaning.

Figure 6.10 shows the valid range of Ⓗ. In the left figure, Ⓗ is specified in K, so the history only records whether it was in F or G. Since the history of which A∼E inside it was not recorded, it enters each initial state (B and C).

On the other hand, the meaning of Ⓗ$^*$ in the right figure indicates that the history extends recursively to child states. In other words, it returns to one of the states A∼E that was in the most recent past.

In the case of figure 6.11, F has an internal history, so it follows that record, but G does not have a Ⓗ, so it starts from



alarms-beep
T hits T1 [P1]
alarm 1 beeps
any button pressed
displays
30 sec in alarms-beep
alarm 2 beeps
T hits T2 [P2]
T hits T1 [P]
both beep
P1: alarm1 enabled ∧ (alarm2 disabled ∨ T1≠T2)
P2: alarm2 enabled ∧ (alarm1 disabled ∨ T1≠T2)
P: alarm1 enabled ∧ alarm2 enabled ∧ T1=T2)

Figure 6.7: State Machine Model Diagram (3)

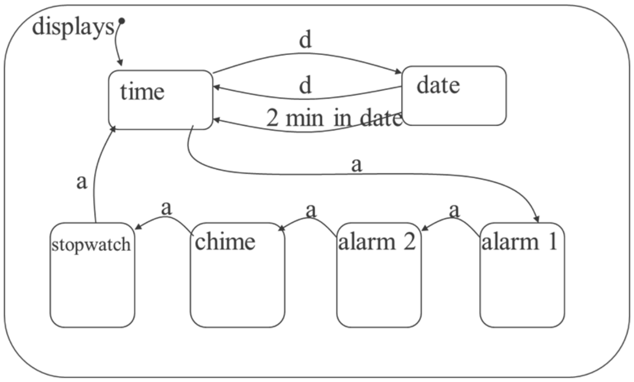


Figure 6.8: State machine model explanation diagram (4)

the initial state B even if it has just left A.

Figure 6.12 uses history to detail the ”display.”

Next, we introduce the notation for concurrent processes.

In figure 6.13, state Y is divided into A and D by a dashed line. In UML, not only are composite states that contain substates as mentioned so far called composite states, but also states that combine multiple state machines with AND are called composite states. Each part separated by a dashed line is called a ”region.” In this example, A and D are regions. When the state A×D is expanded, the number of states increases as a product, but by using the notation for composite states divided into regions, the number of states can be prevented from exploding.

A and D do not behave completely independently, but interact with each other. In this example, by sharing the event $\alpha$, for example, when in state (B,F), if $\alpha$ occurs, both transition simultaneously to (C,G). This is a kind of synchronization, but unlike CSP and CCS, there is no restriction that shared events must transition synchronously. For example, when $\alpha$ occurs in (B,E), a transition occurs only in A, and it moves to (C,E). In CSP and CCS, such transitions are not allowed.

The notation of composite states is useful for describing machines and systems that are composed of multiple components. Figure 6.14 shows an overview of an alarm clock using this notation.

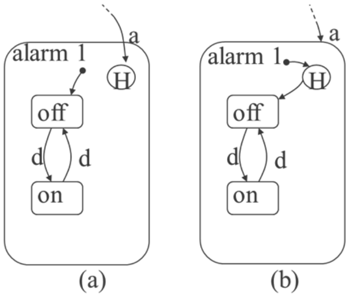


Figure 6.9: State machine model explanation diagram (5)

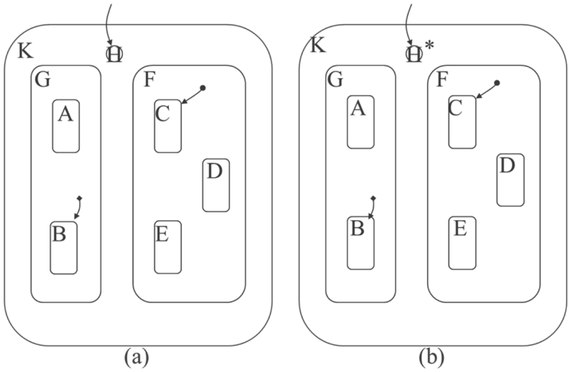


Figure 6.10: State machine model diagram (6)

### 6.4.4 Vending Machine Problem

Draw a state machine diagram for the objects ”cashier” and ”sales” that are assigned roles in the sequence diagram and communication diagram (Figure 6.15, Figure 6.16). The behavior of each state machine should be consistent with the sequence on each time axis of the sequence diagram. Here, the notation for output events and transition conditions appears again. In ”Sales completed [balance = 0] / balance notification”, [balance = 0] represents the transition condition, meaning that the transition occurs when the condition ”balance = 0” is met after the sales completion event has occurred, and the ”/balance notification” part represents the output event, meaning that the ”balance notification” event occurs at the same time as the transition.

As an example of decomposing a state into substates in a state machine model, the ”On Sale” state in sales is decomposed in figure 6.17.

#### Application to algorithm description

Finite state machines are effective for describing logical algorithms such as logic circuits, but they can also be applied to more practical and sequential algorithms. As an example, let’s model the process of calculating bowling scores.

The rules for calculating scores are as follows.

1. The score for a strike is 10 plus the number of pins knocked down in the next two throws.

2. The score for a spare is 10 plus the number of pins knocked down in the next throw.

3. The score for an open frame is the number of pins knocked down in the two throws in that frame, with no addition.

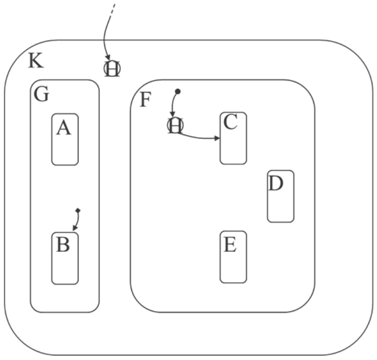


Figure 6.11: State machine model diagram (7)

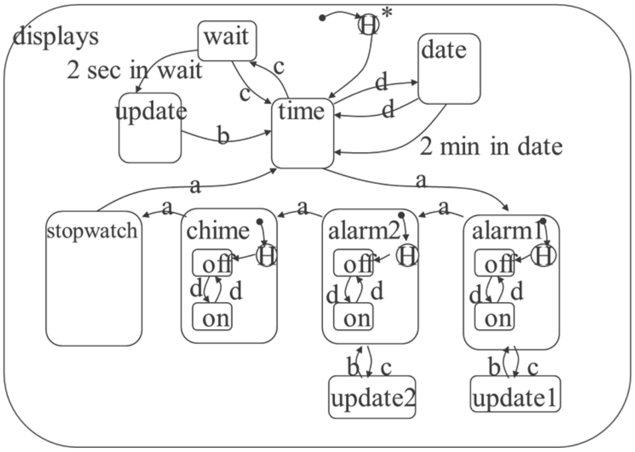


Figure 6.12: State machine model explanation diagram (8)

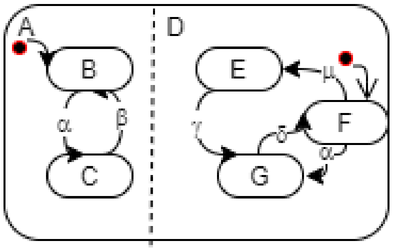


Figure 6.13: State machine model explanation diagram (9)

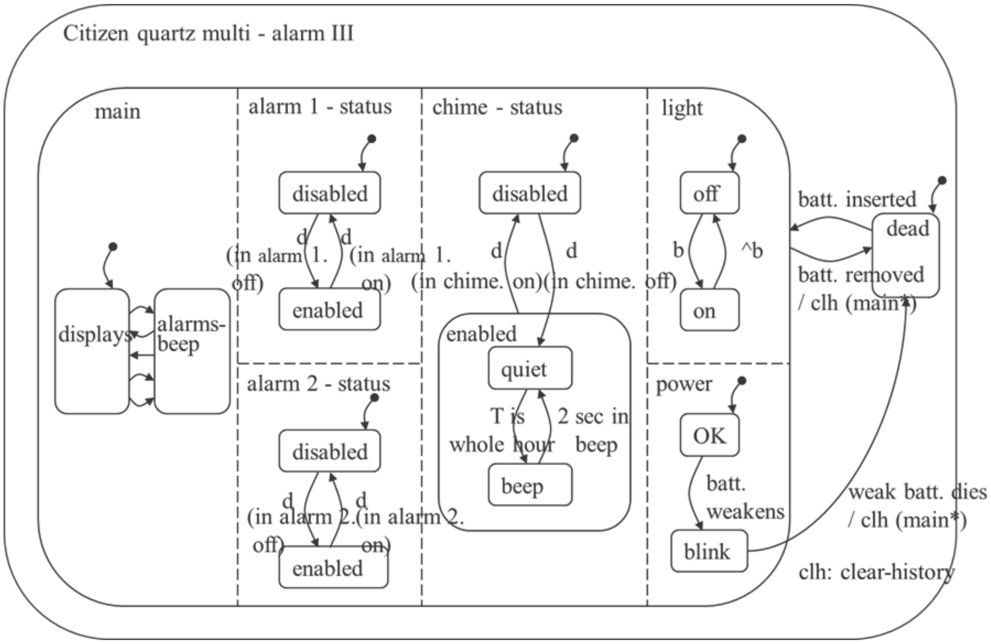


Figure 6.14: State Machine Model Diagram (10)

Actually, special processing is required for the final 10th frame, and there are rules for this, but we will not consider them here.

Figure 6.18 is an example of a state transition diagram modeling the score calculation process.

In this figure, the set of states is {start, open, spare, strike, double, 2 pitches before, strike and 2 pitches before}. Why does the concept of ”double” appear, which does not appear in the rules mentioned above? Conversely, why is there no state of turkey with 3 consecutive strikes (or forth or more)? Or is it okay if something equivalent to the concept of frame does not appear directly?

The set of events is {all down, remaining}. It’s a bit of a lame way of saying it, but ”all down” means that all the pins standing have been knocked down (a strike on the first throw, a spare on the second), and ”remaining” means that one or more pins remain. Why not distinguish between events based on the number of pins knocked down or the number of pins remaining?

These were selected based on the model construction policy. To create a model, there is a purpose. An ”abstraction” must be performed, in which the things necessary for the purpose are selected and the unnecessary things are discarded. Here, we are considering a state that represents the case distinction required to calculate the bowling score. Therefore, the event is the point in time when the ball is thrown and the pins are knocked down, and it is not directly distinguished as the first throw or the second throw, but is abstracted into a single distinction between whether all the pins have been knocked down or not.

This state transition diagram does not explicitly state the end state. Because of the abstraction described above, the concept of a frame is not expressed, and even if an actual bowling game ends in 10 frames, this is not reflected in the process described here.

The bowling score calculation algorithm in Figure 6.18 is useless without the output of information to calculate the score. To do this, we will consider the following three output events.

1. Addition 1 (add the number of pins knocked down)
2. Addition 2 (add twice the number of pins knocked down)
3. Addition 3 (add three times the number of pins knocked down)

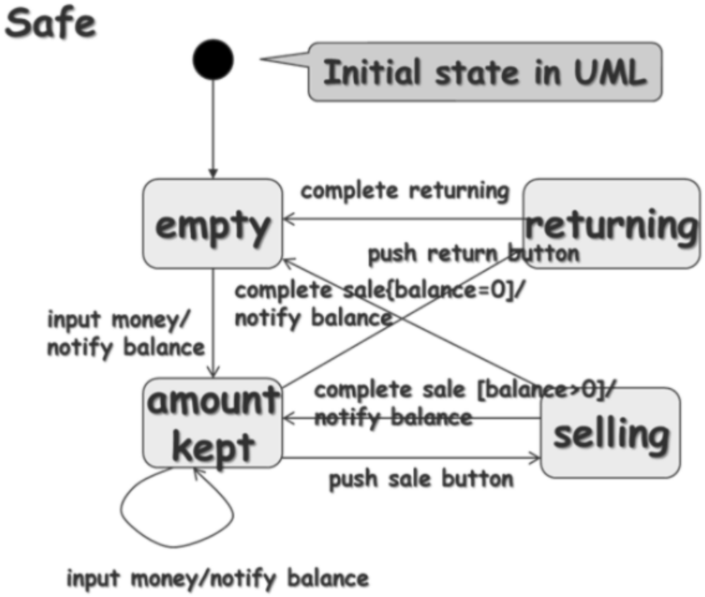


Figure 6.15: State machine diagram of cashier

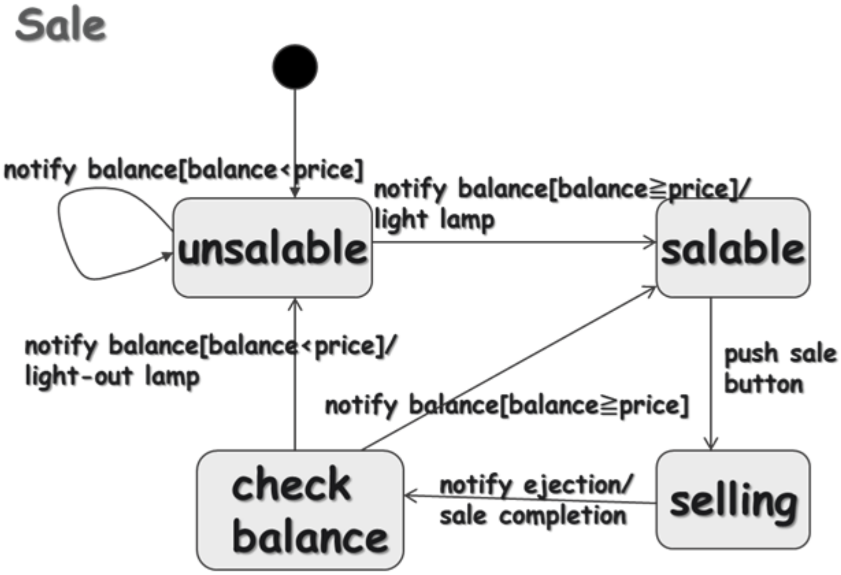

Sale
notify balance[balance<price]
notify balance[balance≧price]/
light lamp
unsalable
salable
notify balance[balance<price]/
light-out lamp
push sale
button
notify balance[balance≧price]
check balance
notify ejection/
sale completion
selling

Figure 6.16: State machine diagram of sale

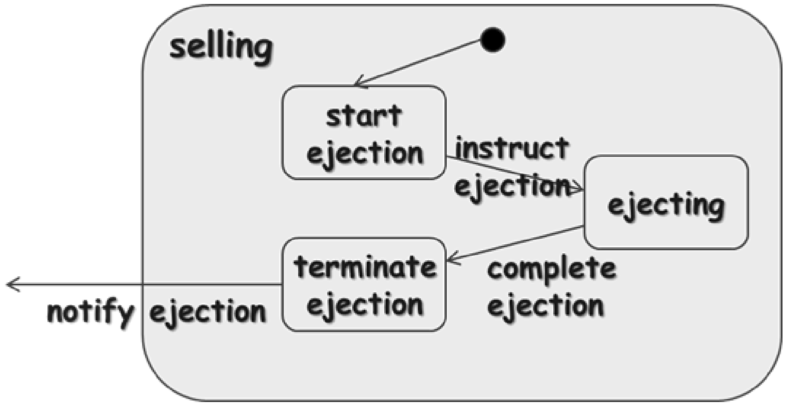


Figure 6.17: State machine diagram for sale

Here, the parentheses are comments that indicate the calculations that would be performed by an external computational entity that receives this output event. For the state transition model in Figure 6.18, all that matters is that three events are distinguished. If we want to match the usual frame-by-frame score calculation, it might be better to say

1. Addition 1 (add the number of pins knocked down to the current frame)
2. Addition 2 (add the number of pins knocked down to the current and previous frames)
3. Addition 3 (add the number of pins knocked down to the current, previous, and previous frames)

This representation is easier to handle, especially when considering the processing of the 10th frame.

Figure 6.19 shows the diagram with this output event added. The edges that represent transitions are labeled with ”input event/output event”.

### 6.4.5 Synchronous and Asynchronous

When considering a concurrent system, it is important to consider whether the processing is synchronous or asynchronous. However, care must be taken because the terms synchronous and asynchronous are used with different meanings depending on the context.

#### Synchronization by Clock

When multiple objects (hereafter called processes) are operating concurrently, there is a model in which the timing of their state transitions is synchronously controlled by a clock. The clock ticks discretely at equal intervals, and each process receives events simultaneously according to that period and transitions accordingly. This method is often used in hardware such as electrical circuits. The semantics of Statechart operation is often a problem, but it is basically a synchronous system that follows this clock. All processes that appear in Statechart are interpreted as transitions due to events according to a certain clock interval.

#### Synchronization by Sharing Events

When multiple processes run without clock control, they basically live in an asynchronous world, but nothing interesting happens if they run independently without any interference. The complexity and interest of concurrent systems comes

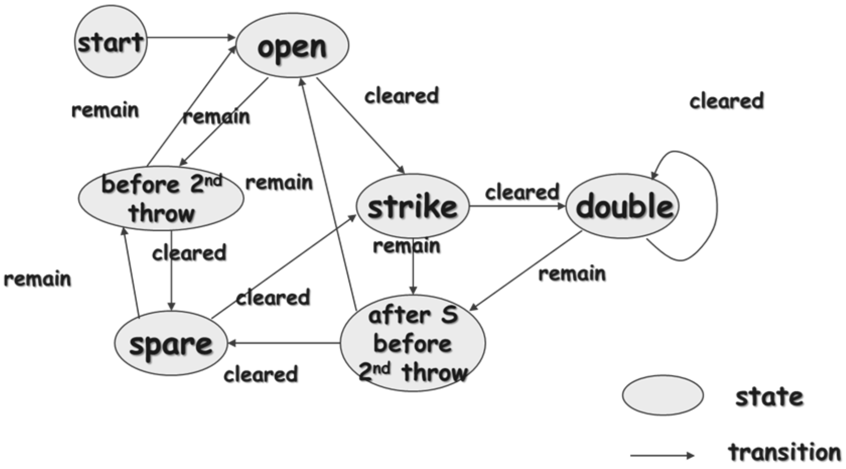


Figure 6.18: State transition model for bowling score calculation

from some kind of interaction between processes. In a state transition model, this interaction occurs when multiple processes share events.

In figure 6.20, two processes X and Y do not share any events. Therefore, X and Y behave independently. The event sequence accepted by process X is $(ab)^*$, and the event sequence accepted by process Y is $(c(de)^*df)^*$. Here, $()^*$ means 0 or more iterations of the string in parentheses.

The semantics of concurrent processes does not assume that multiple machines physically execute each process, but assumes that the event sequences of multiple processes can be executed by a single machine through interleaving. Interleaving is the merging of sequences while preserving the relative order of elements in each sequence. In the above example, a sequence such as (a,c,d,b,e,a,d,f,b) can be considered interleaved because it preserves the order of events in X and Y.

In the figure 6.21, processes X and Y share an event b. Hoare's CSP and Milner's CCS, which algebraically theorize concurrent processes, and Magee & Kramer's FSP, which is based on CSP, provide the semantics that when processes share an event, they must synchronize on that event. In the case of figure 6.20, because b is shared, process Y must wait for process X to transition from A to B before it can transition from C to D. Only when X is in B and Y is in C can they transition due to event b. In terms of the semantics of interleaving, no arbitrary merging of events accepted by X and Y is permitted, only events that can overlap b are permitted.

However, in Statechart, which assumes synchronization by a clock, the interpretation is different. If, when event b occurs, X's state is A and Y's state is C, only Y transitions to D and X remains in A. This is the semantics of Statechart.

## 6.5 Jackson System Development Method (JSD)

Jackson System Development Method (JSD) is a system development method devised by British consultant Michael Jackson. Jackson had previously proposed Jackson Structural Programming (JSP), but JSD extends it to be applicable to the entire system development. Its contents are explained in detail in his book "System Development" [68].

JSD preceded the proposal of object-oriented analysis/design methodology by $7 \sim 8$ years, and its ideas are particularly ahead of object-oriented methodology that focuses on dynamic behavior. The reason we take up JSD here as one of the dynamic behavior models is that the model used there is constructed with a focus on the entities or processes that cause dynamic behavior and the dynamic interactions between them.

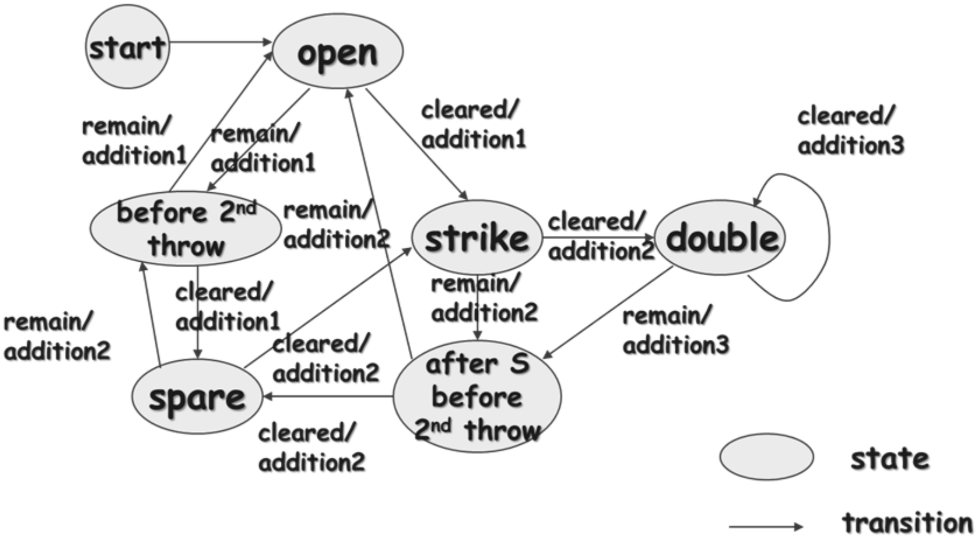


Figure 6.19: Bowling score calculation model (with output events)

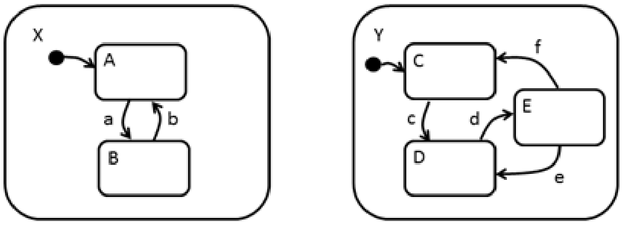


Figure 6.20: Independent Concurrent Processes

### 6.5.1 Basic Structure of the JSD Model

The idea of the JSD model is based on Hoare's CSP model. In other words, the world is considered as a situation in which many processes behave dynamically while exchanging messages with each other.

Therefore, to describe the model, the following two basic elements are considered.

1. Action or event: Something that occurs in the real world. It has attributes.
2. Entity: The subject of an action. It is something that causes a series of actions (events) along a time axis, and in that sense it can be considered a sequential process.

Here, an entity is defined as a subject that causes a series of actions, and it should be noted that the structure and behavior of an entity depend only on the structure that shapes the actions associated with it and the nature of the behavior that results from it. In that sense, there is a difference between objects in the object-oriented world, where objects exist in the first place and entities in JSD where they only denote the structures of action sets..

### 6.5.2 Development Procedure

The development procedure for a system using JSD consists of the following six steps.

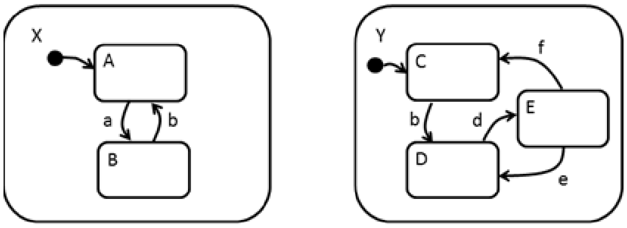


Figure 6.21: Concurrent processes that share events

1. Entity action step
2. Entity structure step
3. Initial model step
4. Functional step
5. System timing step
6. Implementation step

In particular, in terms of modeling techniques, steps 1 to 4 are central.

To explain the JSD procedure, we borrow an example used in Jackson's book [68].

**Bank deposit transactions** "In a bank deposit transaction, a customer first opens an account, and then repeatedly deposits and withdraws money. When the account is canceled, it is closed, and no further deposits or withdrawals are possible. This account allows overdrafts, and there is no interest on deposits or overdrafts."

**Function requirements**

1. The customer's balance can be inquired at any time.
2. When an overdraft occurs, a report is issued.

### 6.5.3 Entity Action Step

In the entity action step, the actions and entities in the target domain are identified. If the target world is described as some kind of text, the candidates for entities are the nouns that appear in the text, especially those that form the subject, and the candidates for actions are verbs. Attributes may be attached to actions, and the candidates for these are object words. After identifying these, they are selected. Furthermore, entities and actions that do not appear directly in the text but are deemed necessary through insight are added.

The entities and actions identified in the deposit transaction example would be as follows:

| | | |
|---|---|---|
| Entity | | |
| | Customer | |
| Action | | |
| | Open | Attributes (Amount) |
| | Deposit | Attributes (Amount) |
| | Withdraw | Attributes (Amount) |
| | Close | Attributes (Amount) |

When listing entities and actions, Jackson's student J. R. Cameron[30] emphasizes that actions come before entities. This is one of the differences from normal object-oriented approach. It is interesting to compare this with DeMarco's advice to focus on the flow of data before the process.

In fact, actions require separate and proper description of their contents, but entities are positioned as actors of those actions, and the sequence, selection, and iteration of actions themselves define the entities. Therefore, there is no need to give entities any other definitions or associated attributes.

Entities identified in this way are suitable for modeling objects that have dynamic properties. Jackson acknowledges that they are not suitable for describing static data relationships. For example, describing data on the composition ratios of chemical compounds is not the purpose of JSD.

### 6.5.4 Entity Structure Step

In the entity structure step, entities are defined as structures formed by actions. The notation used to express the structure of actions is the Jackson structure diagram, which is also used in JSP. Basically, it is expressed as a tree structure using three components: sequence, selection, and iteration. There is a rule that all edges coming out of one vertex are of the same type (sequence, selection, iteration). There is also a text-type notation equivalent to this diagram.

The expressive power of this notation is equivalent to regular expressions. As a result, entities are perceived as subjects that perform sequential actions. There may be cases where regular expressions are insufficient in terms of modeling capabilities, but Jackson shows several ways to deal with such cases, such as decomposing into multiple entities.

Figure 6.22 shows the structure of the entity called "customer" identified in the previous step in the deposit transaction example.

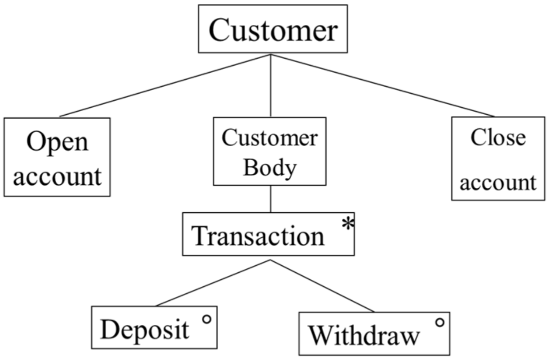
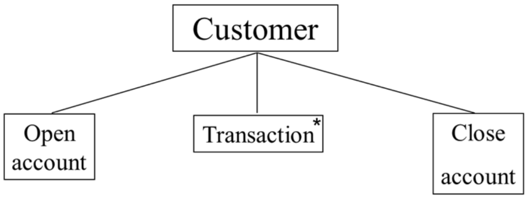


Figure 6.22: Jackson Structure Diagram

A * on the right side of a box representing an action indicates iteration, and a ○ indicates selection.

Note that the structure at the end of a branch branching from a single vertex must be either a sequence, selection, or iteration, and must not be a mixture of these. For example, it may seem more concise to draw the level just below "customer" as



but this is not permitted by the rules.

### 6.5.5 Initial Model Step

In the initial model step, the specification of the system to be developed begins to be written, and the method is based on the idea of simulating the real world. To define this simulation, we consider model processes in the system that correspond to real-world entities. Since model processes are also sequential processes that simulate real-world entities, they usually have a structure similar to the structure of the entity's behavior. In JSD, we use the notation that a real-world process is represented by a hyphenated entity name with 0 appended, and a model process is represented by a hyphenated entity name with 1 appended. For example, if there is an entity called ”customer”, there could be processes called ”Customer-0” and ”Customer-1” (usually, these processes are created for each individual customer instance).

There is a strict boundary between the real world and the system. Therefore, an important element of the model is what kind of connection is made between them. This connection usually takes the form of a real-world process performing some operation and transmitting the result to a process in the system, so it takes the form called a data stream connection. In practice, this defines the interface between humans and machines.

In the bank deposit example, we consider processes ”Customer-0” and ”Customer-1” and connect them with a data stream connection. This means that when a customer makes a deposit or withdrawal at a bank teller or ATM, data flows into the process corresponding to the customer in the system.

If we illustrate this relationship, we get a nearly self-explanatory diagram6.23.

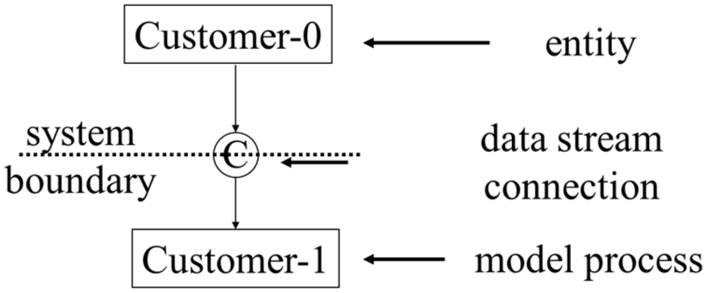


Figure 6.23: Introduction to the model process

Here, the circle represents the data stream connection, which we name C. The arrow indicates the direction of the data stream flow.

Connections can of course also be made between model processes. As we will see later, processes that do not have a corresponding physical entity in the real world are gradually added to the system, and connections are also made between these processes. This allows the entire system to be modeled as a network, including its interface with the outside world. There are two types of network connections between processes:

- Data stream connections
  These are almost the same as the connections between processes in a data flow diagram. However, they take into account the control relationship of synchronization between the sender and receiver of data. The sender initiates the data flow, and the receiver waits for it to synchronize. This type of connection is shown in a circle in the diagram.
- State vector connections
  In this case, the receiver of the data unilaterally looks at the data held by the other party. Therefore, this state data is interpreted as something that represents the internal state of the process that holds it and is made publicly available to the outside world. In practice, this is often realized as a database or shared memory. This type of connection is shown in a diamond shape. An example of this will be shown later (Figure 6.24).

### 6.5.6 Functional Steps

In functional steps, processes to fulfill the functions of the system are introduced as necessary, and connections between them and model processes and between functional processes are determined. At first glance, it may seem confusing that the concept of functional processes to realize functions appears separately from the model processes that represent the entities, but upon careful consideration, it becomes clear that it is important as a bridge from the model to the design.

In the deposit transaction example, there are two functions. The overdraft report is detected in the internal state of Customer-1, so it is natural to add the processing to realize this function to Customer-1's actions. On the other hand, the balance report is performed in response to an inquiry from outside. It is natural to introduce a new process for this purpose, so we introduce it as a process named ”inquiry function”. In order for this process to function, it is necessary to look at the internal state of Customer-1. This is where the state vector connection occurs. The result is a network diagram like the one in Figure 6.24.

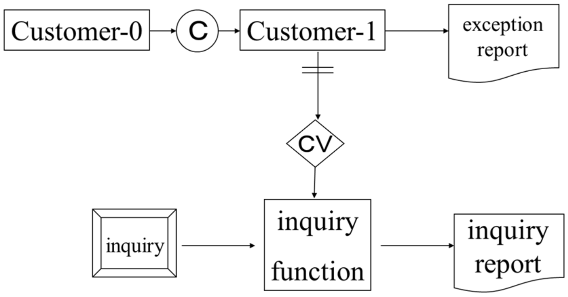


Figure 6.24: Network diagram

Here, the diamonds indicate state vector connections, and are given the name CV. Data flows in the direction of the arrow, that is, from Customer-1 to the inquiry function, but the subject of the action is the inquiry function. In other words, the inquiry function reads the state vector that represents the internal state of Customer-1.

The arrow between Customer-1 and CV has two lines on it, which indicates that there are many Customer-1 processes, which are connected to one inquiry function process. In other words, it is a many-to-one connection. In the case of many-to-many, two lines are also added to the other arrow.

The graph properties of the model represented by this network diagram are as follows.

| JSD network diagram | |
|---|---|
| **mode:** | dynamic |
| **object:** | information flow |
| **vertex:** | process |
| **edge:** | data stream connections, state vector connections |
| **subgraph:** | lower network |

### 6.5.7 System Timing Step

The system timing step deals with consideration of process execution time, timing and synchronization between processes, etc.

### 6.5.8 Implementation Step

Up to this point, the problem is analyzed and the system specifications are created, and the concrete design is carried out in the implementation step.

1. Data design
   When implementing, processes are generally assumed to correspond to modules (procedures, subroutines, functions, etc.) (but this is not necessarily the case), and state vectors and stream data are implemented as module arguments, files, or databases, for example. The choice of which to use depends on the individual situation.

2. Process coupling
   If modeled straightforwardly with JSD, many processes will be created. It may be no longer so unrealistic to run these directly on a multiprocessor, but it is not so when the number of processes is enormous. There are various techniques for coupling these processes and making them run in a normal environment.

   One is to place a scheduler. In many cases, this is necessary in any case as the so-called main program for input/output and execution management.

   Also, when many instance processes of one entity class are created, their internal states can be organized into one by arraying them, or by other methods.

   Two or more processes exchanging messages can be coupled via a buffer, or can be implemented using coroutine techniques. There are also known techniques for folding coroutines to make a single program.

   Conversely, there are cases where it is better to split the process. Either way, allocating the processes identified in this way to processors and scheduling them is a major implementation task.

### 6.5.9 The Sake Warehouse Problem

Let's try to deal with the sake warehouse problem using JSD. The entities that naturally emerge are the container, the customer who orders sake, the receptionist, and the warehouse clerk. First, if we draw an entity structure diagram of the container and the customer, it will look like Figure 6.25.

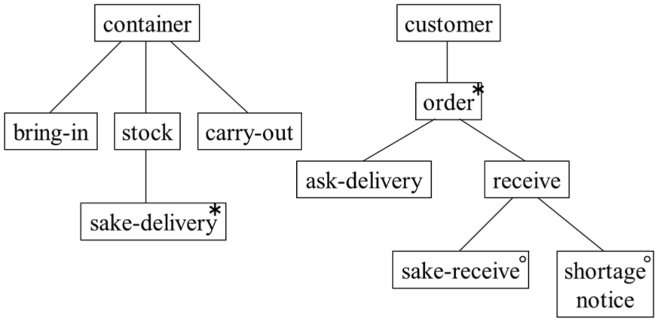


Figure 6.25: Entity structure diagram of container and customer

The receptionist corresponds to the system to be built, but if we straightforwardly structure the work of a receptionist in the real world, it would look like Figure 6.26.

The warehouse staff can also be represented in an entity structure diagram in the same way. However, it is not so easy to proceed with the modeling beyond that. Here is a sample answer by Toshiro Ohno, so I will introduce it here[150]. First, three entities are selected: container, sake brand, and orderer. Leaving aside the container and orderer (the name ”customer” is used above), it seems odd that a relatively small concept like sake brand has been selected as an entity. Looking at the structure diagram of these entities, it is drawn as in Figure6.27.

Here, the entity structure diagram of the container is the same as in Figure6.25. However, the structure diagram of the orderer is quite different from the customer in Figure6.25. Isn't it strange that the orderer is the subject of the actions of ”fulfillment” and ”shortages”? Furthermore, there is no iteration in the structure diagram of the orderer, so is it okay for

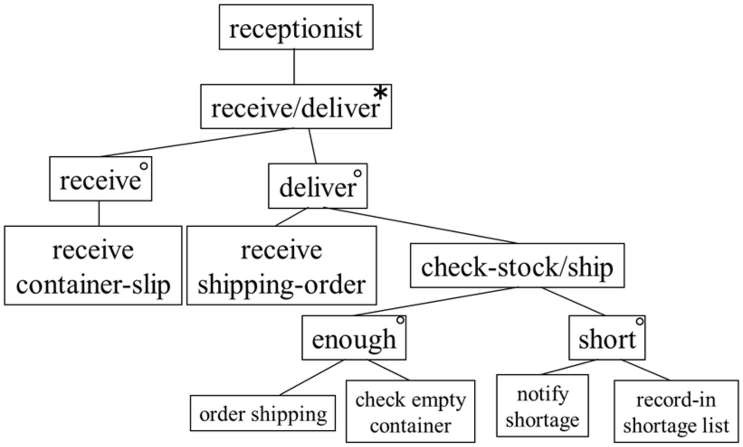


Figure 6.26: Entity structure diagram of receptionist

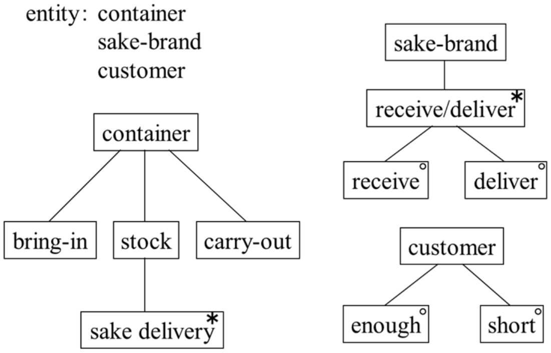


Figure 6.27: Entity structure diagram in Toshiro Ohno's solution

it to disappear after taking either the fulfillment or lack action once? The ”arrival” and ”shipment” that appear in the sake brand are actions that also appear in the receptionist in Figure6.26, but do they both represent the same movement?

Looking at the network diagram of the initial model (Figure 6.28) created by generating model processes from these entities, we can answer these questions to some extent.

The most noteworthy point is that the sake brands are internal processes of the system. In Ohno's paper, it is stated that sake brands are extracted as real-world entities, but as is clear from this diagram, sake brands are processes that are given roles in the system's configuration. In fact, the essence of this sake warehouse problem is that the stock of sake into the warehouse is done in containers, and a certain brand of sake is generally divided into several containers, while sake orders come in by brand, so it is necessary to relate these two sets of data. A software engineer with a lot of experience would intuitively grasp the essence of such a problem and realize that a process is needed to hold information on a brand-by-brand basis, such as the name of a sake brand. However, as an example of a development methodology such as JSD, an explanation of the process of finding such a process would be necessary.

The model process called ”orderer” seems to correspond to an individual order. Therefore, once the processing of that order is completed, the process disappears. In that sense, it seems better to call it ”order” rather than ”orderer”.

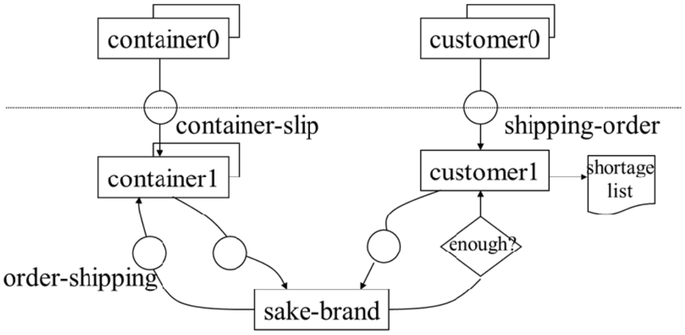


Figure 6.28: Network diagram of Toshiro Ohno's solution

### 6.5.10 Vending Machine Problem

Let's try to deal with the vending machine problem using JSD. First, the real-world entity is the customer who buys a drink. The entity structure diagram would look something like Figure 6.29.

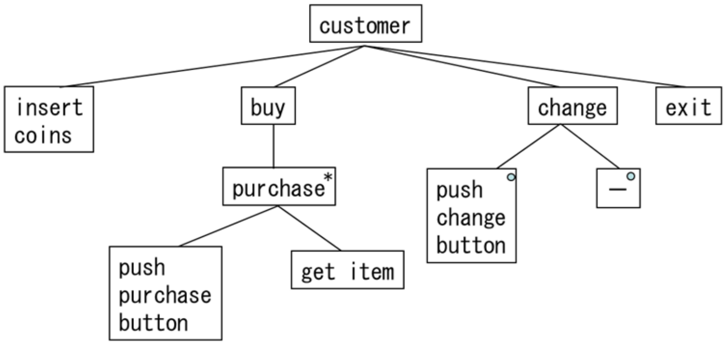


Figure 6.29: Customer behavior diagram

Next, Figures 6.30 and 6.31 show the actual structure diagram of the cashier and the sales process. In reality, the process of finding such a process should be traced according to the JSD, but since similar analyses have already been performed using sequence diagrams and communication diagrams, this has been omitted.

The network diagrams created by these are almost the same as the communication diagram in figure 6.3. However, here the ”customer” is an entity outside the system, and according to the JSD style, it is a process that would be named Customer-0.

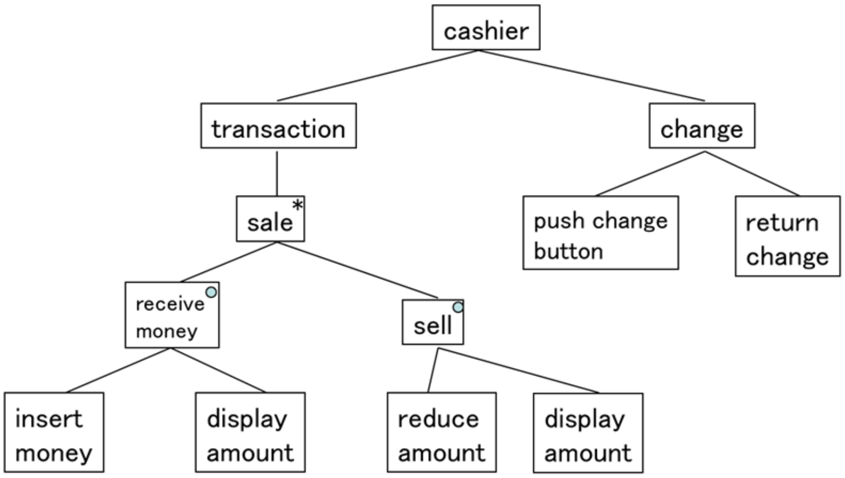


Figure 6.30: Entities of cashier

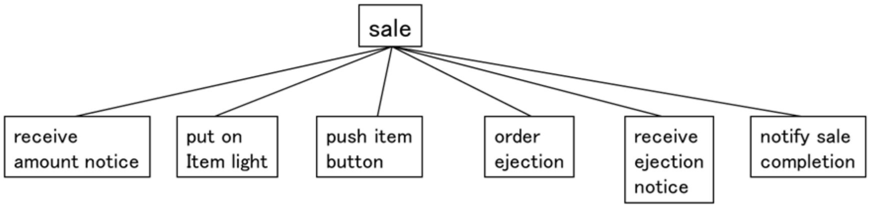


Figure 6.31: Entities of sales

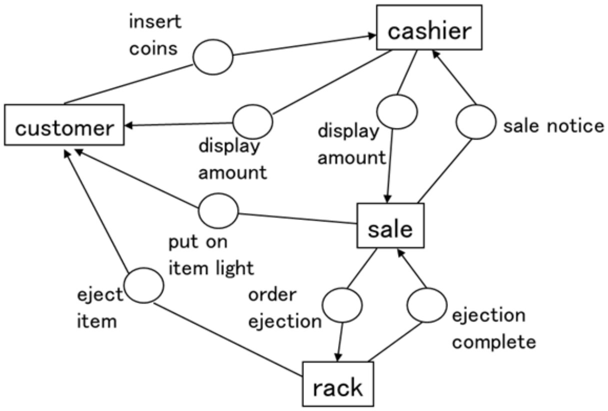


Figure 6.32: Network diagram of vending machines

# Chapter 7

# Object-Oriented Model

In the object-oriented model, the world is considered to be a collection of objects.

An object is a ”thing” that corresponds to an individual physical object in the real world or a concept in the conceptual world, and object orientation is a general term for the idea and technology of building system models and software based on such ”things.”

## 7.1 History of Object-Oriented Technology

The concept of object orientation began to attract attention in the early 1980s, when the specifications for the object-oriented programming language Smalltalk 80 were made public. Since then, object-oriented programming languages such as C++ and Java have been developed and spread. In addition, from the end of the 1980s to the mid-1990s, methodologies for object-oriented design and object-oriented analysis were published one after another, tracing the software life cycle back from programming to design and analysis. This process is similar to the emergence of structured design and structured analysis after structured programming became popular in the 1970s.

Smalltalk 80 was the catalyst for the current idea of object orientation to become widespread, but if we trace its origins back to the 1970s, we can see that most of the basic concepts of object orientation were proposed by the 1970s. They have grown independently in various fields, but let’s take a look at them individually below.

1. **Simulation languages**

   It is well known that one of the leading programming languages that had a major impact on Smalltalk was Simula, a simulation-oriented language developed by Ole-Johan Dahl and Kristen Nygaard in Norway in the 1960s. The first specification for Simula was created in 1962, but it was revised several times afterwards and was completed in 1967 with Simula 67. Simula 67 has the basic properties that a modern object-oriented programming language should have, such as the concepts of classes and inheritance, and its foresight is admirable. It is noteworthy that the first idea of object orientation came from the field of simulation, in relation to the excellent modeling capabilities of object orientation.

2. **Abstract data type**
   In the 1970s, David Parnas proposed a modular specification method based on the concept of information hiding, which led to research on abstract data types, and CLU (1976), Alphard (1976), and EUCLID (1977) were proposed as programming languages that have abstract data types as grammatical constructs. Abstract data types roughly correspond to objects as units of calculation with an information hiding mechanism. It was also in the 1970s that a framework for algebraic specification was proposed and researched as a theory that gives the precise meaning of abstract data types.

3. **Knowledge representation**
   In the field of artificial intelligence, knowledge engineering was also proposed in the 1970s, and a method of collecting and describing knowledge of individual problem domains and using it in practice was vigorously pursued.

Among them, the frame-based representation proposed by Marvin Minsky in 1975 as a means of knowledge representation attracted attention, and knowledge representation languages such as KRL (1977) and KL-ONE (1985) were developed based on this as a basic construct. Frames have slots as components and slots can contain various types of data, relationships, and actions. Therefore, conceptually, they overlap almost completely with objects.

4. **Semantic data model**
   In the database field, the process of constructing a data model is essential for the conceptual design of a database. The Entity Relationship Model proposed by Peter Chen in 1976 is a widely used model that describes the target world in terms of entities and the relationships between entities. Entities are considered to correspond to objects, and this model is very close to the class model, which represents the static structure of objects.

5. **Dynamic process model**
   While the entity-relationship model describes static structures, models that describe the behavior of objects that exhibit dynamic behavior are used as a basis, such as the ACTOR model created by Carl Hewitt in 1973. Actors correspond to the dynamic aspects of objects, and provide one of the theoretical foundations of the object-oriented computation model.

6. **Multimedia**
   Smalltalk, developed at the Xerox Palo Alto Research Center by Alan Kay, was originally designed to be a programming language for personal computers that even children could use. The Alto personal computer, which ran Smalltalk, featured all the prototypes of today's GUIs (Graphical User Interfaces) and multimedia environments, such as bitmap displays, window systems, mice, and pop-up menus, and it was shown that object-oriented programming is effective as a basic concept for manipulating them.

These various trends have merged to form the rich mainstream of object-oriented programming.

## 7.2 Basic Concepts of Object-Oriented Models

When we say object-oriented models, we mean either a computational model that shows how objects, which are computational entities, interact with each other to advance computation, or a model that uses object-oriented concepts to express problem domains in various fields. When we say that object-oriented programming has rich and natural modeling capabilities, we are referring to the latter model, but where does this modeling capability come from? It is derived from the former computational model, so here we will first describe its characteristics as a computational model.

The basic element of object-oriented models is, of course, objects. An object is a target or concept that is recognized as a whole. Objects have static and dynamic properties. Attributes represent static properties, and actions or methods represent dynamic properties, which are the basic units of behavior. This process of combining a set of attributes and actions into a single unit is called encapsulation.

There are several types of relationships between objects, as shown below.

1. **Classes and instances**
   A collection of objects of the same type is called a class, and each object that belongs to a class is called an instance. In the narrow sense, an object refers to an instance, but some consider classes to be a type of object. For example, in Smalltalk, classes are treated as objects with their own behavior (methods).

2. **Generalization**
   A hierarchical structure is created by the inclusion relationship between classes as a set. A higher class contains a lower class as a set. A lower class also inherits the properties of the higher class.

   This type of hierarchical structure has long been used in the classification of biology. A class at a higher level in the hierarchy is called a generalized class of the lower class.

   Conversely, a lower class is called a specialized class of the higher class.

3. **Aggregation**
   When object B forms a part of object A, object A is called an aggregation of object B. For example, a car object is an aggregation of engine, body, and wheel objects. This is also called a part-whole relationship.

4. **Association**
   A relationship is a general relationship between classes. The relationship between two classes is expressed as a subset of the Cartesian product of each class. For example, there can be a relationship between a class called ”student” and a class called ”subject” called ”course enrollment.” In this case, it is a so-called many-to-many relationship, but there can also be one-to-one and one-to-many relationships. Generalization and aggregation can also be considered special cases of relationships.

5. **Dependency**
   When defining the specifications of an object (class) A or when implementing it, if another object B is used, changing the specifications of B will affect A. In this case, A is said to depend on B.

## 7.3 Object-Oriented Development Methodology

Typical object-oriented analysis/design techniques include the Booch method, OMT by J. Rumbaugh et al., OOSE by I. Jacobson, Coad & Yourdon, Shlaer & Mellor, Martin & Odell, etc. Many books have been written about these. [24, 37, 43, 87, 71, 115, 120, 121, 142]. In the mid-1990s, there was a movement to integrate these methods. In particular, the highly influential J. Rumbaugh and G. Booch joined forces to integrate the OMT method and the Booch method, and aimed to build a unified method that also incorporated I. Jacobson’s OOSE. Although it is difficult to say that the methodology is unified, UML has been created as a notation, and many tools based on it have been released and are widely used (see 4.4 section).

Many of the methods proposed so far have been explained using UML as a notation.

Development methodologies based on object orientation aim to take advantage of the following characteristics of object orientation.

1. **Natural modeling ability**
   It is easy to create an analysis and design model that naturally maps the domain where the system is to reside in.

2. **Incorporating existing excellent modeling methods**
   For example, it is possible to utilize the accumulated modeling methods such as class classification, association structure, and state transition.

3. **Incorporating existing excellent design methods**
   It is possible to utilize the accumulated design methods such as encapsulation, polymorphism, common/differentialization, and design patterns.

4. **Descriptive power of concurrent systems**
   It is effective for describing and building concurrent systems based on a model in which objects work concurrently and in cooperation. For example, the concurrent object-oriented language ABCL[143] by Akinori Yonezawa was developed by taking advantage of the fact that object orientation is suitable for describing concurrent systems.

5. **Seamless development process**
   The conceptual cconstruct of objects is used consistently through the analysis, design, and implementation stages. Of course, the division of the work process into analysis, design, and implementation phases is also necessary in object-oriented development, but the differences in notation and ways of thinking between them are much smaller than, for example, in structured analysis, structured design, and structured programming.

## 7.4 Process of Building an Object-Oriented Model

Booch, Rumbaugh, and Jacobson have each published one book on UML. Booch’s book is an introductory book to UML [25], and Rumbaugh’s is a language specification for UML [116], but Jacobson’s book is entitled ”Unified Software Development Process” and deals with a development process that places emphasis on use cases inherited from OOSE [70]. To use UML not just as a notation but as a methodology, it is necessary to refer to an explanation of such a process. Below, we will briefly explain Jacobson’s process.

### 7.4.1 Description of Use Cases

A use case is a typical scenario in which a user uses a system, described for the purpose of clarifying the requirements for the system. Therefore, it roughly corresponds to the scenario described in the chapter on requirements analysis. In a use case, the role of an external person who interacts with the system is called an actor. A diagram showing the relationship between actors and use cases is called a use case diagram. Use case diagrams are recognized as one of the UML diagrams.

For example, Figure 7.1 is an example of a use case diagram. The ovals here represent use cases, and the figures are actors. In the diagram, `<<include>>` indicates that a use case uses another use case as part of itself. Also, `<<extend>>` indicates that a use case extends another use case. This is used, for example, when describing the process for when an exceptional event occurs by extending a normal use case.

Use case diagrams are similar to other UML diagrams that have appeared so far in the sense that they are diagrams with a graph structure, but there is little additional information that can be obtained by making them into diagrams. It is not particularly superior to listing use cases and their related actors.

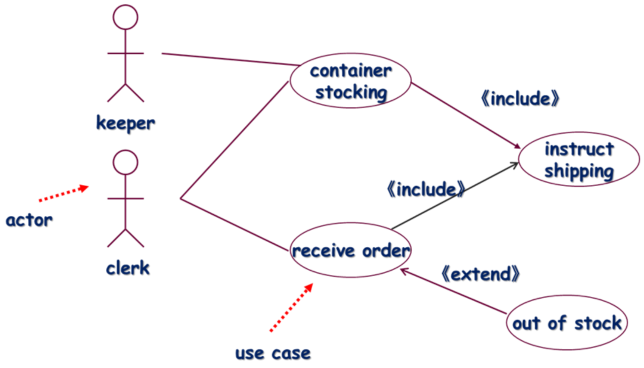


Figure 7.1: Use case diagram for the sake warehouse problem

Listing use cases first is a method that is emphasized mainly by OOSE. In many early object-oriented analyses and designs, the first task was to define the objects that make up the target model and clarify the structure they form. Considering the interaction between the system and the user is equivalent to defining functional requirements, so it was considered to be a task that should be done after the object model was created. However, it has been pointed out that even if the objects in the target domain and their relationships are clarified, it is difficult to get an image of the system. There is also the problem of what clues should be used to determine the right set of objects that have semantic cohesion in the first place. Use cases clarify the system's operational image at an early stage, and through the process of constructing the objects responsible for realizing those use cases, they are said to lead to the identification of objects.

Use cases can be said to replace the functional models described in data flow diagrams in OMT, and the reason data flow diagrams have not been adopted in UML is probably because use cases were introduced.

The following items are described in a use case:

1. Use case name
2. Actors: Participants in this use case
3. Preconditions: Conditions that must be met before this use case is executed
4. Basic sequence: A description of the interactions between the system and actors in the use case, in chronological order
5. Postconditions: Conditions that must be met after the use case is executed
6. Alternative sequence: A description of a case that is similar to the basic sequence, but which splits into a separate sequence at some point due to conditions

### 7.4.2 Identifying classes

Classes are types of objects that have a semantic unity, and in the analysis stage, they naturally correspond to real-world objects and concepts. In the design stage, system-specific classes are added, and classes in the analysis stage are divided, changed, and integrated.

A class has a name, a set of attributes, and a set of operations.

### 7.4.3 Creating a class diagram

Classes have various relationships with each other. A class diagram illustrates these relationships. The relationships depicted in class diagrams are usually static. In particular, it is customary to distinguish and describe relationships such as generalization (or, conversely, specialization or inheritance), aggregation, association, and dependency.

Other diagrams used in object-oriented models, such as communication diagrams, sequence diagrams, and state machine diagrams, have already been explained, but since class diagrams are introduced here for the first time, we will explain them in some detail. The graph structure of a class diagram is as follows:

| Class diagram | |
|---|---|
| **Mode:** | Static |
| **Object:** | Things and relationships between them |
| **Vertex:** | Class |
| **Edge:** | Generalization, aggregation, association, dependency |
| **Subgraph:** | Not applicable |

A concrete example is shown in Figure 7.2.

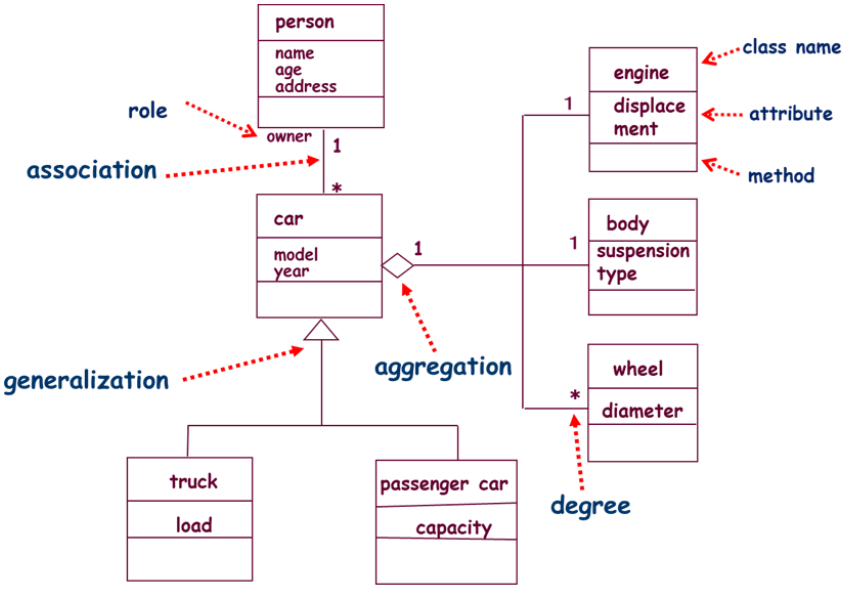


Figure 7.2: Car class diagram

A class is represented by a rectangular box with three vertical sections inside. The top section contains the class name, the middle section contains a list of attributes, and the bottom section contains a list of operations (methods). Attributes and operations may be omitted.

A generalization relationship is represented by placing a triangle at the end of the edge on the parent class side, with one vertex pointing to the boundary of the parent class box. An aggregation relationship is represented by placing a diamond that touches one of the edges of the aggregating class box, A general relationship is represented by an undirected edge, but when the way of reference to trace the relationship is limited to one direction, i.e., A and B are related and the reference is always from A to B, an arrow is placed on the side of the referenced B. Although it does not appear in the figure 7.2, if A depends on B, an arrow from A to B is drawn with a dotted line. However, if the dependency is an interface or abstract class, a triangle is drawn instead of an arrow, just like with generalization.

Multiplicity may be indicated for aggregation and association. In figure 7.2, 1 and * indicate multiplicity. If A and B are related or aggregated, for example, if 1 is displayed on the A side, A will participate in one relationship with exactly one instance, and if *, any number greater than 0 will participate. Also, if you write $m : n$, it means that the value is in the range between $m$ and $n$.

One of the characteristics of the graph structure of a class diagram is that it does not have a natural recursive structure in which expanding the inside of a class results in another class diagram. If the relationship between classes is limited to generalization or aggregation, the subgraph can be contraced to a class, but a class diagram that includes general assorications cannot be considered as a single class. However, if a package (which will be discussed later) that collects classes is considered as a class that has public operations (methods), then it can be said to have a recursive relationship. In that sense, a package diagram can be said to be a diagram with a recursive structure.

# Chapter 8

# Formal Methods

All of the models introduced in the previous chapters were described using diagrams. Although the meaning of diagrams is intuitively easy to understand, they inevitably lack logical rigor.

In this chapter, we will discuss how to give rigor to the description of models by using a description format with formality similar to that used in mathematics or logic.

## 8.1 Meaning of Formal Methods

One method for modeling is abstraction through formalization, as used in mathematics or logic. Form literally means ”shape,” and does not matter about the content. Natsume Soseki had a good understanding of formal logic in this sense. In the tenth chapter of ”I Am a Cat,” there is the following passage:

> ”I think that the relationship between these two is like a noun in a proposition of formal logic, connected regardless of its content.”

Formal logic, also known as mathematical logic or symbolic logic, is a formalization of logic since Aristotle using symbols. It formally defines the grammar and inference rules that make up logical expressions using symbols, and as Soseki said, there is arbitrariness in connecting the symbols to what in the real world. The ”form” in formal methods for software development has almost the same meaning as the form of formal logic.

The significance of using formal methods in software development is as follows:

1. Apply it to specification descriptions to create strict, unambiguous specifications.
2. Verify the correctness of the described specifications and models, and the properties they possess.
3. Verify that the implemented program satisfies the specifications.
4. Synthesize programs systematically or automatically from formal specifications.

If these goals could be realized, formal methods would be a powerful tool for software engineering, and so formal methods have been studied for a long time. However, compared to their long history, formal methods have not yet reached a sufficient stage of penetration into the industrial world. Formal methods have been used from early on to design logic circuits and communication protocols, where their effects are quite clear, but formal methods have not been widely used in general software, where the state space is much larger, and formal methods are not so easy to apply, so their penetration has been limited. Relatively advanced applications are fields that require high reliability, such as space equipment, aircraft, and nuclear reactor control. However, much of today’s software requires high security and confidentiality. Following the success of the aerospace and nuclear reactor control fields, the scope of application has expanded. In addition, for those that are widely and repeatedly used, such as middleware, libraries, components, and services, that is, things that are highly reusable and standardized, the cost of using formal methods pays off even if it costs a little.

There is also a valid criticism that ”human thinking is not formal.” However, since formal methods are also invented by humans, it can be said that humans can think both formally and informally. This ability to switch flexibly is one of the great things about the human brain. When people read formal descriptions, they do not necessarily follow formal reasoning, and even if they do take the trouble to follow it, they do not feel like they ”understand” it. People understand formal descriptions by relating them to some kind of model in their own mind.

Even if people do not usually think formally, they sometimes formalize things because it saves them thought. In such cases, value is created in the ability to move freely between formal and informal. However, when facing a real problem and finding a formalization that fits the problem, not only do they often find it easy to reach the solution, but they can also be moved by the beauty of the form itself.

Below, we will look at the various techniques that formal methods use to achieve each of their goals.

## 8.2 Formal Specification Description

Efforts to expand the scope of formal specification techniques to include practical system development are particularly active in Europe.

From the standpoint of industrial application, emphasis is placed on strict specification description, and proof of correctness of developed software and automatic program generation are not necessarily the main focus.

The reason why such activity is so active in Europe is, of course, the strength of the tradition of mathematics and logic, but it is also thought that the Esprit project, which was promoted by the EC (now the EU) in the 1980s, is a major influence. In the 1980s in Japan, a national project for the ”fifth generation computer” was promoted with a 10-year plan. The project is commonly known as the ICOT project, named after the organization that promoted it, the Institute for Future Computer Technology. Inspired by this, Esprit was born in Europe and its target field was broader than ICOT, including LSI development technology and software engineering. In addition, the development system is intended to link industry and academia, as well as to span the countries of the EC. As a result, attempts have been made to apply formal methods such as Z, which was originally developed in universities, to industry. Several formal specification languages have been proposed, but as we will see below, they are not just specification languages, but software development methodologies using them have been proposed and put into practice.

### 8.2.1 VDM

The fact that the name VDM[73] comes from the Vienna Development Method is now rarely mentioned, but it originally descended from the Vienna Definition Language, which provides the operational semantics of programming languages. Today, it has completely moved away from the framework of operational semantics, and instead, while being influenced by denotational semantics, it has become a general term for a specification description notation based on predicate logic expressions and program development techniques based on it. There is a British school centered around Cliff Jones, and a Danish school centered around Dines Bjørner, and the notations are somewhat different between the two. Support tools for the VDM-SL specification language based on VDM have also been developed and licensed, and there are known examples of its use in practical system development[47].

### 8.2.2 Z

Z[123] is a formal specification language based on set theory developed primarily at Oxford University in the UK. It was influenced by VDM and has many similarities to it, but it has been devised to be more concise in notation. Examples of practical use include the case where the entire specification of IBM’s data communication management system CICS was written using this, which improved maintainability and achieved groundbreaking results in reducing errors in new versions, and the case where a formal specification for floating-point arithmetic was given and applied to the development of actual arithmetic elements[107, 58]. The former is known for the scale of the project, in which a 2000-page specification was written in Z and a 50,000-line program was developed based on it, and for the data that showed that reliability improved several times and costs improved by 9% when Z was used and not used. The latter became famous when the Queen of England awarded it for its results.

### 8.2.3 RAISE

RAISE (Rigorous Approach to Industrial Software Engineering)[26] was developed as one of the European Esprit projects, and was led by Bjørner, who led VDM in Europe, and also placed emphasis on mathematical formalism. It defined RSL as a specification description language and developed a software development methodology using it. RSL differs from VDM and others in that it has not only declarative descriptions, but also basic elements for imperative descriptions and concurrent operation descriptions.

Bj$\phi$rner was a professor at the Technical University of Denmark, but in 1992 he was involved in the establishment of the United Nations University International Institute for Software Technology (UNU-IIST) in Macau, and became its first director. He then applied formal methods using RAISE to the development of railway computer systems in mainland China[15]. He then returned to Denmark and published a massive three-volume book on software engineering centered around RSL descriptions[17, 18, 16]

### 8.2.4 B and Event-B

Jean-Raymond Abrial, who had a major influence on the development of Z, created a formal specification description language called B[2] separately from Z. Furthermore, Event-B explicitly adds a method for describing events to B[3]. The development methodology using B and Event-B is characterized by the fact that proper proof is performed at each step of the gradual refinement from the beginning, and there are known cases where it has actually been applied to the development of the Paris subway system and the automatic bus system.

### 8.2.5 Algebraic specification description language

There is another lineage, algebraic specification description language. The OBJ series started by Joseph Goguen is a representative example, and among them, CafeOBJ by Niki Kokichi and others in Japan has a high degree of completion of the language and processing system[41]. In parallel, Donald Sannella and Andrzej Tarlecki in Europe designed a specification language called Casl (the Common Algebraic Specification Language) and created a processing system for it[118], and at SRI in the United States, Jose Meseguer and others developed an algebraic specification language called Maude and an excellent processing system[86].

### 8.2.6 Lotos and Estell

Languages for describing the specifications of parallel operating systems, particularly communication protocols, include Lotos and Estelle, and systems have also been developed to generate programs from these specifications. In general, specifications and development of parallel systems are prone to errors, so formal development techniques are likely to be effective. Formal methods were introduced relatively early on, particularly in the specification descriptions of standard protocols and languages.

### 8.2.7 Alloy

Alloy is a language for describing software models invented by Daniel Jackson. It is based on relational logic and, like Z, is based on set theory, but the notation has been improved to make it lightweight, and an analysis tool called Alloy Analyzer has been developed and is in practical use. Inspired by model checking systems (described later), Alloy Analyzer can automatically verify various properties of a model by limiting the target to a small, finite set. [67]. Daniel Jackson is the son of Michael Jackson.

### 8.2.8 Property-oriented and Model-oriented

There is a school of thought that classifies the methods used for the above formalization into property-oriented and model-oriented. Property-oriented is an axiomatic way of defining the properties of the system to be specified from the outside, while model-oriented is a constructive way of defining the internal structure that forms the properties. A representative property-oriented approach is a method of semantic definition such as CafeOBJ, which uses algebraic specifications, while a representative model-oriented approach is a method such as Z or VDM, which defines the specifications of a state machine by the data structure that constitutes its state and the input/output specifications of the operations on it.

For example, when defining the specifications of a stack, an algebraic specification would show the basic properties by an axiom expressed in an equation that pushes and pops it back to the original, but in Z or VDM, it is common to define it constructively using sequences, for example.

## 8.3 Verification of Correctness

As introduced in Chapter 10 regarding testing techniques, E. W. Dijkstra said, ”Testing can show that a program has bugs, but it cannot show that it does not have bugs.” In that sense, testing cannot verify that a program is completely correct. If you want ”completeness,” you have no choice but to combine a formal specification with the formal semantics of the program and mathematically prove that the target program satisfies the specifications. Such research has been ongoing since the 1970s, and has been applied to some programs that require particularly high reliability. Although there are not many examples of its application to general programs, this technology has had a major impact on the design of programming languages and programming methodologies.

### 8.3.1 Method of Validity and Verification

The procedure for formally proving the validity of a program is roughly as follows.

1. Proof of validity
   - Describe the specifications in pre- and post-conditions.
   - Place invariant assertions for loops. You can insert any assertions you like.
   - Prove that the final condition of a program execution path that has pre- and post-conditions or assertions before and after it can be derived from the condition at the beginning of the path.

     In this case, Hoare-style axioms are used to determine the meaning of each element of the program (or you can interpret language elements as ”transforming” the program state expressed by predicates).
2. Proof of Termination
   In addition to proving that the result is correct, it is necessary to guarantee that the program terminates correctly. Correctness that includes termination is called strong correctness. This is usually demonstrated by showing that some value (e.g., a non-negative integer) that has an order with finite descending chain property, i.e., an order that has the property that a monotonically decreasing sequence is always finite, decreases monotonically each time it goes around the loop.

The idea of pre/post conditions and invariants is also adopted in ”design by contract,” an object-oriented development methodology proposed by Bertrand Meyer.[88]When designing an object’s method, the pre/post conditions are first described and considered as a contract, and the design is carried out to abide by that contract. In addition, conditions that always hold for a class are described as invariants, and these are also considered as contracts in the class design. Users look at the contract and trust that the object will behave in accordance with it.

If preconditions are written as assertions that can be checked at run time, it is possible to check whether the assertions hold at run time, and if they do not, to issue a warning or interrupt the program, without using tools such as a theorem prover. In exchange, there is a restriction that general logical expressions cannot be written as preconditions, postconditions, and invariants, and only those that are determined as logical values at run time are valid.

In UML, OCL (Object Constraint Language)[139] has been introduced as a language for describing preconditions, postconditions, and invariants to support contract-based design, and it is recommended that it be used in combination with class diagrams.

If a formal specification can be created, it should be possible to mathematically verify whether a program developed according to it meets the specification. Various verification methods have been devised for this purpose, and the main early result is Floyd-Hoare logic.

### 8.3.2 Floyd-Hoare Logic

Floyd-Hoare logic was published independently by Tony Hoare and Robert W. Floyd about the same time, but because of the similarity of ideas, it was later called Floyd-Hoare logic[59, 48].

Floyd presented a logical verification method for flowcharts, while Hoare focused on Pascal programs. Therefore, this paper by Hoare, published in CACM in 1969, is monumental not only for presenting a method for verifying programs, but also for proposing an axiomatic semantics for programming languages (in this case, a sublanguage of Pascal).

Program code and preconditions/postconditions in Hoare logic are written as follows:

{P}C{Q}

Here, C is a piece of program code, P is a logical condition that must be true before C is executed, and Q is a logical condition that is true after C is executed and stopped. In the case of imperative languages, the semantics of assignment statements that assign values to program variables require special attention. Hoare wrote it as follows:

{P[f/x]} x:=f {P}

Here, P is any logical expression, and the precondition {P[f/x]} represents an expression in which all variables x appearing in P are replaced with the value f.

The next problem is how to handle loop (repetition) structures. Therefore, the concept of loop invariant assertions is introduced. A loop invariant assertion is a condition that must always be true when execution enters a loop. With this, the rules regarding loops are written as follows:

If $\{P \land B\}S\{P\}$ then {P} while B do S $\{\neg B \land P\}$

By giving such loop invariant assertions appropriately, and further giving preconditions and postconditions as specifications for the entire program, and repeatedly applying the axioms and inference rules defined for each program statement, the program code will eventually disappear and only logical formulas will remain. If this can be proven, it will be verified that the target program satisfies the specifications.

### 8.3.3 Theorem Provers

If the proof of correctness is done manually, it is not possible to guarantee that the proof is error-free.

Therefore, various theorem provers have been created and tried out.

The proof itself using a theorem prover cannot be fully automated. Therefore, a person must take the time to create a proof strategy and introduce the necessary lemmas. In addition, if the proof fails, there is also the problem that it is not immediately clear whether the problem is with the program, the specifications, the proof strategy, or the capabilities of the proof system.

However, theorem provers have been developed for a long time in the field of artificial intelligence, and their capabilities have been steadily improved throughout their history. Below, we will give an overview of the main theorem provers that are particularly related to program verification.

### 8.3.4 LCF

LCF (Logic for Computerable Functions) is a theorem prover developed by Robin Milner in the early 1970s[54]. It is an interactive theorem prover, and the programming language ML (Meta Language), which was developed at that time to allow users to describe their theorem proving strategies, is still used today as a general functional language, along with its derivative language OCaml.

### 8.3.5 Isabelle/HOL

Isabelle/HOL is a theorem prover that has been developed since the 1980s, succeeding LCF[98]. It is released as free software and has been widely used, so there is a rich library of proofs accumulated.

### 8.3.6 Coq

Coq is an interactive theorem prover that has been under development since the 1980s, just like Isabelle/HOL. It is also positioned as a proof assistant. The core of its development is a group of people including Thierry Coquand and Gérard Huet at INRIA in France. The main development language is OCaml, which also ties it to the lineage of LCF.

### 8.3.7 ACL2

Robert Boyer and Strother Moore at the University of Texas at Irvine developed a unique theorem prover using Lisp in the 1970s. This later became a new system called Nqthm, and is now released as open source software called ACL2 [76]. It is characterized by being an automatic proof system for inductive logic, and is mainly used for verifying software and hardware. It is developed in Common Lisp.

### 8.3.8 PVS

PVS (Prototype Verification System) is a theorem proving system developed by John Rushby of SRI in the United States, specifically for verifying computer programs [104]. It is developed in Common Lisp and is provided under the GPL (GNU license).

## 8.4 Model Checking

Compared to the verification technique using theorem proving as described in the previous section, model checking is a formal method, but has quite different characteristics.

Simply put, model checking is a method to automatically check whether a model that expresses the specifications of a system satisfies the properties required of the system. In particular, for reactive systems that include concurrency, models are described in Labeled Transition Systems (LTS) and properties are expressed in temporal logic, which are often referred to as model checking in the narrow sense[35]. Here, a labeled transition system can basically be considered the same as the state machine introduced in the 6.4 section, but the events that are attached to each transition and cause the transition are called labels. In addition, since the aim is to model reactive systems in particular, the existence of a final state is not assumed. A structure in which a subset of atomic propositions is attached as a label to vertices rather than edges is also often used in model checking techniques, and this is called a Kripke structure.

Model checking was proposed in the early 1980s and was used to verify the correctness of hardware logic circuits and communication protocols, but it has since come to be used to verify models of general software.

Compared to verification by theorem proving, there are the following differences.

1. The subject of verification is a model of the program, not the program itself. However, there are various methods for applying model checking to programs in combination with techniques for automatically generating models from programs, such as Java Pathfinder[138].

2. The specifications to be verified are generally not the complete specifications of a program, but are an extraction of some properties of the specifications related to safety and liveness.

3. In the case of theorem proving systems, humans are generally involved in the proof work, such as providing lemmas for the proof, but in the case of model checking systems, if the system is properly configured, the work is performed automatically and the results are obtained.

4. If verification fails, a counterexample that does not satisfy the specifications is output, which can be used as a reference for error correction.

### 8.4.1 Temporal Logic

Temporal logic, which describes the properties to be verified, is a general propositional logic with the addition of time-related modal operators such as ”always” and ”sometime”. There are various types of temporal logic, but the most widely used are linear temporal logic (LTL) and computation tree logic (CTL).

LTL was first proposed by Amir Pnueli in 1977. This was the beginning of the development of temporal logic for computer science. Meanwhile, CTL was proposed by Edmund Clarke and Allen Emerson at Carnegie Mellon University (CMU) in the early 1980s, and software that automatically performs model checking was developed based on it. This is the origin of the current model checking system NuSMV[34]. The forerunner of LTL-based model checking systems was

SPIN, developed by a group including Gerard J. Holzmann at Bell Labs. Development of SPIN also began in the early 1980s, but it was not until 1991 that it became available to the public for free[61].

LTL and CTL were invented independently, and their processing systems were developed separately, but later Emerson et al. proposed CTL$^*$ as a temporal logic system that encompasses both, and clarified the relationship between LTL and CTL. This relationship is shown in Figure8.1.

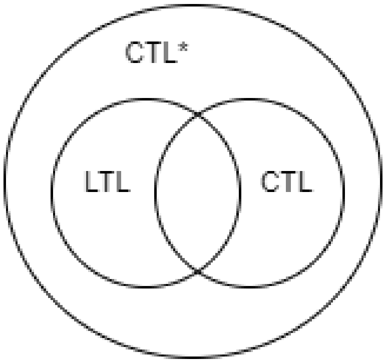


Figure 8.1: CTL, LTL and CTL$^*$

LTL and CTL are not in a relationship where one encompasses the other, and there are expressions that can be expressed in LTL but not CTL, and conversely, expressions that can be expressed in CTL but not LTL. However, CTL$^*$ encompasses both LTL and CTL, and there are expressions that can be expressed in CTL$^*$ but cannot be expressed in either LTL or CTL.

CTL$^*$ uses the following five types of temporal operators.

1. **X** p: Means ”next”. This indicates that p will hold in the next state.
2. **F** p: Means ”sometime”. This indicates that p will hold sometime in the future state transition process.
3. **G** p: Means ”always”. This indicates that p will always hold in the future state transition process.
4. p **U** q: Means ”until”. This indicates that p will hold until q holds in the future state transition process.
5. p **R** q: Means release. This indicates that q will hold until p holds for the first time in the future state transition process. q is released only when p holds, but it is not required that p will always hold at sometime.

Apart from this, there are two other types of path quantifiers related to branching in computation.

1. **E** f: There is a path from the current state in which f holds.
2. **A** f: f is true for all paths from the current state.

Note that the operators $\neg, \vee, and \wedge$ in normal propositional logic are also used. These logical operators are not all independent, and five operators $\vee$, $\neg$, **X**, **U**, and **E** are sufficient, and the rest can be expressed by combinations of these.

CTL adds the following constraint to CTL$^*$: ”When the temporal operators **X**, **F**, **G**, **U**, and **R** appear, they must immediately precede the path quantifier **E** or **A**.” LTL is a CTL$^*$ with the added restriction that ”only things of the form **A** f are allowed, where f does not include a path quantifier.”

### 8.4.2 Properties to Be Checked

Properties that are the subject of model checking are usually divided into the following two types.

- **Safety** The property that an undesirable event never occurs, no matter what transition path is followed. A typical example of an undesirable event is a deadlock. Another typical undesirable event in a concurrent system is a resource access violation (such as a violation of mutual exclusion). In temporal logic formulas, it is usually written in the following pattern:

$$\mathbf{AG}\ \neg P$$

- **Liveness** The property that a desirable event will always occur at some point. What is desirable depends on the system's functional requirements, but a typical property is that in a client/server system, a service request from a client will always be provided at some point. In temporal logic, this is usually described in the following pattern.

$$\mathbf{AF}\ Q$$

To get a more concrete understanding of safety and liveness, let's think about what properties are expected in the examples of the sake store problem and the vending machine problem we have used so far.

First, regarding safety, the vending machine problem description itself shows a certain consideration for safety.

- When the purchase button for a product whose sales lamp is on is pressed, the sales lamp for that product flashes and one product is dispensed into the dispenser. At the same time, the balance is deducted by the product price. **No money can be inserted during this time.**
- When the refund button is pressed, the remaining balance at that time is refunded into the change dispenser. **No money can be inserted during the refund operation.**

The bolded part above requests that the insertion of money be blocked during the process when the purchase button or the refund button is pressed. This is likely to be to prevent inconsistencies in the management of the inserted amount due to simultaneous operations of decreasing and increasing the amount. This problem description reads as a request to physically block the insertion of money, but it is possible to realize mutual exclusion of the amount data in software so that the insertion of money does not appear to be blocked to the user. In addition, if a specific blocking is realized, care must be taken to ensure that deadlock does not occur and this must be properly checked.

Regarding liveness, if we consider the requirement in the sake store problem that a customer's order will always be shipped at some point, this is an example of liveness. To satisfy this, it is not enough to simply devise a system design; it is necessary to assume a business condition outside the system, that there will always be a stock of liquor to satisfy the order. It is surprisingly difficult to accurately describe this condition and design the logic accordingly. For example, suppose you assume that when an order is placed for an item that is out of stock, there will always be a cumulative delivery of the ordered item that exceeds the shortage. In response to this, if shipping is based on a first-in, first-out strategy, meaning that earlier orders are shipped first, can you guarantee that the order will always be shipped at some point? If you think about it, there are cases where this does not work. Even if there are cumulative deliveries that exceed the shortage, if a single delivery is always below the shortage, then the item will be overtaken by a later order of the same item in a smaller quantity. If this situation continues all the time, the customer will be kept waiting indefinitely. It seems like it would be good to set the condition that there will always be a single delivery of the item that exceeds the shortage, but then it could be argued that this condition is too strong.

### 8.4.3 Kripke Structure

The Kripke structure is a set of atomic propositions that are associated with each state of a state machine as a label. While states, which are the vertices of a graph, are labeled, transitions, which are the edges of the graph, are not labeled as in a normal state machine. Formally, the Kripke structure is represented by the following quadruple:

$$(S, S_0, R, L)$$

$S$ is a set of states, $S_0$ is a subset of $S$ and represents the initial state. $R$ is a subset of $S \times S$ and represents the transition relation. The relation is a global relation, that is, for any state $s \in S$, there must exist $s' \in S$ such that $R(s, s')$ holds.

When the set of atomic propositions is $AP$, the label $L$ is a function from $S$ to the power set of $AP$ (a set whose elements are subsets of $AP$), that is, $L : S \rightarrow 2^{AP}$.

Here is an example.

The Kripke structure diagram for a simplified example of a drink vending machine is shown in Figure 8.2.

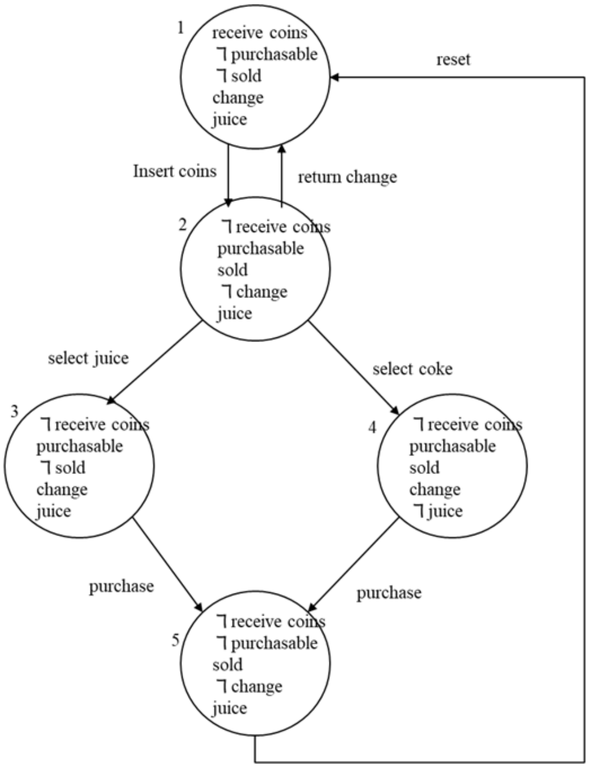


Figure 8.2: Kripke structure of a vending machine

Here, AP consists of the following five atomic propositions:
{Coins accepted, Available for purchase, Sold, Change available, Juice}
The meaning of each is as follows:

- Receive coins: Coins can be inserted
- Purchasable: Drinks can be purchased
- Sold: Drinks have been sold
- Change: Change can be returned
- Juice: Juice is selected as a drink that can be purchased. In the negative case, coka is selected.

Let's assume that the property to be verified is the following formula in CTL.
**AG** (Receive coins → **AF** Sold)

You can probably imagine from the diagram that whether this holds true or not can be determined by searching for a path on the Kripke structure. In fact, model checking is a method of mechanizing such path search to determine the answer. As is clear from this example, it is possible to loop infinitely between vertices 1 and 2 on the Kripke structure, that is, to repeat the action of inserting coins and getting change forever, so the above formula does not hold unless some fairness assumption is made. However, if we rewrite the property as follows, it will hold.

**AG** (Receive coins→ **AF***Change* ∨ *Sold*)

### 8.4.4 Expression of Boolean Expressions

As can be seen from the example of the Kripke structure, the evaluation of logical expressions in each state is essential for model checking, and how efficiently this is done determines the performance of the entire model checking. Basically, the propositional logical expression is evaluated according to the truth value of each element of the atomic proposition set, which reduces to the evaluation of Boolean functions as has been known for a long time. Boolean functions can be regarded as feature functions that define the set of variable vectors that make them true, so sets can be expressed by defining a method for expressing Boolean expressions.

### 8.4.5 Binary Decision Diagrams

CMV, which was developed at CMU in the 1980s, introduced a data structure called binary decision diagrams (BDDs) to reduce the amount of data and speed up the decision procedure. A Boolean function of $n$ variables can be calculated simply by creating a $n$-level binary tree. However, this involves a lot of structural duplication, which is wasteful in terms of both storage space and computational complexity. Binary decision diagrams are a creative way to remove this duplication. However, the size of a binary decision diagram varies depending on the order of the variables, and determining the optimal order is computationally difficult, so heuristic methods are used. However, a binary decision diagram that fixes the order of variables to a single order and fixes it regardless of the path from the root to the leaves of the tree gives a canonical system, so ordered binary decision diagrams (OBDDs) are used. The CMU group called this method of using OBDDs ”symbolic model checking,” in the sense that it performs model checking without expressing specific states.

### 8.4.6 Bounded Model Checking and SAT

The introduction of BDDs has dramatically increased execution speed, but it is still often unable to deal with state explosion, especially due to system concurrency. However, model checking is useful if it can find cases that do not satisfy the required properties, and it has been found that in practice, when a model has a defect, the computational path that leads to the defect is often relatively short. Based on this experience, a method was proposed to limit the length of the path to be searched to within a certain constant $k$, and it has come to be used in practice. This is called bounded model checking.

Since a set of states can be represented by using Boolean functions as characteristic functions, transitions from state to state can also be represented as Boolean functions. If the number of transitions is limited to a finite value $k$ or less, the question of whether the target Kripke structure satisfies a given temporal logic formula is reduced to the Boolean function satisfiability problem (SAT). The satisfiability problem has been tackled since the early days of the development of theorem provers in the field of artificial intelligence, and it has long been known to be NP-complete. The Davis–Putnam algorithm, published in 1960, is famous, and although it does not break through the NP-completeness barrier, it is known as a practical method. Since then, SAT solvers based on this algorithm have been developed and put to use, with various improvements. Bounded model checking techniques have also been made more effective by using existing SAT solvers.

Furthermore, the satisfiability problem has been extended to allow the appearance of predicates in logical formulas, and a method called ”satisfiability modulo theory” (SMT) has been proposed, which combines dedicated theories with these predicates. Various SMT solvers have been developed, one of the well-known is Microsoft’s Z3 Theorem Prover.

### 8.4.7 Abstraction

Another effective method to deal with state explosion is state space abstraction. There are two known methods for this. One is to consider a state space limited to only variables that affect the specifications, and the other is to abstract the data space. Here, we will explain the latter.

For example, if one of the elements that make up a state takes an integer value, the number of possible values becomes infinite. Mapping this to three levels, for example 0, positive, negative, is an example of abstraction.

The state space is reduced by this abstraction, and if the property of the object to be checked is expressed in that abstract space, model checking can be performed. Generally, properties that hold in an abstract space also hold in the original space, but even if they do not hold in the abstract space, they may hold in the original space. Therefore, even if a counterexample is found in an abstracted problem, it is necessary to further refine the abstraction and evaluate whether it is a meaningful counterexample. Such counterexamples in abstract spaces are called spurious counterexamples, and there are various research examples on how to deal with them [33].

## 8.5 Application Example—Description of the sake warehouse problem using Z

As an application example of formal methods, we will look at the description of the ”sake warehouse problem” using the specification description language Z.

As introduced in the 8.2.2 section, Z is a formal specification language developed at Oxford University, and its foundation is set theory, with all data being typed based on set theory. Schema is a framework for describing information units that consist of such collections of data, their properties, and the operations on them, and plays a central role in the Z language.

### 8.5.1 Types in Z

All objects (formulas) expressed in Z have types. Types can be considered as sets, and are either given in advance as basic sets, or have structures constructed from them. A typical basic set is the set of integers (written as $\mathbb{Z}$ in Z), but many are determined according to individual problem domains.

In our problem, “container number” and “item name” are given as basic data sets. We also use the name “quantity” as a natural number that does not include 0 (1 or more). This is written as follows:

$$[containernumber, itemname]$$
$$quantity == \mathbb{N}_1$$

In general,

$$[identifier, \ldots, identifier]$$

introduces the name of the basic type (set). In the following description of Z, the type introduced here will be used as given without going into its internal structure.

Also,

$$variablename == expression$$

can refer to the expression on the right hand side by the name on the left hand side.

Structural types constructed from basic types include power set types, Cartesian product types, and schema types, and concrete examples of these will appear in the following sections.

### 8.5.2 Schema

Now, containers are piled up in the warehouse. Let’s express this as a schema named warehouse.

*Warehouse*

$$Container : container\_number \nrightarrow (brand\_name \nrightarrow quantity)$$

A schema generally consists of a schema name, a declaration section, and an axiom section, and is enclosed in a rectangular frame with an open right side. The schema name is written inside the horizontal line at the top of the frame.

The declaration section and axiom section are written inside the rectangle, and a horizontal line is put between them, but this example consists only of the declaration section, and has no axiom section. This declaration section shows one variable "container" that represents the state of the warehouse, along with its type. In general, multiple variable declarations can be lined up in the declaration section of a schema. This schema can be considered a so-called record-type data declaration. However, since the type of a variable generally includes functions, it is closer to an object in the object-oriented approach.

The type of this container is exactly the function. The symbol $\pfun$ represents a partial function, so a container can be read as a partial function with the container number as its domain and $(\mathit{productname} \pfun \mathit{quantity})$ as its range. The range $(\mathit{productname} \pfun \mathit{quantity})$ itself is a partial function, with the product name as its domain and the quantity as its range. In other words, a container is a collection of product names with a set of quantities. The Z approach is to interpret such functions in terms of set theory. In other words, the type of a container as a set is

$$\mathit{container} \in \mathbb{P}(\mathit{containernumber} \times \mathbb{P}(\mathit{productname} \times \mathit{quantity}))$$

here, $\mathbb{P}$ is the operation that creates a power set. In other words, $\mathbb{P}\,X$ is the set of subsets of $X$. Also, $X \times Y$ is the operation that creates the Cartesian product of $X$ and $Y$. That is, an element of $X \times Y$ is an ordered pair $(x, y)$ of element $x$ of $X$ and element $y$ of $Y$.

Z is characterized by its way of treating functions and relations as sets. Functions from domain $X$ to $Y$ and relations between domains $X$ and $Y$ are all treated as subsets of the Cartesian product $X \times Y$, and their basic types are the same.

Functions are a type of relation, but they satisfy the constraint that for any element of the domain, there is at most one pair that contains it as a relation.

The type of relation between sets $X$ and $Y$ is written as $X \leftrightarrow Y$. In Z, this is defined as

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

This means that a relation $R$ has the type $X \leftrightarrow Y$, i.e., $R : X \leftrightarrow Y$ means that $R \in \mathbb{P}(X \times Y)$, that is, $R$ as a set has the form $\{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$, where $x_i \in X, y_i \in Y (i = 1, \ldots, m)$.

For a relation $X \leftrightarrow Y$, there are standardly defined functions dom and ran of type $(X \leftrightarrow Y) \rightarrow \mathbb{P}\,X$ and $(X \leftrightarrow Y) \rightarrow \mathbb{P}\,Y$, respectively.

For a relation $R$, $\mathrm{dom}\,R$ represents its domain, and $\mathrm{ran}\,R$ represents its range.

Treating relations and functions as sets in Z is not that difficult, but it is easy to confuse types and values, or the hierarchy of types. For example, if we consider a type $R$ such that $R == \mathbb{P}\,\mathbb{P}\,S$, the value of $x$ becomes one of the sets of a subset of $S$ by a declaration $x : R$ of a variable whose type is $R$, or a predicate $x \in R$ that says $x$ is an element of $R$. Furthermore, if we define $y \in x$, the value of $y$ becomes one of the subsets of $S$. Furthermore, if we define $z \in y$, the value of $z$ becomes an element of $S$. In this way, the left side of a type declaration: or predicate $\in$ is one level lower than the right side in the hierarchy created by the relation of sets of sets. On the other hand, the hierarchical level of a set can be increased not only by applying $\mathbb{P}$ directly, but also by taking relations and functions, as can be seen from the definition of relations, so you need to be careful about this until you get used to it.

The axioms part of a schema is used to provide the constraints that the variables defined in the declaration part must satisfy as axioms. In this example, if the expression in the problem statement "Up to 10 brands can be mixed in one container" is given as a constraint, the schema can be written as follows.

*Warehouse*

$$\mathit{container} : \mathit{container\_number} \pfun (\mathit{brand\_name} \pfun \mathit{quantity})$$

$$\forall\, c : \mathrm{dom}\,\mathit{container} \bullet \#(\mathit{container}\ c) \leq 10$$

Here, as already mentioned, dom is a function that takes a function (generally a relation) as an argument and gives its domain, and # is a function that takes a finite set as an argument and gives the number of elements. Both are defined as standard functions in Z.

The axioms section contains predicate logical formulas. The following logical operators are used:

| | |
|---|---|
| $\neg$ | negation |
| $\wedge$ | conjunction |
| $\vee$ | disjunction |
| $\Rightarrow$ | implication |
| $\forall$ | universal |
| $\exists$ | existence |

A logical formula for a universal binding is generally written as follows:

$$\forall\, \mathit{declaration}[;\ \mathit{declaration};\ \ldots] \bullet \mathit{logicalformula}$$

Here, a declaration is

$$\mathit{declaration} ::= \mathit{variablename}[, \mathit{variablename}, \ldots] : \mathit{type}$$

Here, a type is a formula that represents a set. The same applies to logical formulas for existence bindings.

Note that multiple logical formulas may be written side by side in the axioms section of a schema, separated by line breaks. In this case, the whole is considered to be a conjunction of the logical expressions on each line. In other words, a newline can be used instead of a conjunction symbol.

In Z, predicates are also interpreted in a set-theoretic way. In other words, if you write the predicate $P(x)$, $P$ represents a set, and its meaning is equivalent to $x \in P$. Therefore, new predicates are also defined by constructively giving a set.

### 8.5.3 General Definition

**Higher-order functions** The problem is the process of shipping according to customer orders. Customers order how many bottles of a certain name of sake. To fulfill this, it becomes necessary to search for containers of sake in the warehouse by name. For this purpose, let us consider the concept of sake inventory.

$$\mathit{sake\_inventory} : \mathit{brand\_name} \pfun (\mathit{container\_number} \pfun \mathit{quantity})$$

Add this to the warehouse declaration section. The type of the sake inventory can be expressed as a set type as follows:

$$\mathit{sake\_inventory} \in \mathbb{P}(\mathit{brand\_name} \times \mathbb{P}(\mathit{container\_number} \times \mathit{quantity}))$$

The point of this problem is that the container and the sake inventory are essentially the same thing. To do this, let's generalize a little. The types of containers and sake inventory are of the form $(X \pfun (Y \pfun Z))$, which can be converted to $((X \times Y) \pfun Z)$ in a one-to-one relationship. This conversion is the exact opposite of the currying operation in functional programming, which turns a multivariate function into a sequence of one-variable functions. So, let's define a bijective (higher-order) function *uncurry* that converts a function of type $(X \pfun (Y \pfun Z))$ to a function of type $((X \times Y) \pfun Z)$. When defining this function, the target types $X, Y, Z$ do not need to be specific and can generally be any type. Z provides a convenient description format for defining generic functions with types as parameters, called generic definitions. This is written as follows:

$$\begin{array}{|l}
\hline [X, Y, Z] \\
\mathit{uncurry} : (X \pfun (Y \pfun Z)) \bij ((X \times Y) \pfun Z) \\
\hline \forall f : (X \pfun (Y \pfun Z)) \bullet \\
\quad \mathit{uncurry}\, f = \\
\quad \{\, x : X;\ y : Y;\ z : Z \mid x \in \mathrm{dom} f \wedge y \in (f\, x) \wedge z = f\, x\, y \bullet (x, y) \mapsto z \,\} \\
\hline
\end{array}$$

This defines a global function *uncurry* with parameters X, Y, and Z of any type. The axioms below the middle line define the properties of the function. Here, $\bij$ represents a bijection.[1]. Also, $x \mapsto y$ represents a pair that maps the value $x$ to the value $y$ as an element of a function. In other words, if $x \mapsto y \in f$, then $f(x) = y$. When corresponding to the

[1] Bijection means that it is both surjective and injective. $f : X \pfun Y$ is surjective if its range covers the entire $Y$, i.e., if $\mathrm{ran} f = Y$, and is also called a mapping from $X$ onto $Y$. Injection is a so-called one-to-one mapping, and $f : X \pfun Y$ is injective if $\forall x_1, x_2 : \mathrm{dom} f \bullet f\, x_1 = f\, x_2 \Rightarrow x_1 = x_2$.

representation as a set, $x \mapsto y$ is actually the same as $(x, y)$. In Z, parentheses are not usually used unless they are needed to express function application, following the conventions of functional programming. Therefore, $f(x)$ is usually written as $f\ x$.

Set comprehension notation is used to define this function. Set comprehension notation generally has the following form:

$$\{D \mid P \bullet E\}$$

Here, $D$ is a variable declaration, $P$ is a logical expression that expresses a constraint, and $E$ is an expression. Of all the values that the variable declared in $D$ can take, those that satisfy the logical expression $P$ are selected, and the set of values obtained by substituting them for $E$ is represented by this. For example, $\{x : \mathbb{N}_1 \mid x \leq 5 \bullet x^2\}$ is equivalent to $\{1, 4, 9, 16, 25\}$.

Although $\bullet$ is also used as a delimiter in universal or existential binding formulas, care must be taken to avoid confusion with set comprehension notation, as appropriate parentheses may be required.

Note that $E$ can be omitted, in which case it represents the set of $D$ values that satisfy $P$. For example, $\{x : \mathbb{N}_1 \mid x \leq 5\}$ is equivalent to $\{1, 2, 3, 4, 5\}$. In practice, this form may be more commonly used.

Now that we have seen how to write predicate logical expressions and set comprehension notation, let's look at the definition of a function using both. As mentioned earlier, a function is a type of relation such that for any element of the domain, there is at most one pair that contains it. This is defined in Z as follows:

$$\begin{array}{l} X \mathrel{⇸} Y == \{f : X \leftrightarrow Y \mid (\forall x : X;\ y_1, y_2 : Y \bullet \\ \qquad\qquad (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2)\} \end{array}$$

(Note that the $\bullet$ in this formula is not the $\bullet$ that appears in set comprehension notation, but the $\bullet$ of a universal binding.)

Now that we have used *uncurry* to convert a container into a function of two arguments, we will next consider a conversion that swaps the first and second arguments.

$$\begin{array}{|l} [X, Y, Z] \\ \hline \mathit{swap} : ((X \times Y) \mathrel{⇸} Z) \mathrel{⤖} ((Y \times X) \mathrel{⇸} Z) \\ \hline \forall f : (X \times Y) \mathrel{⇸} Z \bullet \\ \qquad \mathit{swap}\, f = \{\ x : X;\ y : Y;\ z : Z \mid (x, y) \in \mathrm{dom} f \land z = f(x, y) \bullet (y, x) \mapsto z\ \} \\ \hline \end{array}$$

Finally, the series of transformations can be expressed as $\mathit{uncurry}^{\sim} \circ \mathit{swap} \circ \mathit{uncurry}$. Here, $\circ$ represents the composition of functions, and $\mathit{uncurry}^{\sim}$ represents the inverse function of *uncurry* (inverse functions can be defined for injective functions). Since it is natural to name this inverse function *curry*, let us set it as follows.

$$\mathit{curry} == \mathit{uncurry}^{\sim}$$

Using these, we can rewrite the warehouse schema as follows.

$$\begin{array}{|l} \mathit{Warehouse} \\ \hline \mathit{container} : \mathit{container\_number} \mathrel{⇸} (\mathit{brand\_name} \mathrel{⇸} \mathit{quantity}) \\ \mathit{sake\_inventory} : \mathit{brand\_name} \mathrel{⇸} (\mathit{container\_number} \mathrel{⇸} \mathit{quantity}) \\ \hline \mathit{sake\_inventory} = \mathit{curry} \circ \mathit{swap} \circ \mathit{uncurry}\ \mathit{container} \\ \hline \end{array}$$

**multiset** When a request for shipment is made, it is necessary to determine whether there is enough stock of the specified brand of sake to fulfill the order. In addition, if a container is empty, there is a request to remove the container, so the total amount of alcohol currently in the container may also be required. The function that gives these will be in the form:

$$\begin{array}{l} \mathit{brand\_inventory} : \mathit{brand\_name} \mathrel{⇸} \mathit{quantity} \\ \mathit{container\_inventory} : \mathit{container\_number} \mathrel{⇸} \mathit{quantity} \end{array}$$

Let's write out the specifications for these two functions.

Comparing these functions with the definitions of sakestock and container, the relationship between them is clear. Brand stock quantity, like sake stock, has the product name as its domain, and is the sum of the quantities for the mapping destination $containernumber \pfun quantity$. Similarly, container stock quantity can be considered as the sum of the quantities for the elements of the container range. This can be described formally.

So, for example, for the elements of $brandname \pfun quantity$, or equivalently, $\mathbb{P}(brandname \times quantity)$, we first consider the operation of collecting the quantities, which are the second components of each, and then take the sum. It seems that we can do this by using the function ran, which is generally defined for relations, but the problem is that it is inconvenient to consider the collection of second components as a set. A set does not consider the same elements to be duplicated, but in this case, even if the same quantity of values appear multiple times, they must be treated as separate items. Such a collection is called a multiset (bag or multiset).

A multiset consisting of elements $a_1, \ldots, a_n$ is written as $\llbracket a_1, \ldots, a_n \rrbracket$ in Z. For example, $\llbracket 1, 2, 2, 3 \rrbracket$ is different from $\llbracket 1, 2, 3 \rrbracket$ (but since the order does not matter, $\llbracket 1, 2, 2, 3 \rrbracket$ is the same as, for example, $\llbracket 1, 2, 3, 2 \rrbracket$). Z defines a multiset as a function from the elements to natural numbers greater than or equal to 1. The natural number to which it is mapped represents the number of occurrences in the multiset. In other words, a multiset $\operatorname{bag} X$ of type X is generally defined as

$$\operatorname{bag} X == X \pfun \mathbb{N}_1$$

. For example, $\llbracket 1, 2, 2, 3 \rrbracket$ is an element of $\operatorname{bag} \mathbb{N}$, but by definition it is represented as $\{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 1\}$.

We want a multiset version of dom and ran defined in Z for relations. Let's call them *bdom* and *bran*, and define them as follows:

$$
\begin{array}{|l}
\hline
[X, Y] \\
bdom : (X \leftrightarrow Y) \to \operatorname{bag} X \\
bran : (X \leftrightarrow Y) \to \operatorname{bag} Y \\
\hline
\forall R : X \leftrightarrow Y \bullet \\
\qquad bdom\, R = \{\, x : \operatorname{dom} R \bullet x \mapsto \#\{\, y : Y \mid x \underline{R}\, y \} \,\} \land \\
\qquad bran\, R = \{\, y : \operatorname{ran} R \bullet y \mapsto \#\{\, x : X \mid x \underline{R}\, y \} \,\} \\
\hline
\end{array}
$$

Next, let's define a function $\Sigma$ that sums up the elements of a multiset of natural numbers $\mathbb{Z}$.

$$
\begin{array}{|l}
\Sigma : \operatorname{bag} \mathbb{Z} \to \mathbb{Z} \\
\hline
\Sigma \llbracket\rrbracket = 0 \\
\forall x : \mathbb{Z} \bullet \Sigma \llbracket x \rrbracket = x \\
\forall B, C : \operatorname{bag} \mathbb{Z} \bullet \Sigma(B \uplus C) = \Sigma B + \Sigma C
\end{array}
$$

The notation used here does not have a schema outer frame, but is called an axiomatic statement, and declares global variables and gives their properties as axioms. Variables declared in a schema cannot be referenced without quoting the schema name, but variables declared in an axiomatic statement are global and can be referenced at any time. In that respect, they are similar to variables declared in a generic definition, but the difference is that a generic definition defines parameterized global variables (usually functions), while an axiomatic statement defines variables (functions) as constants.

Note that $\llbracket\rrbracket$ represents the empty multiset, and the symbol $\uplus$ represents the operation of taking the union of two multisets.

This function like $\Sigma$ can be generalized and defined for multisets $\operatorname{bag} X$ on a set $X$ that has an identity and has a commutative and associative binary operation defined

However, we will not generalize that much here and instead define it for addition over natural numbers.

Based on the above preparations, if we define the transformation to obtain brand inventory and container inventory from the sake inventory and container respectively as a (high-order) function called *subtotal*,

$$
\begin{array}{|l}
\hline
[X, Y] \\
subtotal : (X \pfun (Y \pfun \mathbb{Z})) \to (X \to \mathbb{Z}) \\
\hline
\forall f : X \pfun (Y \pfun \mathbb{Z}) \bullet subtotal\, f = \{\, x : dom f \bullet x \mapsto \Sigma \circ bran \circ f\ x \,\} \\
\hline
\end{array}
$$

Using this,

$$\begin{array}{l} \mathit{brand\_inventory} == \mathit{subtotal}\ \mathit{sake\_inventory} \\ \mathit{container\_inventory} == \mathit{subtotal}\ \mathit{container} \end{array}$$

The warehouse schema is expanded and redefined to include these two functions.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Warehouse}} \\ \hline
\mathit{container}\ \mathit{container\_number} \nrightarrow (\mathit{brand\_name} \nrightarrow \mathit{quantity}) \\
\mathit{sake\_inventory} : \mathit{brand\_name} \nrightarrow (\mathit{container\_number} \nrightarrow \mathit{quantity}) \\
\mathit{brand\_inventory} : \mathit{brand\_name} \nrightarrow \mathit{quantity} \\
\mathit{container\_inventory} : \mathit{container\_number} \nrightarrow \mathit{quantity} \\ \hline
\mathit{sake\_inventory} = \mathit{curry} \circ \mathit{swap} \circ \mathit{uncurry}\ \mathit{container} \\
\mathit{brand\_inventory} == \mathit{subtotal}\ \mathit{sake\_inventory} \\
\mathit{container\_inventory} == \mathit{subtotal}\ \mathit{container} \\ \hline
\end{array}$$

### 8.5.4 Abstract Machine

Up until now, descriptions using Z have mainly captured the static aspects of the problem domain in question. The method used was to use abstract functions to capture the logical structure of the object as concisely as possible. However, one of the major features of Z is that it regards the object as an abstract machine with a state and describes its dynamic behavior. Schemas are often used for this purpose.

**Operation Specification** For example, the warehouse schema has been treated as a definition of the space of possible values for a data type with components of container and sake inventory. If we consider this as an abstract machine, we can imagine a machine (or process, or system) that has one of these values as its state at any given time. The machine changes its state depending on the operation it receives from the outside. To express this change in state, we need a mechanism to describe the relationship between the state before and after the operation.

For example, let's consider the process of receiving containers.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Receiving}} \\ \hline
\mathit{Warehouse} \\
\mathit{Warehouse}' \\
\mathit{cn}? : \mathit{container}_n\mathit{umber};\ \ s? : \mathit{brand\_name} \nrightarrow \mathit{quantity} \\ \hline
\mathit{cn}? \notin \operatorname{dom} \mathit{container} \\
\mathit{container}' = \mathit{container} \cup \{\mathit{cn}? \mapsto s?\} \\ \hline
\end{array}$$

In this schema definition, important notations are used: quoting and qualifying other schemas. First, the ”warehouse” schema is quoted here as a schema reference. In this case, the declaration part of the quoted schema is added as is to the declaration part of the ”warehouse” schema being defined here, and its axiom part is added as is to the axiom part of ”warehouse”. On the other hand, the notation ”$\mathit{Warehouse}'$” is a qualified schema reference, which means that all variables declared in the warehouse declaration part (container, liquor stock, container stock amount, brand stock amount) are appended with ”$'$”, and all of those variables appearing in the axiom part are replaced with ”$'$”. These variables with ”$'$” are usually interpreted as indicating the state of the schema representing the ”warehouse” operation being defined here after the operation is completed. In a schema that defines the operation of a state machine, it is common to reference both a schema that represents a state and a schema with ”$'$” attached to it, so a notation that expresses them at the same time is also defined. To do this, $\Delta$ (uppercase Greek letter delta) is added to the beginning of the schema name to be referenced. In other words, in the above example, it can be written as follows:

*Inventory*

$\Delta Warehouse$
$cn? : container\_number;\ s? : product\_name \pfun quantity$

$cn? \notin \operatorname{dom} container$
$container' = container \cup \{cn? \mapsto s?\}$

**Input/output conditions** This schema also introduces two variables, $cn?$ and $s?$. Variables with a ? at the end like this are interpreted as variables that represent input data for this operation. Also, although it does not appear in this example, it is a rule to put a ! at the end of variables that represent output. However, these are merely a convention for the convenience of interpreting that they represent the operation of a state machine, and in logical formulas, they are just variables that are exactly the same as normal variables.

The axiom part of the schema of such an operation represents the input and output conditions of the operation. In many formal specification description languages other than Z (for example, VDM), input and output conditions are written syntactically distinct, but Z does not make such a distinction. However, among the logical formulas that make up the axiom part (which are interpreted as a conjunction of them as a whole), those that do not contain any variables with ”$'$” or ”!” as a suffix can be interpreted as input conditions, and those that contain at least one of them as output conditions, so we have taken the policy of not introducing syntactic elements that distinguish between input and output conditions.

This ”$'$” notation is powerful, and in particular, the axioms of the quoted schema are automatically included, so there is no need to explicitly write conditions that are invariant to the operation. In the above example, we only write the output conditions for ”container” and not for ”liquor stock”, because the qualified schema is quoted, and the following is implicit (added to the axioms of the inventory schema). This method is unique to Z and convenient, but it can sometimes make the operation specifications difficult to understand.

**Reference type operations** Some operations can refer to a state without changing it. In this case, if you prefix the schema you want to reference with $\Xi$ (Greek capital letter Xi), not only will you cite the schema and the schema with ”$'$” at the same time, but the axioms section will also implicitly add an equation $x = x'$ for all variables $x$ declared in the schema.

For example, let’s define a reference operation that specifies a brand and checks the stock amount using the already defined function ”stock amount”.

*InventoryCheck*

$\Xi warehouse$
$b? : product\_name, v! : quantity$

$v! = brand\_inventory\ b?$

**initialization** If it is a state machine, the initial state must be determined in some way. This can also be defined as a single schema. For example, for the warehouse,

*InitialWarehouse*

$Warehouse$

$container = \varnothing$

If we were to define this as an initialization operation, we would use $\Delta$ warehouse and set $container' = \varnothing$, but the above definition is a declarative description of the warehouse’s initial state.

**error handling** In the ”warehouse” operation, the input condition is that the container number of the container to be stored does not match the container number of the container already in the warehouse ($cn? \notin \operatorname{dom} container$). How should we describe exceptions that do not satisfy this condition as specifications? In Z, the usual style is to describe such exception handling specifications separately as a separate schema.

First, we define a type consisting of symbols to identify errors, the ”error condition”.

$$\mathit{errorcondition} ::= \mathit{normal} \mid \mathit{duplicatecontainernumber}$$

Here, $::=$ is a free-form declarator that declares a type consisting of the elements enumerated on the right-hand side, separated by $|$. In this example, it is a simple enumeration type, but recursive definitions are possible, which allows the introduction of recursively defined types such as trees.

Next, we define the schema that represents the normal case as ”successful completion”.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{successfulcompletion}} \\ \hline
\mathit{Result}! : \mathit{errorcondition} \\ \hline
\mathit{Result}! = \mathit{normal} \\ \hline
\end{array}$$

This may seem like a boring schema, but it can become useful by using a mechanism called schema expressions. Schema expressions are operations that connect schemas together, and there are several types of operators, but here we will only list the logical schema operators $\wedge$ and $\vee$. Both take two schemas and construct a new schema. In both cases, the declarations of the two schemas are merged. In this case, if there are variables with the same name in both schemas, the common part of the types (sets) of the variables in each schema is taken, and a single variable with this type is declared in the resulting schema. In addition, the axiom part is the conjunction of the axiom parts of the two schemas in the case of $\wedge$, and the disjunction of the axiom parts of the two schemas in the case of $\vee$.

Therefore, if you set it to ”warehouse $\wedge$ normal completion”, the function of returning a normal result to the warehouse defined above will be added.

Next, the case where a container number overlap occurs can be described with the following schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DuplicateNumber}} \\ \hline
\Xi \mathit{Warehouse} \\
\mathit{cn}? : \mathit{container\_number} \\
\mathit{result}! : \mathit{errorcondition} \\ \hline
\mathit{cn}? \in \operatorname{dom} \mathit{container} \\
\mathit{result}! = \mathit{duplicatenumber} \\ \hline
\end{array}$$

Using this, the warehouse operation including exception conditions can be written as follows:

$$\mathit{GeneralizedWarehouse} \mathrel{\widehat{=}} (\mathit{Warehouse} \wedge \mathit{NormalCompletion}) \vee \mathit{DuplicateNumber}$$

Here, $\widehat{=}$ defines the left-hand side schema with the right-hand side schema expression.

After this, we need to deal with the ”unloading process” to complete the specification, but this is somewhat cumbersome, and the essential part of the specification has already been completed, so we will leave it in a different accessible location. In addition, we will also add an application example of ”bridge signal control” as a different type of application example using EventB, so interested readers should refer to the ”Formal Method Application Example Addendum” [**?**] .

# Chapter 9

# Design

Design is the center of engineering. The modeling techniques we have seen in the previous chapters can be used to clarify the ”requirements” of what should be made by constructing a model of the target problem domain, and can also be used to provide a basis for the ”design” of how it should be made. In other words, it is important to take a consistent process of developing the model from the model created in the requirements analysis to the design and implementation.

## 9.1 Architecture Design

The job of design is to accurately grasp the requirements to be met and the problems to be solved on the domain model clarified by the analysis, and to construct the structure of the system that will be the solution. This work involves a part that determines the overall structure, that is, the architecture, and a part that designs more specific algorithms and data structures.

### 9.1.1 What is Architecture?

Architecture of course, denotes the work of constructing houses and buildings. When used as a design concept in engineering fields other than architecture, it is used to mean something like a basic design structure.

It has been used for a relatively long time in othrer engineering disciplines, especially in the computer field. For example, there are the following two.

**Computer architecture** It has been established for a long time as a term to describe the basic structure of computer hardware. In a narrow sense, it means the instruction set. This is probably because the basic design philosophy is clearly reflected in the instruction set.

**Network architecture** The protocol layer model of a network is called a network architecture. A typical example is the ISO/OSI model.

The expression software architecture also has a long history. For example, the term software architecture appears frequently in F. Brooks' *The Mythical Man-Month* [27], published in 1975. Its meaning is close to requirements specifications, and although it is slightly different from current usage, it does not feel strange in the sense that basic requirements specifications determine the basic structure of software. It can be said to have a similar relationship to how the instruction set determines the basic structure of a computer.

However, it was in the late 1990s that the term architecture made a spectacular comeback in the field of software engineering. As mentioned in Chapter 2, research on software process became popular from the late 1980s to the early 1990s, but as mentioned there, the history of software engineering is that after interest in process has continued for a certain period of time, interest in products has returned in reaction. The increased interest in architecture in the late 1990s can be seen as a shift in focus from process to product design.

### 9.1.2 The role of architecture

The research community led the way in exploring software architecture. Symbolizing this is the book *Software Architecture*, published in 1996 by Mary Shaw and David Garlan of Carnegie Mellon University [119]. Later, in the industrial world, especially in the development of web-based and component-based application systems, due to the situation of distributed architecture, component deployment, and the use of standard frameworks such as EJB (Enterprise JavaBeans), the naturally came to be conscious of architecture, .

Architecture takes system requirements specifications and determines the structure of the entire system to realize them. The system structure composed of system elements basically corresponds to the structure of the requirements. The architecture determines the functions and behaviors of the system elements and the functions and behaviors of the entire system, which are determined by their composition. In doing so, the following system-wide properties are taken into consideration:

- Processing scale and performance
- Global control structure
- Physical distribution arrangement
- Communication protocols
- Databases and data access
- Outlook for future development directions
- Available components and their suitability

After considering these points, alternative architectures are considered as candidates and selected.

### 9.1.3 Architectural Style

In architectural history, there are many different trends in architectural styles, such as Greek, Roman, Romanesque, Gothic, Renaissance, Baroque, Rococo, Art Nouveau, Modern, and Postmodern. There is a way of thinking to borrow this perspective of style and classify various architectural styles and use them in architectural design.

The following are some representative styles of software architecture.

**Pipes and filters** A filter is a computational component that takes one or more data streams as input, transforms them, and outputs a single data stream. An architecture based on pipes and filters creates an entire computational flow by connecting the output of one filter to the input of another filter via a pipe. A typical example is the pipes and filters of Unix.

**Layered** Computational components are arranged in layers, arranged from top to bottom. Each layer is made up of components that provide similar levels of functionality, and the only components that can be called to realize each function are those in the layer directly below it. Conversely, each layer provides functionality only to the components in the layer directly above it. The layer structure may be drawn as concentric circles, as in Figure 9.2, or it may be drawn as a cross-section of layers that are made up of planes that extend horizontally and have vertical thickness, like geological strata, stacked vertically.

**Event-driven** In this architecture, execution of each component is not initiated by explicit calls between those components, but rather each computational component watches for a specific event and performs a certain action when it occurs. It is also the computational component that generates the event. When the event generator specifies the event receiver, it is close to a procedure call, but when the event is transmitted by broadcasting to an unspecified number of listeners, it is a typical event-driven architecture.

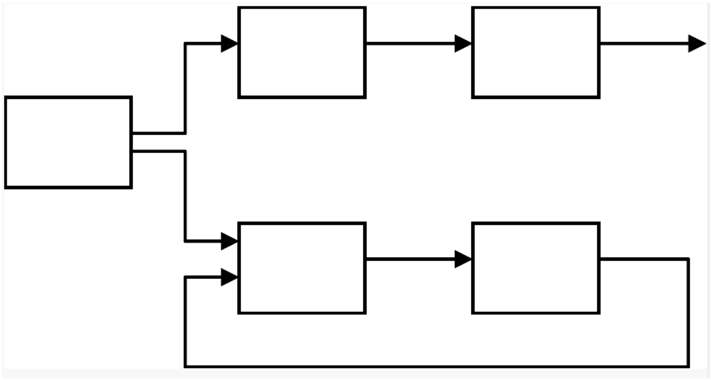


Figure 9.1: Pipe and filter architecture

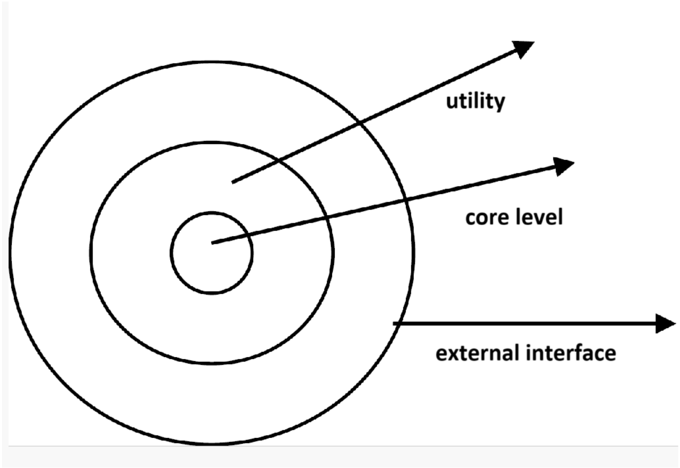


Figure 9.2: Layered architecture

**MVC** An architecture that divides a system into three aspects: model, view, and control, and synthesizes them into a whole is called MVC (Figure 9.3). Originally invented at Xerox's Palo Alto research center and adopted in Smalltalk-80, it has long been used in interactive application systems using GUIs. The "control" receives user instructions and inputs interactively and transmits them to the model, the "model" models calculations and simulations, and the "display" changes the display results according to changes in the model's state. It is relatively easy to replace the display with a different one for the same model or combine it with a different control.

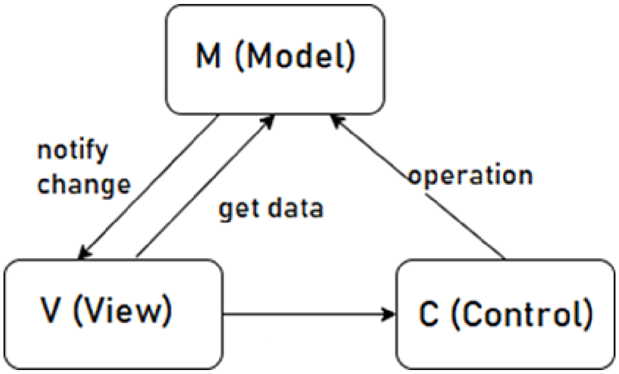


Figure 9.3: MVC architecture

**Rule-based** This is a system architecture that is based on a set of rules that are written as pairs of conditions and actions. Each rule is interpreted in such a way that the action part is executed when the condition part is satisfied. Classic examples of this architecture include Simon & Newell's production system and knowledge-based systems from the days when knowledge engineering was popular. It is suitable for application systems that express knowledge as a set of rules and evolve the overall behavior by adding or changing knowledge.

**Blackboard system** This is also an architecture devised in the field of AI to realize a distributed knowledge system. There are many highly independent knowledge sources, and they interact through a "blackboard," a global shared memory. Each knowledge source reads information from and writes information to the blackboard. It was originally developed at Carnegie Mellon University for use in a speech recognition system.

**Client-Server** The relationship between a web browser and a web server on a network is a typical "client-server" type configuration. A server on a network refers exclusively to a program that provides services. A computer that is running a server program may also be called a server. A client is a program that requests services from a server. The user on the left side of the figure 9.5 uses a web browser to request services such as ticket reservations over the Internet.

Based on the user's request, the Web server creates a page that displays, for example, ticket information or ticket sales status data, and sends it back to the Web browser. In this case, the Web server actually calls an application program that performs the specific processing related to ticket reservations and performs that function. Ticket information and sales status data are usually stored in a database. The application accesses the database server to obtain this information. Here, the application becomes the client, and the side that manages the database becomes the server, creating another client/server mechanism.

### 9.1.4 Architecture Description Languages

Since the late 1990s, when architecture research became popular, various ADL (Architecture Description Language) for describing architectures have been proposed, mainly in academia. Examples include Acme (CMU), C2 (UCI), Darwin

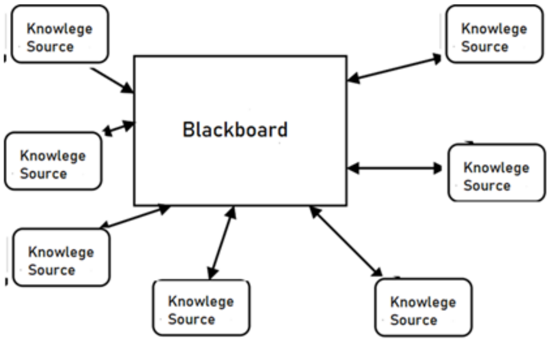


Figure 9.4: Blackboard architecture

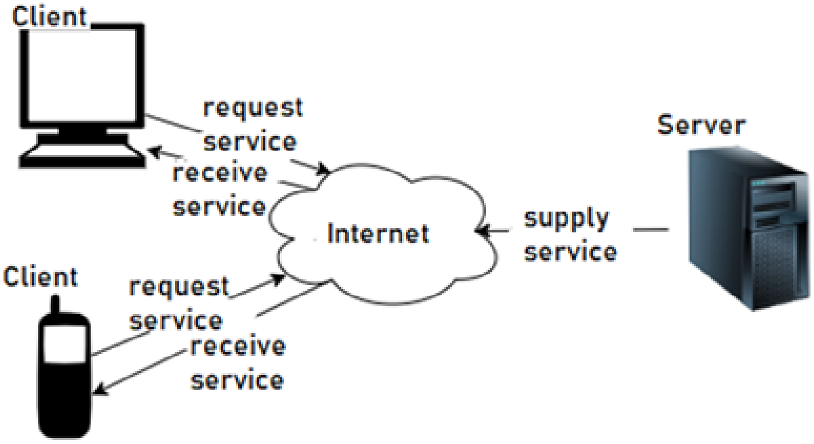


Figure 9.5: Client Server

(Imperial College London), and Wright (CMU). However, these did not become popular in the industrial world.

What these languages have in common is that they have components, often represented as boxes, and connectors represented as lines that connect them. Efforts have been made to clearly define the semantics, and some formal methods are usually used.

However, UML was proposed in the same period and has penetrated the industrial world, so it has become more common to use UML itself or some extended version for describing architectures.

### 9.1.5 Framework

Software architecture is a concept independent of object-oriented design. However, since the mid-1990s, when software architecture was widely examined, and object-oriented technology was also established, many related issues were discussed in conjunction with object-oriented design. Among them, frameworks and design patterns in particular have been widely accepted as a way to increase the productivity of system development and improve product quality.

A framework that embodies an architecture for a specific domain is called a software framework, or simply a framework. By adding specific modules to a framework according to the purpose, a specific application system can be developed. For example, if there is a framework for a sales management system for financial products, it is possible for Bank A to use it to build a specialized system for its own new financial products. The parts with high commonality are already built into the framework as executable code, which differs from architectures that define only the basic structure or those that are written in ADL (architecture description language). Parts that need to be specialized are called hot spots, and are customized using techniques such as subclassing through inheritance. The techniques used are often made to be easy to reuse by design patterns.

A design pattern is a recurring pattern that is smaller than an architecture, as described later. The pattern is expressed by objects and interfaces.

A framework is generally a larger unit than a design pattern, but there are design patterns that are often used within frameworks. The followings are some typical ones.

**Template pattern** This is a design pattern in which the skeleton of an algorithm is written as an abstract class as a template, and client programs that use it create subclasses and redefine methods to use them for individual purposes. In the template pattern, a method with no content is declared in a subclass of the template class, but the control flow is made so that it is called from the the template object as its parent class. The method is intended to be instantiated in a subclass and is called a hook method. Programmers who instantiate hook methods are required to change their way of thinking from writing a subprogram that is called and to writing a program that is called and plays its role.

**Dependency injection pattern** When a client object that uses a service creates an object of a class that provides the service and calls and uses the specific service, the client becomes ”dependent” on the service class. The meaning of dependency here is the same as the dependency relationship between classes in a UML class diagram. Increasing dependency on a specific service class means that the degree of freedom to switch to another service in some cases is reduced. The dependency injection pattern increases the freedom to switch by explicitly creating a mechanism to generate an instance of a service and hand over an access point to the client. For example, when using a database, this can be used to flexibly switch between MySQL and PostgreSQL at run time.

A simple way to implement dependency injection is to set a parameter in a constructor that creates a client object that uses the service, and then ”inject” the dependency when the client object is created by giving the parameter an instance of the service object. Similarly, instead of adding parameters to constructors, you can set an instance variable that holds a reference to a service object, define a setter method for that variable, and use the setter to ”inject” dependencies.

**Inversion of control pattern** The dependency injection described above is considered to be one method that falls under the more general concept of inversion of control. According to Martin Fowler, the inversion of control pattern is as follows[49].

Consider an interactive command line application that listens to the user’s name and requests from the application and responds accordingly, and the application is in control of the entire system. However, when using a window system, the application determines the configuration and layout of the window labels, and writes and assigns methods to be invoked for each component when an event such as keyboard input occurs. In this method, the application does not have overall control, but leaves it to the event-driven mechanism of the window system, so an ”inversion of control” occurs. This corresponds to the ”Hollywood principle” of ”Don’t call us, we’ll call you.” This is the difference between a library and a framework; a library is controlled by the user, but when using a framework the conrol is in the hand of the framework side, in that the framework calls the program code that has been prepared and attached by the user.

**Examples of frameworks**

The Open Group Architecture Framework (TOGAF), which is a framework for the entire enterprise architecture, is one example of frameworks for enterprise applications, but the most widely used frameworks are for web applications. With the spread of the Internet, many business applications have been converted to web applications, and the following parts are commonly included in the basic structure of these applications.

**Data model** Accesses the database to retrieve and update data.

**Business logic** Writes logic according to the functional purpose.

**User interface** Creates the screen displayed to the user, receives input from the user, and manages the interaction with the user.

**Security** Security features are used for user authentication and for sending and receiving payment information.

Web application frameworks realize these features by making full use of the MVC architecture and inversion of control patterns already mentioned. A wide variety of frameworks have been created for different purposes and languages, but some of the most well-known are Spring for Java, Ruby on Rails for Ruby, Django for Python, and CakePHP for PHP.

### 9.1.6 Design Patterns

Design patterns have already been mentioned in the framework explanation, but we will explain them in a separate section. Design patterns are patterns that appear repeatedly in smaller units than architectures. By utilizing a library of patterns, it is possible to create high-quality designs effectively. One particularly well-known book is *Design Patterns: Elements of Reusable Object-Oriented Software* [51], written by four authors, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, which collects and explains 23 design patterns in detail. This book became famous, and the four authors came to be known as the “Gang of Four” or “GoF”. Since then, many design patterns have been collected and published, and various patterns have been proposed, including patterns for higher-level analysis and patterns for lower-level programming (such as a collection of tactics).

The GoF group respects architect Christopher Alexander as a proponent of the idea of patterns in design. Alexander is now the most famous architect in the computer science community. His books *A Pattern Language* [7] and *The Timeless Way of Building* [6] have had a great influence. For the topic of architecture in this section, it is suggestive that Alexander is an architect.

Let’s take the observer pattern as a representative example of a design pattern. The relationship between the model (M) and the display (V) in the MVC architecture corresponds to this observer pattern.

A design pattern generally consists of multiple participants, represented as object classes. There are two types of participants in the observer pattern: the subject, which changes its state, and the observer, which watches the subject’s state changes and takes some action accordingly. There can generally be multiple observers for one subject. The subject adds and removes observers from the list. Figure 9.6 shows the observer pattern, which is taken from the GoF book [51].

The subject, the subject, notifies all observers that the state has changed using the Notify method. Notify performs its role by invoking a method called Update prepared on the Observer side. In this way, design patterns form a basic structure using relationships between abstract classes, but the patterns also include the concrete classes that embody this structure. The box with the folded top right corner of the diagram shows pseudocode.

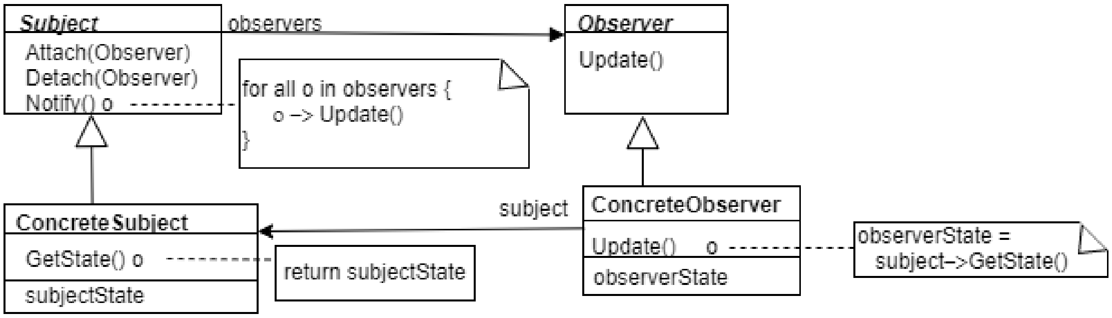


Figure 9.6: observer pattern

### 9.1.7 Software Product Lines

Software product lines (SPL) refer to a group of products that share common parts and are developed and managed as a series of products. They can also refer to the techniques for making effective use of the software assets used in the series and the tools that support them.

The SPL architecture is characterized by a clear distinction between the core assets that are used in common and the variable parts that are added to them. It has been proposed to use the feature model proposed by Kang et al. [**?**] as a way to distinguish what is the base part and what is the variable part.

Feature models are relatively widely used because they identify the features of a product and illustrate their selectability and substitutability, and are visualized and easy to understand.

The idea of adding variable parts to core assets is similar to the framework described in the previous section. In that sense, tools that support SPL can be seen as a type of framework, with software configuration management functions added on top of it.

### 9.1.8 Architecture for the Sake Warehouse Problem

As mentioned in the specification description of the Sake Warehouse Problem in Z (section 8.5), the essence of this problem is that while information on incoming sake comes in containers, sake orders come in brand-by-brand, so a mechanism is needed to mediate between them. One container contains multiple brands of sake, and one brand of sake generally exists in multiple containers. The central decision in the architecture will be how to realize this mechanism.

There are three possible methods:

1. Store data by container, and extract brand-by-brand data from it as needed.
2. Store data by brand, and convert the data to brand-by-brand every time a container is received.
3. Store data for both containers and brands.

Of these, method 2, which stores only brand-by-brand data, is difficult to meet the requirement of detecting when a container is empty and issuing a removal command. At the very least, it will be necessary to retain data on the total number of bottles of sake per container, and whenever the number of bottles of alcohol is reduced from the brand-level data by a shipping instruction, a process will be required to reduce the total number of bottles in the corresponding container. Therefore, below we will consider a method of retaining only container-level data, and a method of retaining both container-level and brand-level data.

The difference between the two comes down to a typical design choice: data or procedure. The method of retaining only container data will be called Architecture 1, or A1 for short, and the method of retaining both container and brand-level data will be called Architecture 3, or A3 for short. The difference between A1 and A3 is whether the inventory information is obtained by applying a procedure to the container data or directly from the brand data, in response to the system requirement to determine the inventory for an order by brand, create a shipping instruction if there is inventory, and notify the customer of the inventory shortage if there is no inventory, and add the order to the inventory shortage list. In this respect, A1 can be seen as a procedure method, and A3 as a data method.

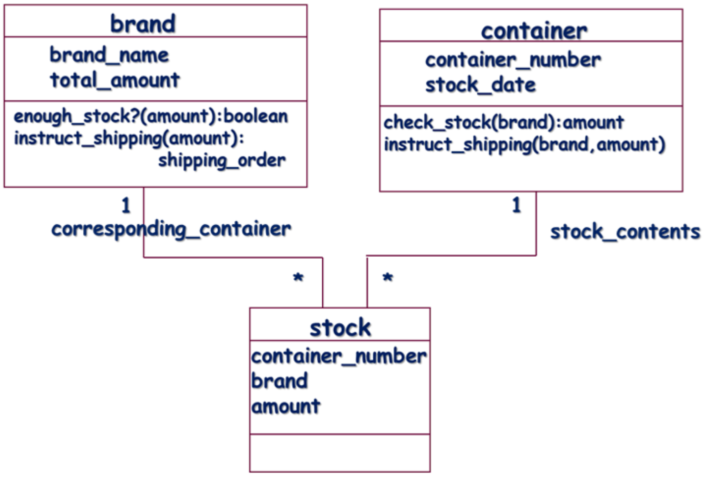


Figure 9.7: Introducing the brand class

The procedure for obtaining brand-level information from container data is not so simple. To realize the function of fulfilling an order, it is necessary to carry out iterative calculations, which look at the containers in order and check the stock of the ordered brands. Moreover, if we think about it simply, it means that one iterative calculation is carried out to check whether there is stock or not, and another iterative calculation is carried out to create a shipping instruction if there is stock.

When the Information Processing Society of Japan held a competition to create solutions using various design techniques for this common problem of the sake warehouse problem, many of the solutions that were created using a method that tried to model the object as naturally as possible took the form of A1. This is because the entity that appears directly in the problem domain is the container, and there is no physical entity that corresponds to a brand with a container number and its quantity in it. In a model that directly reflects the real world, the specific details at the calculation level should be hidden, but by adopting the A1 architecture, it was necessary to bring in low-level descriptions into the model, such as the container being a sequence of sake bottles arranged by brand, and repeatedly searching them according to the order, resulting in a somewhat cumbersome model.

On the other hand, those who chose A3, like Ohno's solution introduced in 6.5.9, often gave explanations of the reasons for choosing the design policy in a way that was given without so much reasoning. It is unreasonable to argue that the entities corresponding to brands can be found naturally from the problem description, so a conscious shift from the requirements model to the architectural design is necessary.

If we introduce an brand class and show its relationship with containers, it will look like Figure 9.7.

Methods of the warehouse class

- enough_inventory?(order): boolean
- instruct_shipment(order): shipment_instruction

If you think of this as an interface for processing according to an order, it doesn't matter whether it is realized by delegating it to the brand class method

- enough_stock?(quantity): boolean
- instruct_shipmen(quantity): shipment_instruction

or as a procedure to search containers. In other words, according to the flow of the discussion so far, the introduction of the brand class is due to the selection of an architecture that realizes it as data, but it can also be considered as a way to

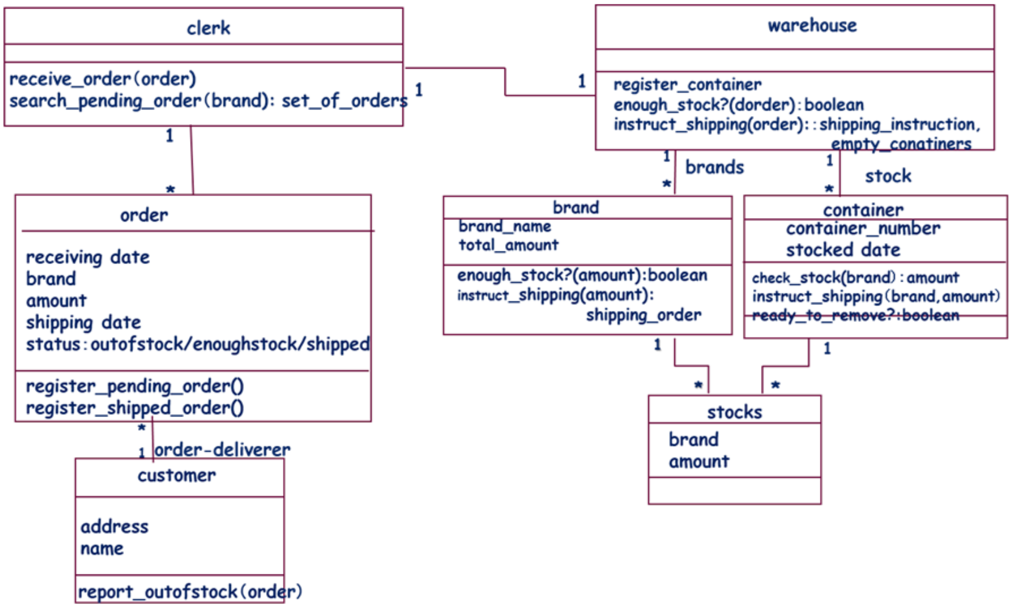


Figure 9.8: Class diagram for the sake store problem: Introducing the item class

simplify the description of the requirements model, leaving aside the decision of whether it is a procedure or data. This kind of consideration is interesting, as it is exactly on the border between requirements analysis and architecture design.

In Figure 9.7, brands and containers are associated with a smaller class, ”stock” as having a one-to-many relationship. This can be seen as a typical method to avoid the many-to-many relationship that would result if brands and containers were directly associated. The original problem of having to view information in units of containers for incoming shipments as units of brands for outgoing shipments is itself a problem of resolving the many-to-many relationship that often appears in data models, so it is not particularly new. As for how to realize this in a database, it may be possible to make brands the secondary index for container data.

Although this is a well-known problem pattern, it is important to consciously consider it as a design decision that arises when considering the architecture from the requirements model and to document the result. Finally, the class diagram of the sake warehouse problem with the brand class introduced is shown in Figure 9.8 (corresponding to Figure **??**), and the corresponding sequence diagram is shown in Figure 9.9 (corresponding to Figure **??**).

## 9.2 Module Design

After the architecture design, the next step is to break down the architecture into more specific component units and design the structure that can be assembled from them. Here, we refer to these component units as modules. The term module has also been used in other engineering fields, such as automobile manufacturing, and in the field of architecture. In those cases, it was emphasized that modules have standardized interfaces and can be replaced with other modules. In particular, from the perspective of production management in business administration for the automobile industry, Fujimoto Takahiro and others have compared American modular production with Japan’s integral production method, evaluating the competitiveness of the Japanese automobile industry [164]. In business administration, the comparison between modular and integral production methods is also discussed in terms of differences in architecture.

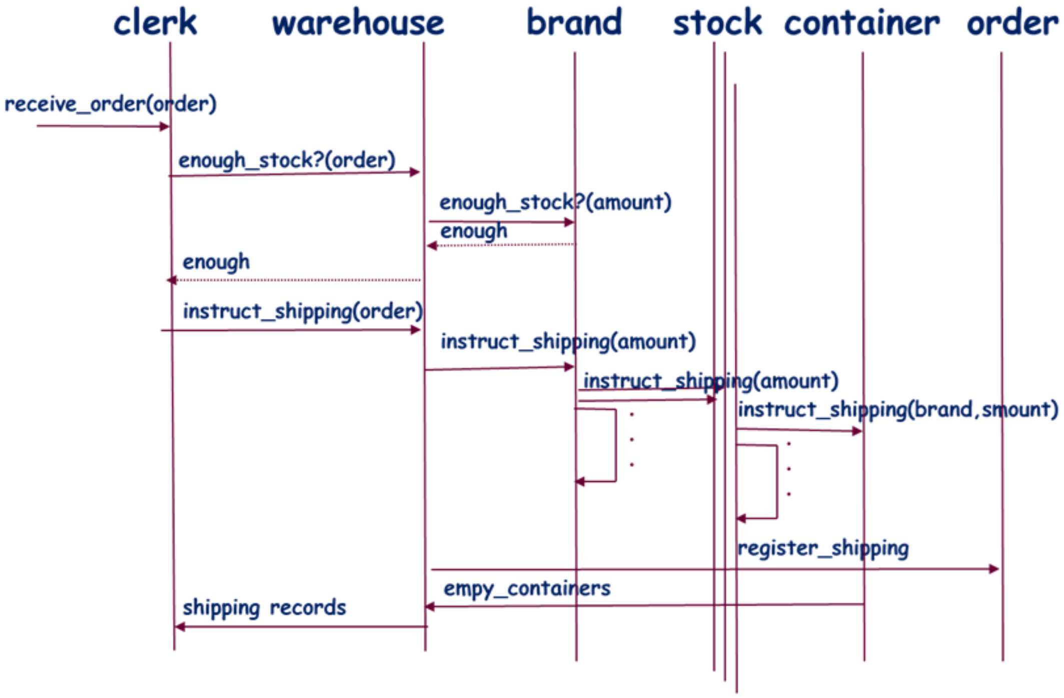


Figure 9.9: Sequence diagram of order acceptance (when in stock): Item class introduced version

### 9.2.1 Meaning of module

The terms module and modular programming in software have been used since the late 1960s, but they suddenly came into the limelight with David Parnas's paper "On the criteria to be used in decomposing systems into modules" [105] published in 1972. The programming language Modula, whose outline was introduced in a paper by Niklaus Writh in 1977, is, as its name suggests, a programming language that places modules at the center of its design. It was later revised to Modula-2 and Modula-3, but it offers syntax and functions that are almost similar to those of modules in modern programming languages such as Python, Ruby, and OCaml.

Since the early days of programming languages, subroutines, procedures, and functions in functional languages have been used as reusable parts of programs. A module usually refers to a larger grouping than these. In fact, a module in programming languages since Modula is a collection of declarations such as variables, types, and methods (functions), usually stored in a single file, and compiled and used as a unit. In terms of a collection of variable and method declarations, it is similar to an object in an object-oriented language.

In the sense that a module is a collection of variable and method declarations, it can be said to have the same granularity as objects and classes in object-oriented languages, but it should be noted that object-oriented languages such as Python and Ruby have a declaration unit called a class, but also a declaration unit called a module, and each plays a different role.

However, modular design is not only necessary when using a programming language that has a language construct called "module." As Parnas also says, a module has the following properties.

1. When creating a module, you do not need much knowledge about other modules.

2. When updating one module, you do not need to update other parts of the program.

Furthermore, the following are listed as the effects of modularization.

- It is easier to divide up the work, which makes project management more efficient.

- Even if a major change is made to one module, it does not affect the others, and the product becomes flexible.
- The system can be analyzed module by module, making it easier to understand.

This means that the modules are highly independent as components, the interface and implementation are separated, the implementation details are hidden, and the modules can be replaced as components. This also increases the reusability of the modules. In software engineering, the term module was used from early on, while software components, although used in examples since the 1960s, have been in the spotlight again, especially from the late 1990s to the 2000s. In this case, components are more conscious of distribution than modules, and the task of component design is expected to involve finding and procuring existing components that suit the purpose. Also, while a module can refer to something written in source code, a component can also mean something that can be used as a unit that can be executed as it is without the trouble of compiling.

The concepts of coupling and cohesion have also long been used as criteria for evaluating modules. The lower the coupling between modules and the higher the internal cohesion of each module, the better the modularity is evaluated as. This became well known through Glenford Myers' book on composite design[94].

### 9.2.2 Aspect-oriented programming

A system composed of modules is divided into modules from a certain perspective. This division reflects the separation of concerns intended by the designer. However, the concept of aspect-oriented was born from the recognition that there may be crosscutting concerns that cross the boundaries of such concerns[80].

A simple example often given is logging. Logging, which records the operation progress of a system, needs to be embedded in various places in the system. However, we want to write it all together, rather than embedding it in separate code. However, if it is made into an independent module, it will have a different character from other modules and will not work as intended.

To solve this problem, a description unit called an aspect, which is different from the usual module unit, was proposed. An aspect is realized by inserting the execution of another aspect code into the middle of the execution of normal program code. This mechanism is usually realized by the following elements.

**Join point** The point where aspect code is inserted and the execution of aspect code joins the execution of normal code is called a join point.

**Pointcut** A set of join points where the same aspect code is applied is called a pointcut. For example, in the case of a logging aspect, the set of execution points after (or before) the action to be logged is the pointcut. This set is usually defined as a conditional expression that determines the characteristics of the join point set.

**Advice** The aspect code executed at the join point is called advice.

We have already introduced the discussion of the architecture of automobile production. In this case, the limit of modular architecture is discussed by raising the question of whether there is a part that is responsible for the ”ride comfort of the car” as an example of a ”cross-cutting concern”. This is similar to the aspect-oriented approach of software. Quality requirements such as software usability are exactly the same as the requirement for the ”ride comfort of the car”. In fact, the aspect-oriented approach has attracted attention in requirements engineering, and various proposals have been made to incorporate aspects in the requirements extraction stage [111, 72].

There are many languages that support aspect-oriented approach at the programming level, including AspectJ. Many frameworks, including Spring, also support aspect-oriented approach.

The concept of AOSD (Aspect Oriented Software Development) was born by generalizing the aspect-oriented approach, and the first international conference on this theme was held in Enschade, the Netherlands in 2002. At that time, the conference chair was Harold Ossher, and the program chair was Gregor Kiczales. Masuhara Hidehiko and Chiba Shigeru joined the conference from Japan as program committee members, and these two subsequently served as conference and program chairs and played an active role in this field. The name of this international conference was later changed to Modularity, and it is still held today.

### 9.2.3 Role model

As another example of a flexible modular structure, we will briefly discuss the role model[133] developed by Ubayashi Naoyasu and the author. This model is based on the following observation.

A computational model with objects as unit elements maps the real world, which is made up of ”things.” In the real world, things are placed in various environments. The environments they are placed in change for various reasons. If the object is a person, the environment changes depending on the day and night, and also periodically changes depending on the day and weekend. If the object moves, the surrounding environment naturally changes, but the environment can change dynamically even if the object does not move. The object also changes in response to changes in the environment. Or, the object itself can change spontaneously, causing a change in its relationship with the environment. Also, multiple environments can exist around an object at the same time, and the object chooses a part of those environments to belong to. When an object binds to its environment, a reflexive change in the environment can also occur.

It is natural to think of the way in which an object and its environment are bound together as being based on ”roles.” At work, people play the roles of boss or subordinate. At home, they play the roles of father, mother, or child. Looking at it from another perspective, the home environment is composed of the roles of father, mother, and child(ren). As a computational model, we can consider a dynamic and flexible modular structure by recognizing the environment, which is composed of roles, as an equal entity to the object, and considering a mechanism in which an object can dynamically bind to a role in an environment to change its behavior, or leave the environment by abandoning that role.

Based on this idea, the Epsilon model was devised. A programming language developed based on Epsilon is the Java-based EpsilonJ, and its processing system is being developed by Supasit Monpratarnchai[93, 130].

Subsequently, Robert Hirschfeld and others proposed Context Oriented Programming (COP) and conducted research activities. The goal of realizing flexible and dynamically changing modules and some of the mechanisms have some similarities to the role model. There is an excellent survey of Context Oriented Programming by Kamino Tetsuo[163].

## 9.3 Interface Design

There are two important types of software interfaces: the user interface and the external interface between other systems.

### 9.3.1 User Interface

If there is a problem with the design of the user interface, it can cause great harm or damage. The following example is a stock order mistake incident that occurred in 2005[128, 159]. For details of the incident, please refer to Chapter 3 ”The Fear of Human Error” in my book ”The Future of Software Society”[159].

#### Background to the Incident

December 8, 2005 was the first day that J-Com, a start-up recruiting company, on the day its shares were offered first on the Tokyo Stock Exchange Mothers Market. J-Com is a company that specializes in the mobile phone business and provides temporary staffing services. At 9:27 a.m., a corporate client of Mizuho Securities placed an order to sell one J-Com share for 610,000 yen. The Mizuho Securities employee who took over the order entered the price at a terminal in Mizuho Securities’ order system, but mistakenly entered 1 as the price and 610,000 as the number of shares. A warning message ”Beyond price limit” was displayed on the screen, indicating that the price had exceeded the limit, but the employee ignored it and completed the transmission to the TSE trading system. The warning was ignored if the Enter key was pressed twice. The total number of J-Com shares issued was 14,500, so 610,000 shares was 42 times that number. At 9:28, this order was displayed on the TSE system order book. In other words, this order was officially accepted by the TSE trading system. At that time, J-Com shares were showing a special bid price of 672,000 yen, and the opening price had not yet been determined. Then a sell order for 1 yen came in, and a deal was made with the opening price at 672,000 yen. Mizuho Securities, realizing the mistake, immediately tried to cancel the order. However, several attempts were unsuccessful. The TSE also discovered the anomaly and, realizing the possibility of a mistaken order, called Mizuho Securities. After several phone calls, both parties realized that the order could not be canceled. The TSE considered suspending trading, but at 9:35, Mizuho Securities began buying back the J-Com shares, so they decided not to suspend trading. As a result of this counter-order, Mizuho Securities ultimately bought back about 510,000 shares, but was unable to buy back 100,000 shares. For the shares that Mizuho Securities could not buy back, even if the buyers who had bought them requested stock certificates, there was nothing to give them. Therefore, a special measure was taken to force-settle the approximately 100,000 shares that could not be bought back in cash at 912,000 yen per share. Mizuho Securities' losses amounted to about 40 billion yen, including losses from the loss from this forced settlement and losses from the buyback.

On December 12, four days after the incident, TSE President Tsurushima held a press conference and announced that the failure to cancel the order was due to a flaw in the TSE's trading system. Between March and August 2006, Mizuho Securities and TSE held several negotiations over damages, but the talks ended in failure. Mizuho Securities then filed a lawsuit against TSE in the Tokyo District Court in October, seeking 41.5 billion yen in damages.

In December 2009, the Tokyo District Court ruled that TSE must pay Mizuho Securities approximately 10.7 billion yen. Mizuho Securities appealed the ruling, but the Tokyo High Court's appeal ruling in July 2013 basically followed the District Court ruling. Mizuho Securities appealed to the Supreme Court, but the Supreme Court ruled in September 2015 not to accept the appeal, and the High Court ruling became final.

### Problems with interface design

This incident can be analyzed from various perspectives, but here we will focus on problems with the user interface design. The following design errors are clearly suspected.

**Separation of price and number of shares** The specific structure of the input frame is unknown, but it is possible that the distinction between these fields was confusing.

**How to issue a warning** It is possible that the warning "Beyond price limit" was easily overlooked. It is also possible that a stronger measure should have been taken to prohibit users from proceeding beyond the warning rather than simply issuing a warning.

**Specification that allows warnings to be ignored by pressing Enter twice** The specification that allows warnings to be ignored by pressing Enter twice is clearly too simplistic. The frequency with which warnings are issued also plays a role, but it is quite likely that users will mechanically press Enter twice.

**Input check** As mentioned in "How to issue a warning," more detailed input checks should have been performed for both the price and number of shares, and their combinations.

**Undo processing** The inability to undo was due to a program error, and not a problem with the interface design, but this clearly demonstrated that undo and cancel are extremely important features of an interface, and that if they do not work properly, it can have a major impact.

The classic work on how such interfaces should and should not be designed is *The Design of Everyday Things* [100] by Don Norman, published in 1988. A revised and expanded edition was published in 2013, 25 years after the first edition was published [101]. The following points are repeatedly emphasized in this work:

- There should be a natural correspondence or mapping relationship between the handles, knobs, levers, etc. used to control tools and equipment and the objects or actions that are being controlled.
- After the user performs an operation or control action, feedback should be given immediately to keep the user informed of the current state.
- Make it easy for users to construct conceptual models of tools and equipment.
- Impose various constraints to prevent erroneous operations.
- Early detection and reporting of erroneous operations, easy correction such as undoing, and quick recovery from error states.
- Do not confuse users with excessive warnings or information.

These points are well suited to the lesson learned from the erroneous order problem mentioned above.

**Affordance**

The term and concept of affordance, coined by psychologist James J. Gibson, was already known, but became even more famous with Norman's 1988 edition of the book. Affordance is the ”meaning” or ”value” that things afford animals when they perceive ”things” and other animals and plants in their surrounding environment and recognize what they can do and what they can be used for. For example, the ground can support or place things on it, and you can walk or run on it, and these are the affordances of the ground. Norman applied this concept to artificial designs, explaining that if a door has a horizontal bar on the front, you can pull it towards you to open it, and if it has a square plate, you can push it to open it. This was a persuasive way of thinking about interface design and became widely known, but later, the 2013 version added a note that affordance means what actions are possible, and that ”signs” that people use to know what actions are possible (such as the shape of a door handle) should not be called affordances. So what should we call such ”signs”? Norman introduced the word signifier from Ferdinand de Saussure.

However, from the perspective of interface design, it is important to be able to intuitively understand how to operate something from the outside, so the ”incorrect” use of affordances seems to make more sense. If we call it ”signifier”, it may not make sense.

### 9.3.2 External Interfaces

Modern software is closely connected to other systems via networks, and often uses external libraries and frameworks. This means that the system or module being developed may be used elsewhere. Therefore, as with interfaces for using external things, how to design the interface for the outside world is an important theme.

There are many things to consider about such external interfaces, but here we will briefly discuss ”Application Programming Interfaces (APIs)”.

An API is an interface for services and libraries used to develop applications that is publicly available. However, the scope of availability is not necessarily limited to the general public, and there are cases where it is limited to within the company or to a group involved in business collaboration. Web APIs for applications on networks are particularly widely used. APIs usually refer to something that is provided as a specification, but the term API can also include the implementation.

By creating API specifications using a documentation system such as Javadoc or Pydoc, the style is standardized, the creation efficiency increases, and it is easy for users to understand.

## 9.4 Design of Algorithms

A certain algorithm is realized in a module.

In the 1970s, how to design data structures and algorithms was one of the important themes in software engineering. After that, software engineering became specialized, and the fields of data structures and algorithms were separated as independent fields. From the software engineering perspective, it can be said that the emphasis shifted to analysis, design, and management at a higher level than the program level.

However, the design of algorithms and data structures is not only an important part of software design, but also provides fruitful concepts and methods for considering higher-level architectural design. In that sense, it is unfortunate if software engineering neglects this level of design.

Fortunately, many excellent books have already been written on the topic of algorithms and data structures (for example, Aho, Hopcroft & Ullman[5] and Kiyoshi Ishibata[161]). I will leave the systematic description to those textbooks, but here I will look at some examples of how to construct iterative and recursive programs.

The formal methods discussed in Chapter 8 describe specifications formally, so in principle it is possible to derive a program from those specifications using formal methods. If this could be fully automated, it would be called automatic programming. If it were done by a human, rather than automated, it could be considered a design task. In that case, too, it would be possible to derive a systematic algorithm/program that overlaps with automation. Designing data structures and algorithms that are linked to formal specifications in this way can be said to ensure high reliability from the start. It is also closely related to the verification technology discussed in Chapter 10, and the fusion of design and verification technologies is expected to produce high quality.

The main material below is taken from David Gries [55] and Jon Bentley [13, 14].

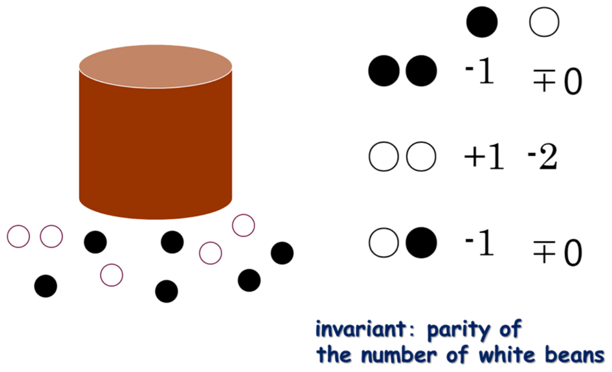


Figure 9.10: Coffee can problem

### 9.4.1 Invariants

When designing an iterative (or recursive) program, it is important to think like this:

1. Clarify the preconditions and postconditions of the program.
2. Define the invariant condition $I$ that will not change during the iteration. (One way to find a meaningful invariant is to relax the postcondition appropriately.)
3. The precondition before entering the iterative structure must satisfy the invariant. (Usually, consider the initial settings before entering the iteration. In some cases, change the invariant.)
4. Clarify the condition $R$ for continuing the iteration.
5. Verify that the condition $I \wedge \neg R$ that holds when you escape the loop is the desired postcondition (or rather, determine $I$ and $R$ so that it does).
6. Verify that the loop is moving in the direction in which the loop exit condition holds (usually the direction in which the postcondition holds, guaranteeing that the loop will end).

The next problem makes you think about the meaning of invariants.

**[Problem 1] Coffee can problem**

A can contains white and black beans. Two beans are randomly taken out of the can. If the two are the same color, they are discarded and a black bean is put back instead (assuming there are many black beans outside). If the two are different colors, the white bean is put back and the black bean is discarded.

If you repeat this process, eventually one bean will remain in the can. What can you say about whether the bean is black or white?

Before looking at the answer, please think about it for a while. The hint is to consider the invariant condition.

**Answer**

With each operation, one bean is removed from the can. How does the number of white beans and the number of black beans change at that time? If we divide the cases according to the description of the problem,

| Two beans taken out | beans to put back | number of whites | number of blacks |
|---|---|---|---|
| White-white | black | -2 | +1 |
| Black-black | black | ±0 | -1 |
| White-black | white | ±0 | -1 |

In other words, the number of blacks increases or decreases by 1 each time, but the number of whites either decreases by 2 or remains the same.

Therefore, we can see that the parity of the number of white beans in the can is invariant.

In other words, if the number of whites is an even number at the beginning, it will always be an even number, and if it is an odd number at the beginning, it will always be an odd number.

Therefore, if the number of white beans is an even number at the beginning, the number remaining at the end will be black, and if it is an odd number, the number remaining at the end will be white.

### 9.4.2 Designing an algorithm using invariants

Next, the program problem.

**[Question 2] The longest support length**

Suppose an array of integers is given, and the sequence is monotonically non-decreasing. In other words, if the array is $a$ and its size is $n$,

$$a[0] \leq a[1] \leq \cdots \leq a[n\ -\ 1]$$

a support is a sequence of numbers with the same value. In this case, find the longest support length where $n > 0$.

**Concept**
The following is from Gries.

If the maximum support length obtained as a result of the program is $p$, what will happen if the output conditions are written correctly? $R:\ (\exists\, k : 0 \leq k < n-p : a[k] = a[k+p-1]) \wedge$ This shows that array $a$ has a support of length $p$
$(\forall\, k : 0 \leq k < n-p-1 : a[k] \neq a[k+p])$
but not $p+1$.

It is almost self-evident that writing this program requires repetition. So how do we choose an invariant? One common way is to weaken the output condition $R$. Another common way is to change the constant that determines the size of the problem into a variable. In this case, instead of saying that the maximum support length in the interval $[0\ :\ n\ -\ 1]$ is $p$, we introduce the variable $i : 1 \leq i \leq n$ and say that the maximum support length for the interval $[0 : i-1]$ is $p$.

The initial value can be set to $p = 1;\ i = 1$, and the termination condition for the iteration is $i = n$. If $n - i$ is made to decrease monotonically with each iteration, termination is guaranteed.

Invariant

$$\begin{aligned} P:\ & 1 \leq i \leq n \wedge \\ & p \text{ is the maximum support length for } a[0 : i-1] \end{aligned}$$

The loop can be written to satisfy this invariant as follows.

```
i=1;  p=1;
while (i!=n)
    if (a[i]==a[i-p]) {i++; p++;}
    else i++;
```

**[Problem 3] Greatest common divisor**

Two natural numbers $x$ and $y$ are given, and we create a function $gcd(x,y)$ that finds the greatest common divisor using Euclidean algorithm.

1. Precondition: $x = \alpha \wedge y = \beta \wedge \alpha > 0 \wedge \beta > 0$

2. Postcondition: $return\_value = gcd(\alpha, \beta)$

It is natural to take the invariant assertion as follows.

$$x > 0 \land y > 0 \land gcd(x,y) = gcd(\alpha,\beta)$$

[Method 1] Use subtraction. In this case, use the following axioms. In the following, $x > 0, y > 0$.

$$\begin{array}{rclcl} gcd(x,y) & = & gcd(x,y-x) & = & gcd(x-y,y) \\ gcd(x,x) & = & x & & \\ gcd(x,y) & = & gcd(y,x) & & \end{array}$$

Example program

```
/* gcd by subtractions */
gcdSub(int x, int y) {
   while (x != y) {
      if (x > y) x = x-y;
      else y = y-x;
}
return x;
}
```

[Method 2] Use division. The axioms used are

$$\begin{array}{rclcl} gcd(x,y) & = & gcd(x,y\%x) & = & gcd(x\%y,y) \\ gcd(x,0) & = & gcd(0,x) & = & x \end{array}$$

In this case, the invariant changes slightly as follows:

$$x \geq 0 \land y > 0 \land gcd(x,y) = gcd(\alpha,\beta)$$

Program example

```
/* gcd by divisions */
gcdDiv(int x, int y) {
   int t;
   while (x) {
      t = x;
      x = y%x;
      y = t;
   }
   return y;
}
```

**[Problem 4] Binary search**

Given an array $x$ with numbers sorted in ascending order and an arbitrary number $q$, if $q$ is in the array, return its position, otherwise return $-1$.

Let's define the problem a little more precisely.

1. Precondition: Given an integer array $x[0:n-1]$ of size $n$ and an integer $q$.

$$n \geq 0 \land x[0] \leq x[1] \leq \cdots \leq x[n-1]$$

2. Postcondition: If $q$ is in $x$, then $x[p] = q$, otherwise $p = -1$

The method used is binary search. According to Knuth, the binary search algorithm was first published in a rough form in 1946 and was widely known, but a proper one that could be applied to arrays of any size was not published until 1960 by D. H. Lehmer [82]. Even after that, this algorithm was prone to errors, and many versions with errors were published.

Bentley's solution to this problem (especially the explanation of the derivation of the algorithm using the concept of invariants) is roughly as follows.

1. First sketch

```
Initialize the interval to $0 : n - 1$
Repeat the following
    {Invariant: If there is one, it is in the interval}
    If the interval is empty
        End as not found
    Find the middle of the interval and call it $m$
    Shrink the interval using $m$
        If $q$ is found when shrinking the interval, return its position
```

2. Concretize the interval

```
l=0; u=n-1;
Repeat the following
    {Invariant: If there is one, it is between l and u}
    If l > u
        Set p=-1 and exit the loop
    m=(l+u)/2
    m to reduce the interval l:u.
        If $q$ is found during the interval reduction, record its position and exit the loop.
```

3. Case classification by comparing *q* and *x*[*m*].

```
l=0; u=n-1;
Repeat the following.
    {Invariant condition: If there is, it is between l and u}
    If l > u, then
        p=-1; break
    m=(l+u)/2
    case
        x[m] < t:  l = m+1
        x[m] == t:  p = m; break
        x[m] > t:  u = m-1
```

Similar works include Dijkstra & Feijen[42], Nogi [167], and Arisawa[155], all of which provide abundant related materials.

# Chapter 10

# Verification & Validation

Product inspection is always performed before the shipment of any industrial product. This is the same for a microchip that fits on your fingertip or a car. For software, verification work to ensure that it is made according to the required specifications is also extremely important. However, unlike making many identical products and inspecting them to see if they are made the same way, each piece of software is a unique and independent entity, and the concept and methods of inspection and verification are often different from those of other industrial products.

## 10.1 Basic Concepts of Verification

Verification is the process of verifying that software meets the required quality and is reliable.

### 10.1.1 Terminology

There are many terms related to verification, and they are confusing. In the United States, the term Verification & Validation, or V&V for short, is often used. However, there is not necessarily a common understanding of the difference between verification and validation. It is relatively widely accepted that the former is the process of verifying the quality at each stage of software development, while the latter is the process of verifying the quality of the entire process, especially the final product.

From a slightly different perspective, verification is sometimes described as showing that a program correctly meets each requirement specification, while validation is inspecting whether the product meets the customer's expectations.

According to B. Boehm, the former is concerned with whether the product is made correctly and the latter is concerned with whether a correct product is made.

There is also confusion over the terminology used for errors that verification should detect. When a component of a system loses its function, it is usually called a fault, and if this manifests as an error, and if the error is not corrected, it becomes a failure.

### 10.1.2 Requirements and Verification

As mentioned at the beginning of this section, if verification is the task of confirming that software meets the required quality, verification technology is closely related to requirements technology, and it is desirable to determine the verification plan according to the content of the requirements at the requirements confirmation stage.

As mentioned in the 3.5.1 section, the general idea is to divide requirements into functional requirements and non-functional requirements or quality requirements. Functional requirements describe how software should operate and what tasks it should perform. Therefore, the verification of functional requirements is to demonstrate that the operation of the implemented system realizes the required functions. Many of the traditional verification techniques, such as testing, have been used mainly to verify functional requirements.

Some of the characteristics of non-functional requirements, such as execution speed, memory capacity, security, interface, ease of use, portability, and maintainability, can be verified through testing, but the data to be prepared and the evaluation method are naturally different from those for functional verification. Simulation technology can be used to verify performance such as execution speed and memory capacity, and model checking technology can be used to verify security.

### 10.1.3 Various Verification Techniques

Depending on what is being verified, verification can be divided into those that target requirements or design specifications and those that target programs.

**Verification of Specifications** As a method for verifying specifications, if the specification description is written in a formal language, some kind of (automatic) analysis method can be considered. In particular, for specifications related to behavior, as mentioned in the 8.4 section, model checking is effective for verifying their safety and liveness, and its application has expanded not only to the verification of hardware logic circuits and communication protocols, but also to general software.

A more practical method is a team review, which is carried out by many organizations. There are terms and methods with slightly different nuances, such as walkthrough and inspection, but they are all based on human visual inspection.

**Program Verification** There are various methods for verifying programs. Let's consider them as dynamic and static verification.

Dynamic verification is verification that actually executes the program. Testing is one of the most representative methods. Other methods include analysis of the frequency distribution of execution counts (called a profile) by module or by executable statement of the source program, measurement of the ratio of executed statements (called coverage), and checking by inserting assertions (logical conditions that should be true) into the program.

Static verification is the analysis of the characteristics of a program without executing it. Static type checking is an extension of the syntax error checking performed by compilers. Research on type checking has progressed, particularly for functional languages, and now powerful checking is possible for many general languages at compile time. Type checking is based on type theory, but the application of type theory goes beyond type checking, and has had a major impact on type inference and even the design of programming languages. For example, Atsushi Igarashi's research on generic types has been adopted in Java 5.0.

Furthermore, static program analysis technology analyzes the control flow and data flow of a source program, and based on that, it extracts the structure and meaning of the program, or points out parts that seem inappropriate. In fact, static analysis tools for various languages are in practical use and are being used. Another type of static verification is the correctness proof technique using Floyd-Hoare logic, as mentioned in the 8.3 section. There are also various efforts to use model checking directly to verify programs, while it was introduced to verify specifications,

Figure 10.1 illustrates the relationships among a variety of V& V techniques mentioned above.

## 10.2 Program Verification Techniques

We will introduce program verification techniques, focusing on testing techniques. Testing is one of the oldest techniques for inspecting and verifying software quality, and it still plays a central role today.

### 10.2.1 Basic properties of testing

Testing executes the target program under certain conditions and analyzes the results.

In that sense, it is a dynamic technique, in contrast to static techniques that analyze program code and verify quality. "Under certain conditions" means that test data is selected and the execution environment is controlled for execution. When executing a test case, it is important to be able to predict in advance how the program will behave and what results will be obtained. If the prediction and reality differ, it indicates that there is a defect in the target program.

E. W. Dijkstra famously said, "Program testing can be used to show the presence of bugs, but never to show their absence!" There are usually an infinite number of possible execution cases for a program, or even if they are finite, the

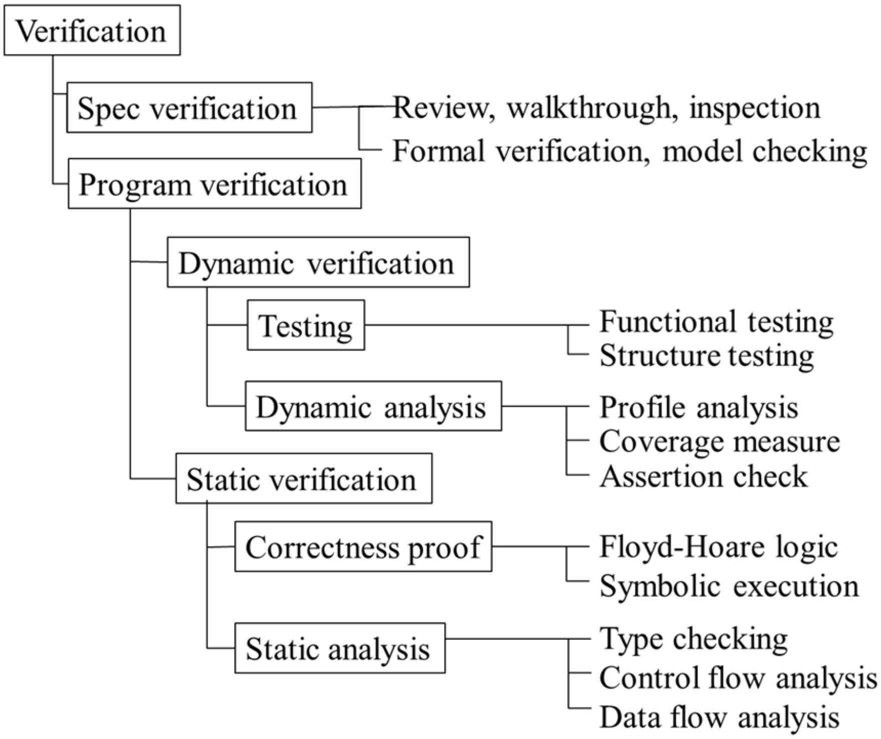


Figure 10.1: Types of verification techniques

number is often so large that it is practically impossible to cover them all. Therefore, testing is essentially a sample survey. A perfect test does not exist except in very exceptional cases.

Of course, this does not mean that testing is meaningless. If a bug is found during testing, it is possible to prevent possible problems during operation by removing the cause, at least the problem that was found. If no problems occur during testing, reliability will increase. However, the fundamental reason why the problem of how much testing should be performed and how to select test cases are inherent to testing comes from the fact that testing cannot be complete.

Due to the basic nature of such testing, there is a school of thought that the purpose of testing is not to ensure that a program satisfies its specifications, but to find errors[95]. Expanding on this, the argument goes that a test is considered successful when it finds an error, and fails if it does not. This explains well the psychological tendency that it is easy to find bugs in programs written by others, but difficult to find bugs in one's own programs, and one's own programs are lax in testing. This is why in software development, it is common for the development team and the inspection team to be independent organizations. However, if you consider that the purpose of testing is to find errors, it may lead to somewhat backward-looking psycology. It also does not match the pleasure that programmers feel when they test and the results are good. Therefore, in "Software Testing Techniques," which I published quite some time ago with Yoshitake Mishima and Shigehiro Matsuda, I attempted to define testing as follows[160].

**Definition of testing:** Testing is the process of running a program on selected data and evaluating the results in order to find as many errors as possible and, if no errors are found, to increase confidence in the quality of the program.

Books on testing techniques include the classic *The Art of Software Testing* published by Glenford Myers in 1979 (revised in 2011), as already mentioned, and more recent works such as *Software testing and analysis: process, principles, and techniques* [106] by Mauro Pezze and Michal Young.

### 10.2.2 Selection of Test Cases

The technical challenges of testing include the following.

- How to select test cases.
- How to create an environment for executing tests.
- How to evaluate test results.

A test case is a combination of the conditions for executing a test and the expected results of that execution.

Test cases are often determined by determining test data, but in the case of responsive software, the execution conditions are determined not only by data given from the outside but also by the internal state of the system.

There is a school of thought that distinguishes between functional testing and structural testing based on the method of selecting test cases. Functional testing is a method of determining test cases only from the functional specifications of a program, and is also called black-box testing. Structural testing is a method of creating a set of test cases that depend on the internal structure of a program and cover that structure as thoroughly as possible, and is also called white-box (or glass-box) testing.

#### Functional Testing

1. **Equivalence Partitioning**
   This involves dividing the input space into subspaces that are equivalent to the test, and selecting one test case from each part. Equivalence to the test means that if there is no error when it is run with one test data, it is guaranteed that there will be no error with other data of the same type. The image is illustrated in Figure 10.2. If the parts 1 to 5 in the figure are each equivalent classes, then by taking one appropriate point from each of them (the image is shown in the figure with ×), an appropriate set of test data can be obtained.

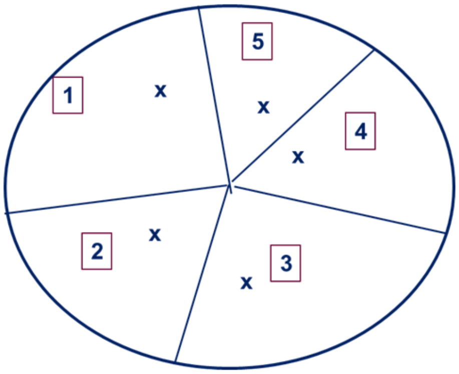


Figure 10.2: Equivalence Partitioning

   However, this is an ideal story, and in reality, equivalence classes in this sense cannot be determined, or even if they can be determined, the number of classes is usually too large to be practical.

   Therefore, a convenient method is used, such as if the input is a range, it can be inside or outside that range, or if it is a discrete value, each discrete value can be classified into one class.

   Let's consider the example of binary search given in the 9.4 section. The specifications are as follows:

   (a) Precondition: Given an integer array $x$ of size $n$ and an integer $q$.

$$n \geq 0 \land x[0] \leq x[1] \leq \cdots \leq x[n-1]$$

```
/* binary search */
int bsearch(int x[], int n, int q) {
/* pre  n >= 0,
           x[0]<=x[1]<=...<=x[n-1]
    post x[p]=q, if q is in x,
           p=-1, otherwise,
           where p is the returned value. */
    int l, u, m;
    l = 0; u = n-1;
    while (l<=u) /* if q is in x, q is in [l:u] */ {
        m = (l+u)/2;
        if (x[m]==q) return m;
        else if (x[m]<q) l = m+1;
        else u = m-1;
     }
     return -1;
}
```

Figure 10.3: Binary search program

(b) Postcondition: Let the return value be $p$, and if $q$ is in $x$, then $x[p] = q$, otherwise $p = -1$

An example of a program written in C that satisfies this specification is shown in Figure 10.3.

The input data for this problem are $n$ and an array $x$ and $q$. Table 10.1 considers natural interval divisions for each. Here, invalid equivalence classes are also considered that fall outside the conditions of the specification. This is an important way to check the robustness of a program against unexpected inputs.

Table 10.1: Example of equivalence partitioning for binary search

| Input | Valid equivalence classes | Invalid equivalence classes |
|---|---|---|
| $n$ | 1) $n > 0$<br>2) $n = 0$ | 3) $n < 0$<br>4) Size of array $< n$ |
| $x$ | 5) $x[0] < x[1] < \cdots < x[n-1]$<br>6) For at least one $i$, $x[i] = x[i+1]$<br>7) $x[0] = x[1] = \cdots = x[n-1]$ | 8) For some $i$, $x[i] > x[i+1]$ |
| $q$ | 9) $x$ contains $q$<br>10) $x$ does not contain $q$ | |

In terms of equivalence classes, it is a combination of these three kinds of data classes. Trying to exhaust all combinations would result in too many test cases. Except for cases where the correlation between data is strong and the behavior changes significantly depending on the combination, it is more practical to assume that each data class is included in at least one test case.

2. **Boundary Value Analysis**

   As with equivalence partitioning, the input space is divided into subspaces by some criterion, and test data is selected from the boundaries of these subspaces or their vicinity. Also, pay attention to the output space and devise test cases that produce results on its boundaries. It is based on the knowledge that errors are likely to occur when exceptional events occur, and that these often occur on or near the boundaries of the input and output subspaces. In addition, by consciously selecting data that is outside of the normal input values, it also serves the purpose of testing the robustness of the program.

   For the equivalence partitions given as intervals, for example, we select those near the boundaries as follows.

```
/* test ’bsearch’ */
main() { int x[11], n, i;
    for (i = 0; i <= 10; i++) x[i] =2*i;
    for (n = 0; n <= 10; n++) {
       printf("n= %d\n", n);
       for (i = 0; i < n; i++) {
          assert(bsearch(x,n,2*i) == i);
          assert(bsearch(x,n,2*i-1) == -1);
          assert(bsearch(x,n,2*i+1) == -1);
        }
        assert(bsearch(x,n,-2) == -1);
        assert(bsearch(x,n,2*n) == -1);
     }
}
int assert(int a) {
   if (!a) printf("Assertion failed!\n");
}
```

Figure 10.4: Test program for binary search

**1)** $n > 0, n = 1, n = 2$, etc.

**3)** $n < 0, n = -1$, etc.

**4)** $n =$array size$+1$, etc.

**9)** $q = x[0], q = x[n-1]$, etc.

**10)** $q < x[0], q > x[n-1], x[i] < q < x[i+1]$ for some $i$, etc.

Figure 10.4 is a modified version of Bentley’s test for the binary search program above. It includes a number of such boundary test cases.

3. **Cause-Effect Graph**
Different input conditions are identified as causes, and different output conditions are identified as results. The logical relationships between them are represented in a graph (cause-effect graph) that follows certain rules. The cause-effect graph can be converted into a decision table, and specific test data is generated from the decision table. This is effective for testing programs that have complex combinations of inputs and outputs, but do not have an explosive number of cases. For example, this method is suitable for objects that can be represented by a state transition model.

As an example, let’s consider the bowling score calculation (see 6.4.4 section), which was taken up in the state transition model. The causes are the following events.

- Result of previous frame: {open, spare, strike, double}
- Which throw in current frame: {first throw, second throw}
- Number of pins knocked down: {cleared, remaining}

The results are as follows.

- Score calculation method: {add the number of pins knocked down, add twice the number of pins knocked down, add three times the number of pins knocked down}

A cause-effect graph is a graph in which each cause item is a vertex, and their logical connections are represented by edges of ”and” connections or edges of ”or” connections, and these are connected to the vertex of the result. Figure 10.5 shows the cause and effect of this bowling score calculation.

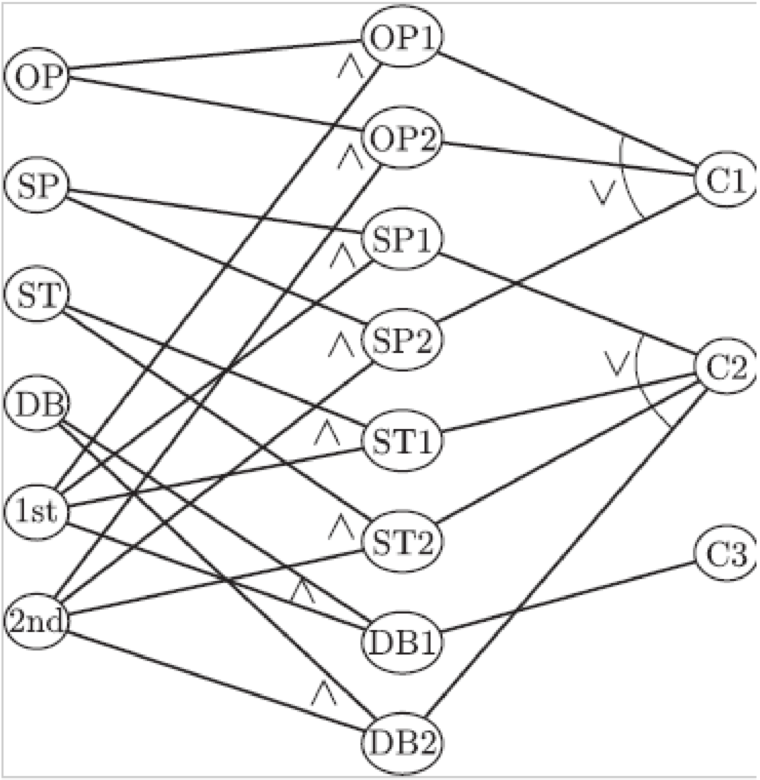


Figure 10.5: Example of a cause-effect graph (bowling score calculation)

In the figure, the vertices OP, SP, ST, and DB correspond to the causes open, spare, strike, and double, respectively, and 1st and 2nd correspond to the first and second throws, respectively. The cause event of whether all pins were knocked down or left standing does not directly affect the result event of the score, so it is omitted from the graph. In addition, the vertices C1, C2, and C3 correspond to the result events of adding the number of pins knocked down, adding twice the number of pins knocked down, and adding three times the number of pins knocked down, respectively.

The set of edges entering the vertices of the figure are marked with a $\wedge$ symbol if they are connected with ”and,” and a $\vee$ symbol if they are connected with ”or.” Other logical symbols such as negation and exclusive disjunction can also be used, but they are not necessary here, so we will omit the explanation.

The result events are distinguished by the three types of addition methods of points, but the next state to move to is also another result event. When considering this, it is necessary to add the event of the number of pins knocked down (especially whether all pins were knocked down or whether one or more pins remain), which was ignored earlier, to the cause events. This will greatly increase the number of cause cases, but the state transition diagram in Figure **??** (reposted as Figure 10.6) actually summarizes this to as few states as possible. In this diagram, the state is determined by the result of the previous frame and the current number of throws, and the transition event is determined by whether it is ”all knocked down” or ”remaining”. In addition, if all the pins are knocked down with the first throw of a frame, there is no second throw, and if there is a second throw, the score calculation is the same after an open and a spare, and the same after a strike and a double, so the number of states can be reduced by taking advantage of this.

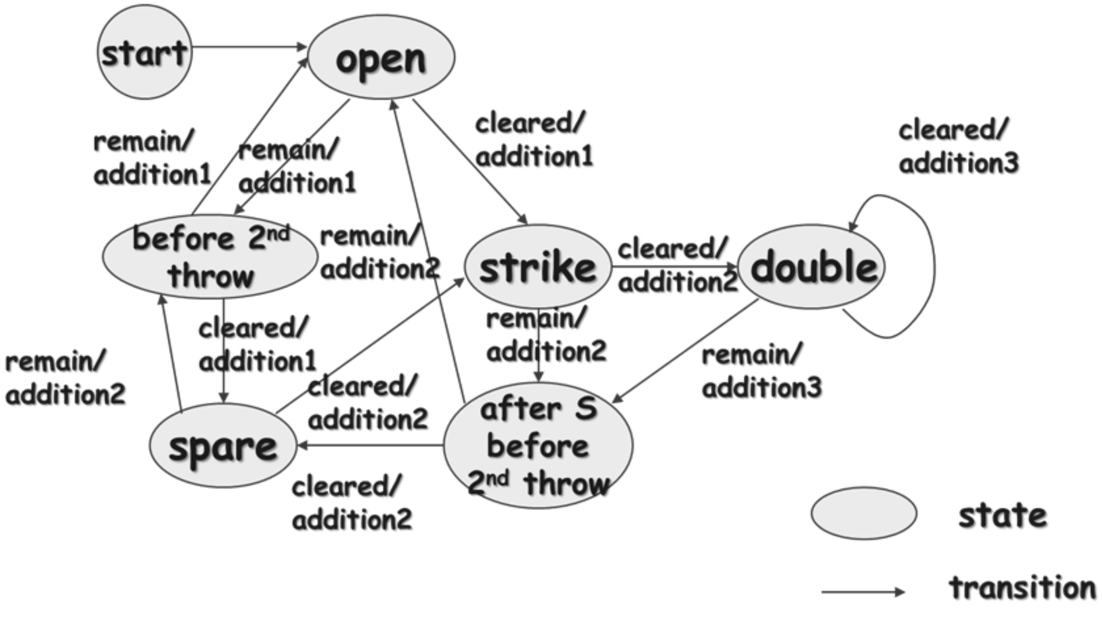


Figure 10.6: Bowling score calculation model (reposted)

Using this state transition diagram, we can create a decision table that matches the causes, i.e., the current state and transition events, with the corresponding results, i.e., the addition of points and the next state, as shown in the table 10.2.

Here, there are 12 columns with values 0 and 1, each of which corresponds to a test case. The state and transition event define the test input, and the result and next state represent the expected test result. 1 corresponds to the corresponding event occurring, and 0 corresponds to its non-occurrence.

Table 10.2: Decision table for testing bowling problems

| | | |
|---|---|---|
| State | Open | 1 1 0 0 0 0 0 0 0 0 0 |
| | before 2nd throw | 0 0 1 1 0 0 0 0 0 0 0 0 |
| | Spare | 0 0 0 0 1 1 0 0 0 0 0 0 |
| | Strike | 0 0 0 0 0 0 1 1 0 0 0 0 |
| | before 2nd throw after strike | 0 0 0 0 0 0 0 0 1 1 0 0 |
| | Double | 0 0 0 0 0 0 0 0 0 0 1 1 |
| Transition event | cleared | 1 0 1 0 1 0 1 0 1 0 1 0 |
| | remaining | 0 1 0 1 0 1 0 1 0 1 0 1 |
| Result | Addition 1 | 1 1 1 1 0 0 0 0 0 0 0 0 |
| | Addition 2 | 0 0 0 0 1 1 1 1 1 1 0 0 |
| | Addition 3 | 0 0 0 0 0 0 0 0 0 0 0 1 1 |
| Next state | Open | 0 0 0 1 0 0 0 0 0 1 0 0 |
| | 2nd throw | 0 1 0 0 0 1 0 0 0 0 0 0 |
| | Spare | 0 0 1 0 0 0 0 0 1 0 0 0 |
| | Strike | 1 0 0 0 1 0 0 0 0 0 0 0 |
| | 2nd throw before strike | 0 0 0 0 0 0 0 1 0 0 0 1 |
| | Double | 0 0 0 0 0 0 1 0 0 0 1 0 |

4. **Test cases based on use cases**
   When requirements analysis is described based on use cases, it is natural to consider test cases that correspond to them. This can also be considered a type of functional testing, as test cases are selected from the specifications.

   A use case generally consists of several scenarios (one execution sequence), so at least one test case will be generated for each scenario. Since exceptional cases should be described as separate scenarios, there are cases where it is not necessary to independently use a method such as limit value analysis, but when the input data space for one scenario is large, it may be desirable to use the idea of equivalence space or boundary values in conjunction with the idea of what kind of test data to select from within that.

### Structural testing

1. **Test case selection based on coverage**
   This is a test that covers as many of the execution units (execution statements and branch decisions) that make up a program as possible based on the program's structure.

   A commonly used indicator is test coverage, which measures how much testing has been achieved by accumulating the execution of a series of test cases. Specific indicators include statement coverage and branch coverage.

   - **Statement coverage** An indicator of what percentage of all execution statements have been executed at least once. The symbol $C_0$ is often used.
   - **Branch coverage** An indicator of what percentage of all execution branches, such as if statements, case statements, while statements, etc., have been executed at least once. The symbol $C_1$ is often used.

   The basic principle of structural testing is to select and execute data that maximizes statement and branch coverage. To achieve this, it is necessary to extract statements and branches that have not yet been executed and generate test data that executes them. To do this, we define an execution path that includes statements and branches that have not been executed, and prepare test data that passes through that path.

   For example, Figure 10.8 is a flow graph of the binary search program in Figure 10.7 (Figure 10.3 has been slightly modified by removing the annotations and adding line numbers, etc.). A flow graph is a graph in which a continuous execution fragment of a program is a vertex, and the movement of execution control is represented by an edge.

```
/* binary search */
int bsearch(int x[], int n, int q) {
/* pre   n >= 0,
         x[0]<=x[1]<=...<=x[n-1]
   post  x[p]=q, if q is in x,
         p=-1,    otherwise,
         where p is the returned value. */
  int l, u, m;
  l = 0; u = n-1;
  while (l<=u) /* if q is in x, q is in [l:u] */ {
    m = (l+u)/2;
    if (x[m]==q) return m;
    else if (x[m]<q) l = m+1;
    else u = m-1;
  }
  return -1;
}
```

Figure 10.7: Binary search program (another representation of Figure10.3)

It is a type of control flow graph, and is roughly equivalent to a flowchart (see section 5.2.1), but it is simplified to have only one type of vertex.

In this example, if test data that realizes two execution paths, for example 3-4-5-6-8-9-11-12 and 3-4-5-6-8-10-11-4-5-6-7, can be prepared, both statement coverage and branch coverage will reach 100%. Of course, because the program is short and its structure is relatively simple, only two test cases are obtained, and in practice the following problems arise:

(a) Even if an execution path is specified, it may be impossible to find test data that actually executes it. In general, input data that satisfies a path must be found as a solution that satisfies a logical conditional expression consisting of a combination of branch conditions on that path, but such a solution may not exist, and even if it does exist, finding the solution generally involves solving a non-linear simultaneous inequality, which is often impractical.

(b) When performing integration testing on a large program, it is common to measure the test using data used in actual operation or test data generated according to the statistical distribution of input data during operation, but in such cases, it is rare for coverage to reach 100%, and at most it usually reaches only about 80% to 90%.

(c) Conversely, even if statement coverage or branch coverage reaches 100%, it is not sufficient for testing. In fact, the binary search case above reached 100%, but both cases only performed one iteration, which is insufficient compared to equivalence partitioning and boundary value analysis in functional testing. In general, to create as comprehensive test cases as possible from the structure of a program, it is ideal to increase **path coverage**, which covers all possible execution paths, but in a program with such repetitions, the number of paths generally becomes infinite. Even if the number of loop iterations is limited to a finite range, if the control structure of the program becomes even slightly complex, the number of paths will increase explosively unless the number of iterations is limited to an extremely small value such as 1 or 2.

(d) Since structural testing creates test cases based on the structure of the program, it is not possible to generate test cases that check if a part of the program does not realize the specification.

Despite these problems, it is useful to be able to grasp the test achievement level numerically, and for this reason, various tools for measuring coverage have been developed and are actually used. It is also important to devise structural test cases, in the sense of mechanically creating test cases from execution paths, and various efforts are being made in testing organizations to achieve this.

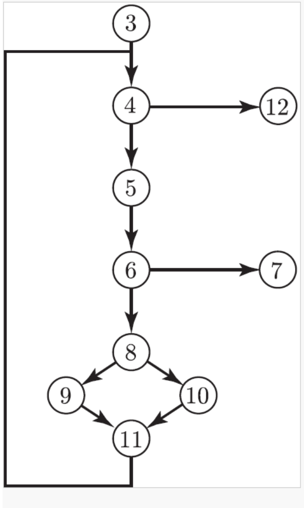


Figure 10.8: Flow graph of binary search program

Instead of using control flow-based coverage as a criterion, there is also a method of measuring coverage based on a data flow graph that shows the definition and reference relationships of data values. In general, this method produces more detailed test cases than those based on control flow.

2. **Test Case Selection from State Transition Diagrams**
State transition diagrams are generally abstracted from programs, but they represent the behavior of the corresponding programs. Generating state transition cases can be done in a similar way to selecting test cases from a program flow graph based on statement coverage or branch coverage.

   Let's use the example of bowling score calculation again. Consider a path in Figure 10.6 that executes all transitions at least once. Is there a path that executes all transitions exactly once? It comes down to the question of whether you can draw this graph in one stroke. There is a famous Euler's theorem about this. A necessary and sufficient condition for there to exist a single path starting from any vertex in a connected graph and returning to the same vertex is that for every vertex, the number of edges entering the vertex is equal to the number of edges leaving it. In the graph in Figure 10.6, this Euler's condition holds for the connected graph excluding the starting vertex. So, for example, the following path exists as an example of one that executes each transition exactly once:

   > Begin → Open → before 2nd throw → Spare → Open → Strike → Double → Double → After Strike and before 2nd throw → Spare → Strike → Open

   In the case of a state transition diagram without transition conditions, a transition sequence that realizes any path always exists, so it is easy to create the corresponding test data. In that respect, it is easier to handle than the method of selecting an appropriate path from the program flow graph and generating test data that realizes it.

### 10.2.3 Test environment and evaluation of results

There is a tool called a test bed that supports the generation of test environments, and it is used in some places. Its main functions are to generate a driver for starting the module under test and a stub that simulates the behavior of the lower module called by the target module. It supports incorporating functions to measure and log the testing process.

Evaluation of test results can be related to individual test cases or the entire test process. It is relatively easy to incorporate a function for comparing expected test results with actual results into the test environment. By saving and managing the evaluation results, they can be used for regression text (described later) during maintenance, etc.

Evaluation of the entire test is important in the sense that it provides a criterion for stopping the test at an appropriate point. In addition to the test coverage, various reliability models based on the occurrence history of errors discovered during the test process are used, but these will be explained in another section.

### 10.2.4 Testing process in the software life cycle

Testing types are generally divided into unit testing, integration testing, and acceptance testing.

- Unit testing: Testing modules as program units independently against their respective module specifications.
- Integration testing: Testing by combining units into a subsystem that executes a set of processes. This can be considered testing against design specifications.
- Acceptance testing: Testing to determine whether the system satisfies the requirements specifications.

In this way, these three types of testing are performed against each specification in the reverse order of the process of creating the requirements specifications, design specifications, and module specifications of the software life cycle (see the V-shape in Figure 2.4). Therefore, it is desirable to create a plan for each test when creating the corresponding specifications.

Among the tests performed in the maintenance process, regression testing is used to check whether the changes have had any adverse effects on existing functions. This involves re-running tests that were performed on the old version of the system, but selecting tests that take into account the management of test cases and the scope of their impact is an important issue.

### 10.2.5 Reliability Model

As the testing process progresses, errors are discovered and corrected one after another, improving the reliability of the program. However, testing is not a means that can guarantee complete removal of errors, so to some extent testing must rely on experience. Possible criteria for testing completion include the following:

1. Criteria based on test duration: When planning the project, the required test duration is determined according to the size and nature of the system being developed, and testing is completed when that period is over.
2. Standard based on labor: Similar to the standard based on period, the labor to be put into testing is determined during planning, and testing is terminated when the actual labor reaches the planned value.
3. Standard based on number of test items/number of test cases: When planning testing, the number of test items or test cases required is determined according to the scale and nature of the target system, and testing is terminated after that number of tests have been performed.
4. Standard based on test coverage: For example, testing is terminated when statement coverage exceeds 90%.
5. Standard based on number of found errors: For example, the number of errors per 1000 lines of the source program is estimated from past data, and testing is terminated when that number is reached.
6. Standard based on convergence of number of found errors: The cumulative number of errors found as the testing process progresses is observed, and testing is terminated when it is determined to be in the process of convergence.

Standards based on test period and labor are dangerous. In fact, the author has experienced the following example. The systems department of a large company measures and stores the productivity of each phase of a development project. Productivity is calculated by dividing the size of the target system (number of lines of source code) by the number of man-hours put into the process. The data shows that the productivity of the testing phase is extremely high in many projects. The reason for this is that the progress of these projects is significantly behind schedule, and since the start of system operation cannot be delayed, the testing phase is strained, the period is shortened, and the apparent productivity increases.

This situation may not be solved by changing the test termination criteria to something other than the period or man-hours, but it is likely to reflect the idea that if you cannot estimate how many errors remain in the target program at the time of finishing the test, it does not matter where you stop.

The six test completion criteria listed above are listed in order of the more ”scientific” the criteria are, the lower they are in the list. In other words, it seems most reasonable to decide to finish testing based on the convergence of the number of error discoveries. To determine the convergence of errors, it is necessary to model the process of finding errors in the testing process. In other words, the progress of the test is measured on a time axis (it can be elapsed time, cumulative time of test execution, or cumulative number of test cases), and the distribution of the number of errors found is modeled accordingly. Such a model is called a reliability model.

Many models have been proposed, but the most basic one is the following equation, which models a non-homogeneous Poisson process:

$$\mu(t) = V_0(1 - exp(-\frac{\lambda_0}{V_0}t)) \tag{10.2.1}$$

Here, $\mu(t)$ represents the cumulative number of errors found up to time t, and $V_0$ represents the total number of errors that were present at the beginning.

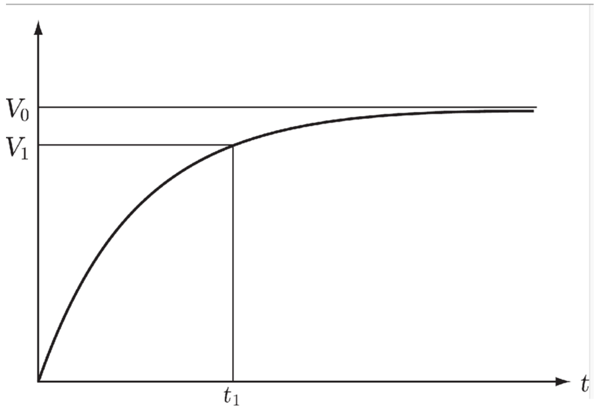


Figure 10.9: Non-homogeneous Poisson process

In Figure 10.9, the current time is $t_1$ and the cumulative number of errors found at that time is $V_1$. The estimated number of remaining errors is $V_0 - V_1$.

At an intermediate stage of the testing process, the cumulative number of errors up to that point is plotted against time, and the model is fitted to estimate the parameters $\lambda_0$ and $V_0$. The number of remaining errors can be estimated by the difference between $V_0$ and the number of errors found up to now. In addition, if a target number of errors to be found is set, the test time required to reach that number can be estimated. The differential form of the equation (10.2.1) is the following equation

$$\lambda(t) = \lambda_0\, exp(-\frac{\lambda_0}{V_0}t) \tag{10.2.2}$$

The derivative $\lambda(t)$ of $\mu(t)$ represents the error intensity, i.e., the strength of the probability that an error will be found at time t (Figure 10.10).

There have been many other proposals for reliability models. For example, a model that approaches an upper limit is not realistic, so one that uses an unbounded log-Poisson process, and one that realizes the shape of the reliability curve, assuming that most real data has a low error detection rate in the early stages, a high error detection rate in the middle, and a low error detection rate in the later stages, creating an S-shaped curve. In addition, a model has been devised in which, in reality, when an error is detected, code is inserted to correct it, but new errors are introduced, so the total number of errors increases. However, no matter how sophisticated the model is, the question remains as to whether it will fit the

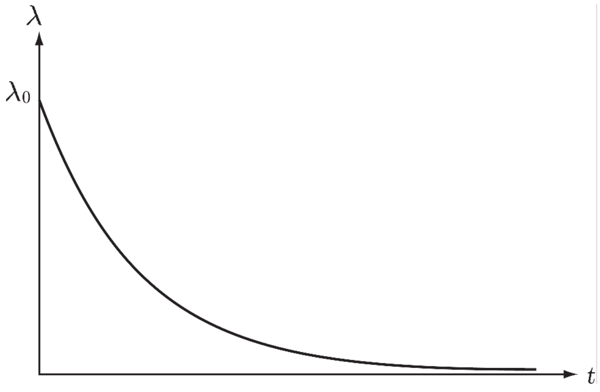


Figure 10.10: Error Intensity

development organization. It is desirable to use a simple model to gain experience in its use and estimate parameter values with a certain degree of validity before testing begins.

### 10.2.6 Test-driven Development Process

There are some development processes and methodologies that place special emphasis on testing.

**cleanroom software engineering** [90, 110]

The cleanroom method combines mathematically rigorous design methods and statistically-based testing methods to develop highly reliable software. Testers and system developers are strictly separated, and developers are not even allowed to run the programs they develop through a compiler. Testing is performed using test data generated by statistical sampling based on the input data distribution during operation.

**extreme programming** [10]

The agile process model described in the 2.4 section, and especially extreme programming, emphasizes writing test cases before writing a program. In other words, a test program that performs the expected behavior is written first. Since the program that realizes the function has not yet been created, a problem occurs when it is ”run”. The program is modified to resolve the problem. If the test produces the desired results by continuing this operation, that part of the program is complete. In this case, a unit testing tool such as JUnit is used.

### 10.2.7 Static Analysis of Programs

As shown in Figure 10.1, there are two types of program verification: dynamic verification, which is represented by tests, and static verification, which statically analyzes the code without executing the program. We have already mentioned type checking as a static verification technique, but let’s look at other methods.

#### Data Flow Analysis

Data flow analysis technology was developed because it was useful for compiler optimization. Its basic function is to track the relationship between the to variables and the reference of those values. As an algorithm, the iterative method described in a paper published by Gary Kildall in 1973 is well known and is still effective today[81].

In compiler optimization, it can be used for constant propagation, detection of common expressions, and register allocation. In program analysis, it can be used for detecting uninitialized variables, discovering redundant assignment statements, and determining input and output variables.

**Symbolic Execution**

Symbolic execution is a technique in which, instead of giving concrete input data to a program and executing it, input values are given as symbols, such as $\alpha$, $\beta$, and the execution result is expressed as a mathematical formula containing those symbols. It has a relatively long history, with Robert Boyer's system Select, James C. King's Effigy, William Howden's Dissect, and Lori Clark's system being announced in succession in 1975-76. The paper explaining how King et al. used symbolic execution to prove correctness is available in Japanese translation by the author [144, 145]. The original paper was publised in *ACM Computing Surveys (CSUR), 1976* [56].

Due to technical difficulties in handling branching in execution paths and in processing arrays and pointers, it was not put to full-scale practical use, but it was used as a tool for proving and testing the correctness of programs. In the 2000s, ”concolic testing” (concolic is a combination of concrete and symbolic), which combines symbolic execution and testing using normal data, was proposed and attracted attention.

Java Pathfinder by Willem Visser et al., which was mentioned in the 8.4 section as an example of a model checking system for programs, is known for generating test cases using symbolic execution. This is also an example of combining symbolic execution and normal testing in another sense.

**Code clone detection**

There are many ways to use the analysis of source code, and one of them is code clone detection. Code clones are identical or nearly identical code fragments scattered in multiple places in a program. This phenomenon occurs when programmers repeatedly copy and paste parts of code. This not only unnecessarily increases the size of the program, but also reduces the maintainability of the program. When a change is made to one clone, the other clones should be changed in sync, but this is often overlooked.

Tools that automatically detect clones have been developed. In particular, CCFinder[75], developed by Toshihiro Kamiya at Katsuro Inoue's laboratory at Osaka University, is well known and widely used.

### 10.2.8 Fault tolerance

In addition to removing errors and preventing faults through verification techniques, it is also important to have techniques that prevent faults from becoming failures even if they do occur during operation. This is generally called fault tolerance. However, fault tolerance is a technology that was originally devised for hardware, and it is difficult to say that it has been developed independently for software. Two representative examples are the recovery block method and N-version programming. In a sense, these can be considered applications of testing techniques.

**Recovery block method** In the recovery block method, N independently developed modules that perform the same function are prepared. It is also assumed that an acceptance test can be prepared that can determine whether the function is working correctly at run time. A sequence is set for N modules, and at execution time, module 1 is executed first and the result is subjected to an acceptance test. If the result is successful, proceed as is. If it is unsuccessful, module 2 is executed and also subjected to an acceptance test. If module N is still unsuccessful, it is judged to be abnormal and special processing is performed for that purpose.

Not only is it not easy to prepare N independent modules, but the extra load at execution time is large, so it seems that there are not many cases in which this is actually implemented.

**N-version programming** As with the recovery block method, N independently developed modules that perform the same function are prepared. If the results match, they are output as is, but if they do not match, a majority vote is taken among them to determine the result. It is well known that a similar process was performed in the control system on the Space Shuttle.

In the case of the Space Shuttle, a majority vote is taken at run time, but there is also a method of taking a majority vote at test time to determine the correct module. In the simplest case, two independent modules are created and their behavior is compared. With $N = 2$, there is no majority vote, but if there is a difference between the two, a different method will be used to determine which is correct. In principle, it is also possible to set N to 3 or more and bring in a

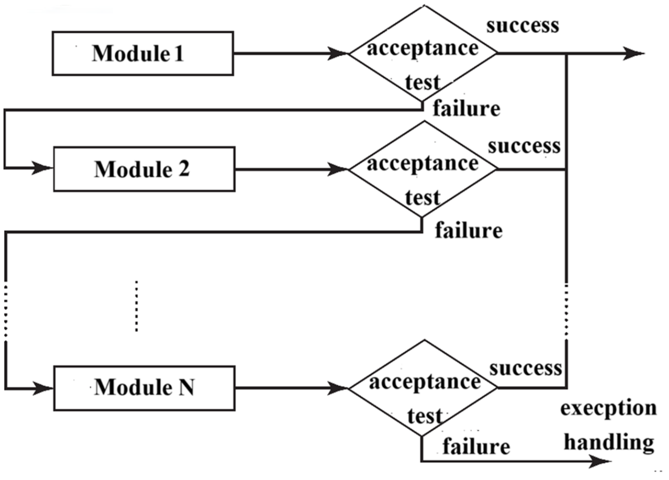


Figure 10.11: Recovery block method

majority vote at test time. In addition to comparing whether the results are correct, it is often the case that non-functional requirements such as performance are compared and the better one is selected.

As with the recovery block method, it is difficult to prepare N independent modules. Even if they are created by independent development organizations, analysis has shown that there is a correlation in the locations where errors occur.

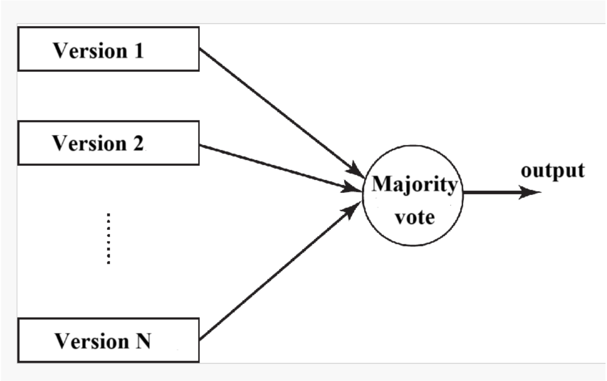


Figure 10.12: N-version programming

### 10.2.9 Verification of AI software

In the ”Software types” section of the 1.1.3 section, we mentioned the following characteristics of AI software.

- It is difficult to judge whether the behavior is ”correct,” and the results are often not explicitly expected from the specifications.
- The development process for a system using machine learning has two stages: the learning stage and the application stage using the learning results. When there is a problem with the operation, it is difficult to determine which stage is the problem.

- Software execution is low in reproducibility.

These characteristics suggest the difficulty of verifying AI software. In particular, because there are two stages, learning and application, various new attacks have been discussed, such as tampering with the training data used for learning to induce incorrect behavior. In addition, attacks that deceive image recognition systems, especially during the application stage, have been assumed. There have been reported cases where a small amount of graffiti that is not noticeable to humans was added to road signs, causing stop signs to be mistaken for speed limit signs, or right turn signs to be mistaken for stop signs, etc., providing a warning in this age of autonomous driving [44].

In addition, if there is a problem with the learning data of machine learning, or if care is not taken in how it is applied, it may cause human rights issues. For example, in 2016, Microsoft began offering the AI chat robot "Tay", but was forced to suspend it just 16 hours later. The reason for this was that the robot was making a lot of racial hate speech, and the training data contained data that was likely to cause such speech.

## 10.3 Specification Verification Techniques

### 10.3.1 Review, Inspection, Walk-Through

The method of inspecting design documents and program code was proposed by Michael Fagan of IBM, and became famous after being put into practice at IBM and achieving good results. [45]. Daniel P. Friedman and Gerald Weinberg's book, which specifically describes review, inspection, and walk-through [50] is often used as a reference. According to this, for example, code review is characterized by the following:

- A team of four people, the chairperson, the program designer, the test expert, and the person who wrote the program
- Materials are distributed in advance
- Any errors found are recorded without fail
- The programmer reads the code aloud
- Checks using a checklist
- The results are a document with a set format
- The time required is 1-2 hours
- Errors are not corrected during this process

Walk-throughs are similar, but they differ in that they define simple test cases and humans simulate the computer's execution. Also, the term review is often used as a general term for inspections and walk-throughs.

### 10.3.2 Applications of Model Checking

In the 8.4 section, we outlined model checking. Here, we introduce specific examples of its application as one of the verification techniques for specifications.

**Model Checking of EJB Architecture**

As an example of model checking, we introduce a case study by Shin Nakajima and the author in which the validity of the architecture of Enterprise JavaBeans (EJB) was analyzed using model checking techniques.[96, 147]

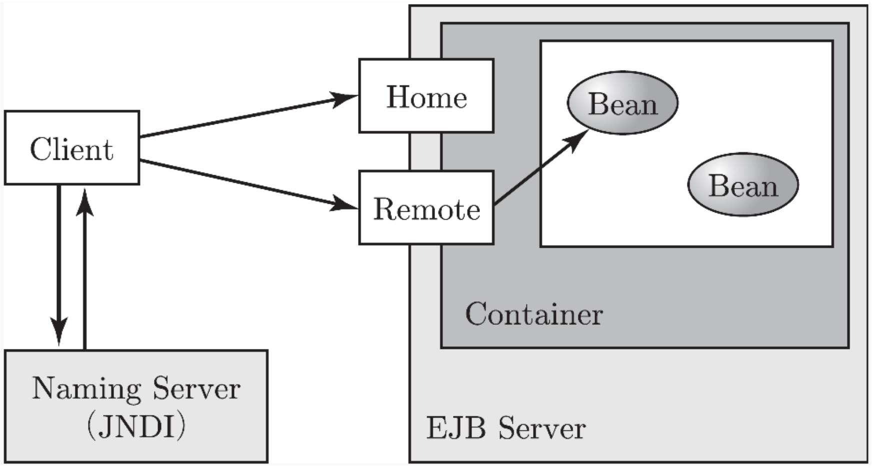


Figure 10.13: Overview of EJB Architecture

**EJB Component Architecture** EJB is a component architecture for distributed business applications. The EJB architecture has the structure shown in figure 10.13.

In EJB, components are treated as units called beans. When a client requests a bean using a name server called JNDI, it obtains an object reference called Home. Home creates a bean object, creates a Remote as its proxy object, and passes the reference to the client. Subsequent processing requests issued by the client are received by the Remote and delegated to the bean. Beans run under an operating environment called Container. The container uses the functions of the EJB server to perform runtime services such as bean evacuation, persistence, and automatic deletion. There are two types of beans: entity beans and session beans. Entity beans reside in a persistent storage device and are shared by multiple clients (entity beans were removed from the specification in the subsequent EJB3.2 specification, but the discussion here is based on EJB1.1). Session beans are generated for a single client and generally access multiple entity beans to execute a series of transaction processes. Figure 10.14 illustrates this relationship.

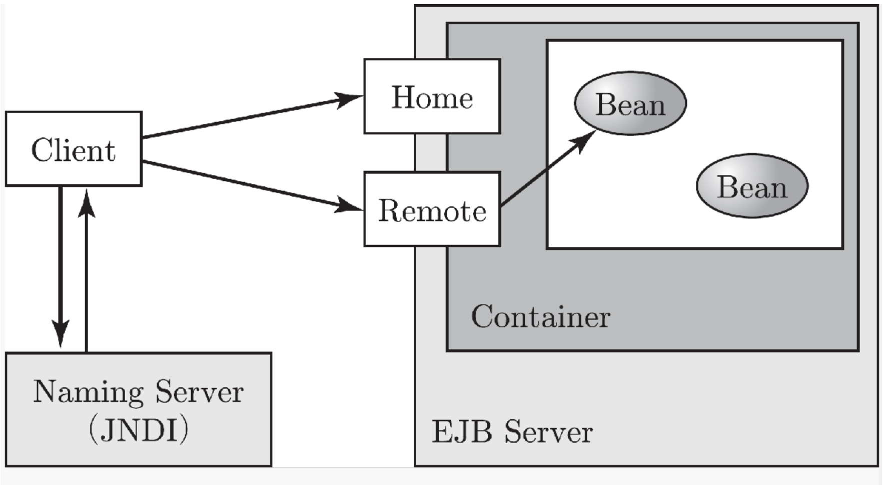


Figure 10.14: Relationship between session beans and real beans

Real beans behave as shown in Figure 10.15. All of the labels attached to the transitions here are real bean methods, and state transitions occur as shown in the figure when the methods are invoked. A bean is created with ejbCreate and deleted with ejbRemove. ejbStore and ejbLoad are methods related to persistence processing, and write to and read from persistent storage, respectively. ejbPassivate and ejbActivate are methods related to evacuation processing, and correspond to evacuation from and recovery from main memory, respectively. Business methods are methods that realize the business functions specific to this bean.

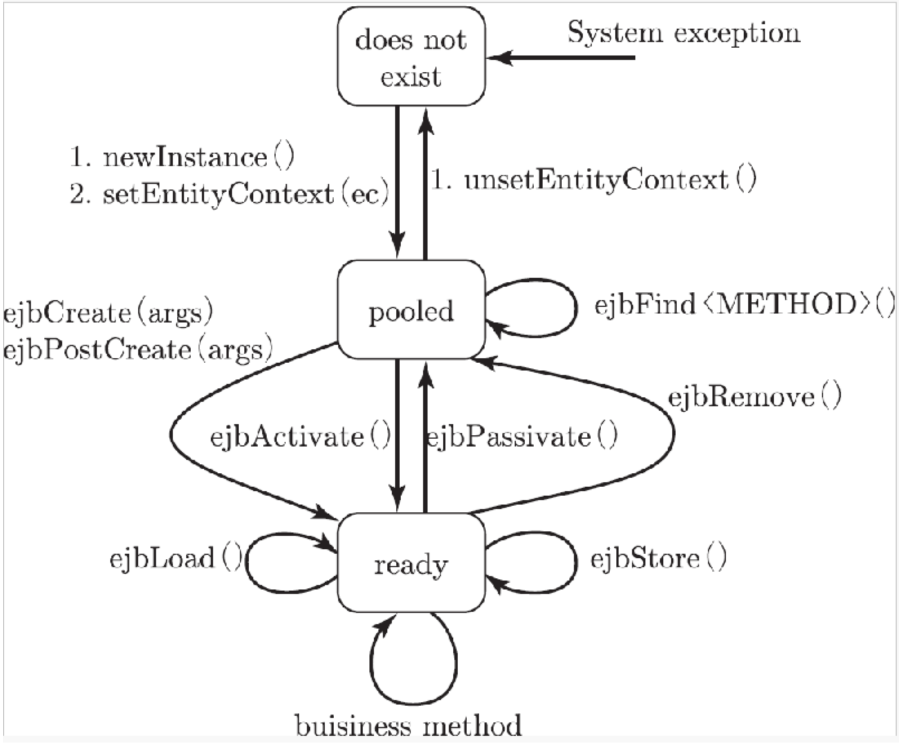


Figure 10.15: Life cycle of an entity bean

**SPIN model checking system** SPIN is a model checking system developed by G. Holzmann at AT&T Bell Labs[61]. SPIN was used to analyze the component architecture of EJB. In SPIN, models are written in a specification language called Promela. The properties to be checked are written in linear temporal logic (LTL).

The components of a description in Promela are processes and channels between processes. Here is an example of a description of an entity bean.

```
proctype EntityBean ()
{
endLoop:
do
:: mthd?ejbActivate -> if :: retv!Void :: excp!SysError fi
:: mthd?ejbPassibate -> if :: retv!Void :: excp!SysError fi
:: mthd?BM -> if :: retv!AppError :: excp!SysError fi
...
od
}
```

Promela uses Dijkstra-style guarded instructions as control structures. The do statement for guarded instructions has the following syntax:

do $B_1 \rightarrow S_1 :: B_2 \rightarrow S_2 :: \cdots :: B_n \rightarrow S_n$ od

Here, $B_i$ represents a condition expressed by a logical formula, and $S_i$ represents an executable statement. What this means is that if any of the conditions $B_1, B_2, \cdots, B_n$ are evaluated as true, one of them is non-deterministically selected, and if it is $B_i$, the corresponding statement $S_i$ is executed and the loop is repeated. If all conditions become false, the loop ends.

The if statement of the guarded instruction has the same syntax

if $B_1 \rightarrow S_1 :: B_2 \rightarrow S_2 :: \cdots :: B_n \rightarrow S_n$ fi

, which means that if any of the conditions $B_1, B_2, \cdots, B_n$ are evaluated as true, one of them is selected nondeterministically, and if it is $B_i$, the corresponding statement $S_i$ is executed once and the program terminates.

Note that if there is no guard, it is equivalent to $\mathit{true} \rightarrow S$. In EntityBean programs, if statements without guards are used, which means "execute one of them nondeterministically."

Processes interact by exchanging messages through channels. The concept of channels here is the same as channels in CSP and the $\pi$ calculation. In other words, c!e means to send the value of expression e to channel c, and c?v means to store the value received from channel c in variable v. Note that in the EntityBean program, `mthd?ejbActivate` and similar are used as guards, which represent a pattern match that there is a message called ejbActivate at the top of the queue for channel mthd, and if there is a match it will be true. Here, ejbActivate, ejbPassivate, and BM are all constants. BM refers to a business method.

Next, we show an EJBObject defined as a Promela process. EJBObject is a proxy object for the bean provided by Remote.

```
proctype EJBObject()
{
   chan returnValue;  chan exceptionValue;  short value;
progressLoop:
endLoop:
   do
      :: remote?remove,returnValue,exceptionValue
        -> { request!reqRemove;  retvalFC?value
              -> returnValue!value;  goto endTerminate }
            unless { exceptFC?value ->
                        exceptionValue!Error;  goto endTerminate }
      :: remote?BM,returnValue,exceptionValue
        -> { request!reqBM; retvalFC?value -> returnValue?value }
            unless {exceptFC?!value ->exceptionValue!Error }
   od;
endTerminate:
   skip
}
```

Here, we omit the definition of the channel remote, which receives a message consisting of three values. The first is the message content, and the last two are the channels to send back values. The message "remove" received is a constant with the same meaning as ejbActivate, and whereas ejbActivate is the name of a method for an actual bean, this is the method name defined for EJBObject.

The behavior of the container that holds the actual bean is complex. Figure 10.16 shows its state transition diagram.

Based on this behavior, the container is also modeled in Promela. It is about 250 lines of Promela code.

**Property verification** Describe the property you want to verify in LTL.

Since there is no branching of computation paths in LTL, there is no need to use the quantifier **A**, which says "any computation path is taken," or the quantifier **E**, which says "there exists a computation path."

The following are used as temporal operators in LTL in SPIN.

- **G** *p*  p always holds.

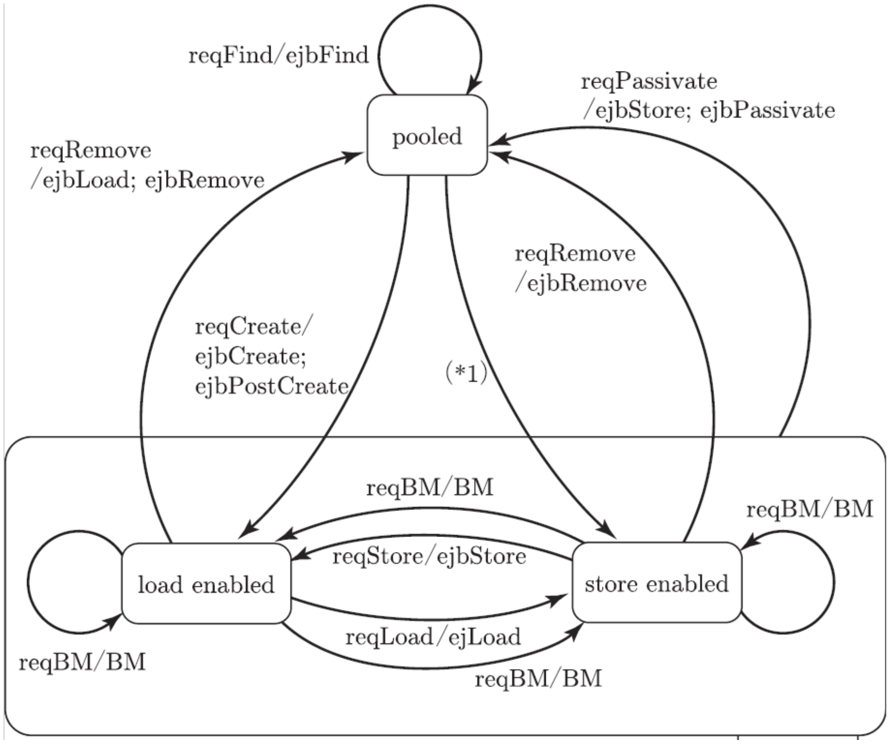


Figure 10.16: Container behavior

- $\mathbf{F}\,p$ p will hold someday.
- $q\,\mathbf{U}\,p$ p will hold someday, and until then q always holds.

For example, here is an example of a verification description for the behavior of an entity bean.

1. If a client requests BM startup for a pooled bean, ejbActivate will be started at some point.

$$\mathbf{G}\,(pooled \wedge BM\_Client \rightarrow \mathbf{F}\,ejbActivate) \tag{10.3.1}$$

2. If BM is executed after ejbActivate, ejbLoad will be executed before it.

$$\mathbf{G}\,(ejbActivate \wedge \mathbf{F}\,BM \rightarrow (\neg BM\,\mathbf{U}\,ejbLoad)) \tag{10.3.2}$$

3. If the EJB server does ejbPassivate after BM, it must execute ejbStore before that.

$$\mathbf{G}\,(BM \wedge \mathbf{F}\,ejbPassivate \rightarrow (\neg ejbPassivate\,\mathbf{U}\,ejbStore)) \tag{10.3.3}$$

SPIN converts such LTL expressions into Büchi automata. Büchi automata are a type of finite-state automaton that are almost the same as those that accept regular expressions, but in order to accept infinitely long strings, an accepting state is defined instead of a final state. In other words, when an accepting state is reached, the input of the string does not necessarily have to be terminated, and the transition can continue.

For example, if the simplest liveness formula

$$\mathbf{AF}Q$$

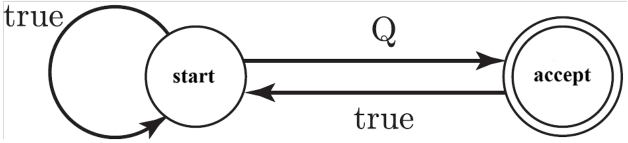


Figure 10.17: Conversion of LTL expressions to Büchi automata

is converted into a Büchi automaton, it will look like Figure 10.17.

The LTS describing the model can be regarded as an automaton as it is. A product automaton is created by combining this with a Büchi automaton

created by converting the negation of the LTL formula to be verified. If the product automaton does not have an accepting language, the property to be verified is satisfied. If there is an accepting statement, that statement represents a counterexample that violates the property to be verified. This is the effect of combining the negation of the LTL formula.

When the EJB model is synthesized with a client model, and an automaton created by the negation of the LTL expression to be verified is further synthesized, and the above three properties are verified with SPIN, it is confirmed that all of them are satisfied.

On the other hand, it can be seen that the property of liveness of the type in which the corresponding operation of the entity bean will always be executed at some point when the client issues a request is not necessarily simply satisfied. For example,

4. When a client requests remove, ejbRemove will be invoked at some point.

$$\mathbf{G}\ (\textit{remove} \rightarrow \mathbf{F}\ \textit{ejbRemove}) \tag{10.3.4}$$

When we try to verify this, we find that it is not satisfied. The counterexample shows that a livelock can occur, where ejbStore and ejbLoad alternate infinitely, preventing progress. This livelock can be removed by making some fairness assumptions, but the equation (10.3.4) is still not satisfied. As can be seen from the figure 10.15, a bean transitions from the ready state to the pooled state either by ejbRemove or by ejbPassivate. ejbRemove is invoked in response to a client's remove request, while ejbPassivate is invoked by the Container as a runtime evacuation service. If the Container invokes ejbPassivate independently after the client issues a remove, ejbRemove will lose the opportunity to be invoked. This situation is not anticipated in the EJB1.1 specification and is considered a flaw in the specification.

In this way, model checking techniques can be used to analyze what properties a model does or does not satisfy, and in what cases it does not satisfy them.

# Chapter 11

# Software Maintenance and Evolution

In the traditional software development process, it was thought that the development phase would switch to the maintenance phase once development was completed and the system was up and running and used by users. However, in a process such as agile, where updates are constantly being made, the boundary between development and maintenance is not clear. The traditional concept of maintenance can be seen as having been replaced by the view of software evolution. In this chapter, we will first give an overview of traditional ”maintenance”, and then look at software evolution.

## 11.1 Software maintenance

### 11.1.1 Characteristics of software maintenance

The term ”maintenance” for software, like many other terms, is a repurposed term that has been used for hardware products. However, software maintenance differs in many ways from hardware maintenance, and it could be argued that it is inappropriate to use the same term.

The three major characteristics of software maintenance are as follows:

1. It is not uncommon for software to have some kind of defect when it is delivered or shipped, and both the manufacturer and the user anticipate the possibility of defects to some extent, and the work of correcting such defects is called maintenance.

   Defects are rarely discovered in hardware products after shipment and as this is considered an exceptional situation, correcting such defects is called by other terms, such as a recalland and distinguished from maintenance.

2. Software does not change over time, such as wear and tear on parts.

   Therefore, no maintenance work is required due to wear and tear on parts.

3. It is much easier to improve, change, and expand the functions of software than it is with hardware. Therefore, functional changes and expansions are usually included in maintenance. Work that would be called new product development for hardware products is often called maintenance for software.

In other words, while hardware maintenance refers to the deterioration of parts over time, there is no software maintenance in that sense. Instead, it refers to the removal of simpler latent defects or, conversely, the modification and expansion of more advanced functions.

### 11.1.2 The magnitude of maintenance work

Due to these characteristics of software maintenance, the importance of maintenance is high, and the cost required for it accounts for a large proportion of the total system cost. The ratio is often estimated as 70% by Boehm, but a well-founded figure based on research is 50% by Bennet Lientz & Burton Swanson[84]. In spite of this, maintenance is treated relatively lightly in software engineering, and in practice, it is not as strategically conscious as development, it lacks planning, and it is difficult to allocate capable personnel. However, while development creates the flow of software, maintenance relies on stock. Moreover, unlike other assets, this stock is constantly required to change and has the characteristic of actually changing. Therefore, as software stocks accumulate, both on a micro-base (a company) and a macro-base (a society), maintenance work becomes an increasingly big problem.

### 11.1.3 Maintenance process

Traditional software life cycle models are development-centered, and the maintenance phase is often considered as an added phase at the end of the life cycle, as is typical of the waterfall model. However, it is necessary to turn our attention to the longer-term evolution process of software.

In fact, the following factors suggest that a life cycle model that further combines development and maintenance should be considered.

1. Even in development projects, there are very few cases where the development team is completely inexperienced in the target domain and creates everything from scratch.

2. Maintenance work also follows the same process as development: analysis and definition of (change) requirements, design, programming, and testing. In other words, the maintenance process can be seen as a normal development process with the addition of the task of understanding the existing system.

3. In long-term software development, user requirements and design changes often occur during development. This is equivalent to the occurrence of maintenance work for requirements specifications and design specifications during the development process. Conversely, the continuous software update process can be seen as a state in which development is postponed until maintenance.

Agile processes can also be seen as aiming for a process of perpetual development that removes the barrier between development and maintenance. Furthermore, there is a trend to remove the boundary between development and operation, such as DevOps.

Some maintenance work has characteristics that are not found in long-term development, such as quickly making changes to a running system due to a failure or user request to maintain the service level. These are closer to operation than development.

It should also be noted that the boundary between the development/maintenance environment and the operation environment is disappearing. As already mentioned, in a reactive system operating in a distributed environment, ”maintenance” and adding new functions will have to be done in parallel with operation. In other words, the previous step of creating and verifying a program in a development/maintenance environment and then migrating it to an operational environment is no longer realistic.

### 11.1.4 Classification of maintenance

Various classifications of maintenance have been proposed, but the most commonly referenced are the following three classifications by Swanson[84].

1. Corrective maintenance: Maintenance to correct errors

2. Adaptive maintenance: Maintenance to adapt the system to changes in hardware, OS, peripheral devices, data specifications, etc.

3. Perfective maintenance: Maintenance aimed at expanding functions and improving efficiency

A fourth type of preventive maintenance is sometimes listed, which improves the structure of the system to increase reliability and maintainability. Some also argue that maintenance to expand functions should be called adaptive maintenance, as it responds to changes in the usage environment, and that maintenance to improve efficiency and maintainability without changing functions should be called perfection maintenance.

### 11.1.5 Maintenance Organization

There is some debate about what kind of organizational structure maintenance should be. Basically, there are two options: to create an organization dedicated to maintenance that is separate from the development team, or to have the development organization perform maintenance in parallel. In reality, there are few cases where maintenance is a completely separate organization. However, there are also reports that the effects of having a separate organization are great.

Even if there is no separate organization for maintenance, there are many cases where personnel are divided into development and maintenance. In some cases, the division of responsibilities is fixed for a long period of time, and in other cases, it is rotated for a relatively short period of time. It is rather rare for the same personnel to perform both development and maintenance work at the same time.

There are various levels of independence for maintenance, from a completely separate organization to parallel maintenance and development even at the personnel level. The higher the independence of maintenance, the clearer the division of budgets, personnel, and resources between maintenance and development, and the higher the quality of planning and management. Thus, the advantage of the higher independence is:

1. Maintenance involves responding to irregular requests from users, and the workload is not uniform, so if developers also take on both roles, it can have a significant impact on the progress of the development project, but if maintenance is independent, this problem can be avoided.

2. It is easier to respond quickly and precisely to maintenance requests from users.

On the other hand,

1. The burden of transferring knowledge between development and maintenance is high, and problems tend to occur due to poor transition.

2. The morale and technical ability of maintenance personnel is likely to decline.

In particular, the issue of motivating personnel is a headache for managers. The relatively common method of assigning new people to maintenance has been criticized for reducing the productivity of maintenance work, which is inherently more difficult than development, while there is also the danger that assigning capable people to maintenance work will result in a decline in morale and their skills will not be utilized. In this respect, too, agile development can be argued to have an advantage.

### 11.1.6 Maintenance techniques and tools

Many parts of maintenance work are similar to development. Therefore, the processes of analysis, design, programming, and testing use techniques and tools that are common to new development. On the other hand, there are also techniques and tools that are unique to maintenance. Here are some representative ones.

#### Ripple effect analysis

Since maintenance involves modifying parts of existing software, it is a fundamental requirement to understand the scope of the impact of the change or addition. At the micro level, for example, changing the value of a program variable in one place can affect other parts of the program that reference that variable. On a slightly larger scale, when one module is changed, all other modules that call that module are affected. Analyzing such effects is called ripple effect analysis.

Methods such as control flow analysis and data flow analysis can be applied to ripple effect analysis, and tools based on these have been developed. However, there are limitations to the scope of analysis, and the use of these tools has not progressed much.

#### Regression testing

Regression testing, described in the 10.2.4 section, is a testing technique used in maintenance work. Tests to verify the results of changes made during maintenance are first performed to ensure that the changed parts have been made correctly.

Equally important is to ensure that the parts that were not changed still function normally. To do this, it is necessary to rerun some of the tests that were applied to the system before maintenance and to verify that the results are the same as before. This type of testing is called regression testing.

In regression testing, it is not necessary to repeat all of the original test cases. Therefore, it is possible to select the necessary tests using the results of the ripple effect analysis described above. However, determining the minimum required test coverage accurately is not so easy. Regression testing tools help you manage past test data and rerun it. Comparing test results with past results can also be automated with the tools.

**Program Slicing**

The statements that affect the value of a variable in a statement of a program at execution time are limited, not all statements in the program. There are two ways to affect it: by the flow of data definition reference relationships, and by the flow of control such as if statements and while statements. The fragments of these statements are called slices, and the method of extracting them is called program slicing. The concept and method of slicing were proposed by Mark Weiser, who is also well known as the proponent of ubiquitous computing [141]. There are two methods of program slicing: static analysis of a program and dynamic execution.

The results of slicing are used to extract the parts of the program that are affected by changes made to the program for maintenance, and to perform tests on them.

### 11.1.7 Maintenance Strategy

Maintenance is an activity that involves the entire long life cycle of software, and therefore in a sense requires more strategic planning and management than development.

When it comes to maintenance strategies, the following two are the major decision-making targets (Victor Basili[9]).

1. Deciding whether to continue maintenance or to discard the current system and create a new one
2. Deciding on the criteria for each maintenance task. Specifically,
    - Simple type: A strategy to minimize the burden of immediate maintenance tasks. In other words, patch-type maintenance
    - Structural preservation type: Maintenance that places the highest priority on keeping the software structure as well-organized as possible

The simple type minimizes short-term costs, but in the long term, maintenance costs may increase exponentially due to structural deterioration of the software. The structural preservation type requires short-term costs and time, but for long-lived software, it is effective in greatly reducing future maintenance costs. As a strategy, you can choose a suitable middle ground between these two extremes. In other words, after a certain amount of changes have been accumulated through repeated simple maintenance, restructuring is performed to preserve the structure. The refactoring emphasized in extreme programming is based on this idea.

This decision is related to the decision of maintenance or rebuilding (1). In general, if simple maintenance is continued, structural deterioration will force the software to be rebuilt sooner. However, if structural maintenance is performed, the software's lifespan will be extended, and the need for rebuilding can be delayed. However, as described in the 11.2.2 section, structural deterioration is not the only factor that causes the need to rebuild software. For example, changes in hardware or OS, requests from users for functional changes or extensions, or responses to changes in legislation or business environments often force the software to be rebuilt. In such cases, it may be that simple maintenance is cheaper in total cost since the software will be rebuilt anyway. In other words, an appropriate strategy is to perform simple maintenance for software that is expected to have a relatively short lifespan due to factors other than structural deterioration, and to perform structural maintenance for software that is expected to have a long lifespan.

## 11.2 Software Evolution Process

Software continues to change, and this change is called evolution here. ”Evolution” here means that those that are best fit to the environment will survive in the Darwinian sense, and is different from ”progress.” Software products that survive as a result of evolution are not necessarily the best or most optimal.

### 11.2.1 Basic Characteristics of Software Evolution

The phrase ”evolving” software essentially refers to the same phenomenon as software maintenance, but has a more positive tone.

Furthermore, ”evolution” and ”development” are intransitive verbs, while ”maintenance” is a transitive verb. In other words, software evolves and develops, but it is maintained by people. The former perspective suggests a method for objectively observing the evolution of software, while the latter keeps in mind that software is an artifact and that maintenance is a task performed by humans.

The factors that cause software to evolve include external factors such as expansion or change of requirements, changes in the system environment (e.g., hardware, OS, middleware, network, and other related systems), and changes in the usage environment, as well as internal factors such as reorganization of the internal structure to improve maintainability, reliability, and performance. In addition to these diverse factors, software is more easily tweaked than hardware, so it continues to evolve.

There are many different forms of evolution. In particular, when looking at the scale and time of evolution,

1. Diversity of scale: Even when targeting one system, partial modifications may involve the change of one sentence or even one character. Large-scale modifications involve changes to several tens of percent of the entire system. When the ratio exceeds a certain level, it becomes something that should be called a rework rather than a correction, but rework can also be seen as a form of evolution. Also, the unit of evolution is not limited to the system. Components often evolve as a whole library or individually.

2. Diversity in time: The lifespan of software can range from a few days to several decades. In general, software continues to evolve throughout its lifespan. Also, when a piece of software reaches the end of its lifespan and is reworked, evolution can be seen as continuing across generations. The period required to realize each change is also very short, for example, when changing a single sentence as an emergency measure, but in the case of large-scale changes, the work is equivalent to new development and it is not uncommon to take several years. Also, the timing of the change can be either to stop the system, reconfigure it, and restart it, or to make the change dynamically without stopping operation. Many software systems today are reactive and in constant operation, and the latter is becoming increasingly necessary. In terms of evolution over time, the former can be seen as discrete, and the latter as continuous.

### 11.2.2 Software evolution model

**Beladay & Lehman’s Law of Evolution**

Les Belady & Manny Lehman’s well-known research empirically shows the software evolution process[12, 83]. They analyzed the evolution of IBM’s OS/360 and proposed the following law of software evolution:

1. Law of continuous change
   A system in use continues to change until it becomes cost-effective to freeze the changes and rebuild it.

2. Law of increasing entropy
   The entropy (structural disorder) of a system increases over time unless deliberate efforts are made to prevent or reduce it.

3. Law of statistically smooth evolution
   The evolution of system attributes appears locally as a random process, but statistically it is a combination of regular fluctuations and smooth long-term trends.

They argued that software generally has dynamic properties in that it continues to evolve functionally but structurally deteriorates and eventually dies.

### Model of evolution across generations

Unlike Belady & Lehman, there is also a study by Yosuke Torimitsu and the author that traces the evolutionary process across generations associated with the reworking of application software[132, 169]. The paper focuses on the fact that commonly used software is rebuilt after a certain period of time, i.e., the process by which software continues to evolve over the long term through successive generations, and summarizes the following findings:

1. The lifespan of one generation of software, that is, the period from when it is put into practical use until it is replaced by a new one through reconstruction, is about 10 years on average.

   However, there is a large variation in lifespan.
2. Small-scale software has a shorter lifespan.
3. Systems for administrative tasks (e.g. accounting or personnel systems) have a longer lifespan than systems for operational tasks (e.g. production management or sales management systems).
4. The size of software increases as a result of reconstruction.
5. The determining factors for reconstruction are complex. In the majority of cases, the reason for reconstruction is to meet users' requests for functional changes and expansions.
6. Although deterioration of maintainability is an important reason for reworking, it is not the dominant one. Moreover, the lifespan of software that is reworked for this reason is long, on average.

### Object Evolution Model

There are two types of evolution in object-oriented systems: at the object level and at the system level.

**Object-level Evolution** Objects are inherently dynamic. At the most basic level, objects are first generated, exchange messages with each other, and change their state to function, then disappear. Furthermore, when considered over a longer time scale, the internal structure of objects, their roles as seen from the outside, and the organizational structure composed of objects continue to grow and evolve.

Here, the evolution of the system and the evolution of the objects do not necessarily synchronize. Even after a system has ended its life, some of its objects may move to another system and continue to evolve. This can be compared to the difference between the evolution of a species (individual) of a living organism and the evolution of DNA. From this perspective, research is being conducted to empirically analyze the evolutionary process of objects and to construct models of the evolutionary process.

The essence of Darwin's theory of evolution can be reduced to two mechanisms: one is replication and the other is natural selection. A process that includes both of these can be considered evolution, whether it is a natural phenomenon or an artificial phenomenon. Richard Dawkins proposed the term meme as a counterpart to genes to explain social phenomena in which various artificial concepts are widespread [39]. According to Dawkins, a meme is a unit of cultural replicator, and examples include songs, ideas, catchphrases, fashion, how to make a pot, and how to build an arch. Many people have developed this concept of meme.

A representative example is the bold development by Susan Blackmore [19]. The evolution of software components clearly has the characteristics of meme-based evolution.

**Observational Examples of Object Evolution** Tayako Nakatani and the author have reported the results of an empirical analysis of object-level evolution and development[97, 131], which focuses on three types of application systems. For each, there are four or more versions of data, and quantitative measurements are made on three layers: system, class, and method. For the system layer, the number of classes and the depth of the class tree; for the class layer, the number of methods, the number of instance variables, and the number of subclasses; for the method layer, the number of primitive code lines, etc. The aggregate values and averages of the data from the layer one level below each layer are also measurements of the layer above. For example, the total and average of the number of lines of methods are measurements of the class layer.

The resulting observations are summarized as follows:

1. Basic statistics and distribution shapes are relatively stable over time.
2. On the other hand, there are cases where data with exceptionally large singular values exists. These are likely to reflect design flaws or exceptional design decisions.
3. Although most data shows a tendency to increase upwards over time, the change is not continuous. There are periods of rapid increase and periods of slow change. Discontinuous changes often indicate architecture-level changes.
4. A unique metric that characterizes class trees was discovered. As expected, the number of lines of code and the number of methods per class are strongly correlated. When linear regression is performed on the number of lines and the number of methods, clear linear regression is observed between classes belonging to the same class tree. The regression coefficients differ significantly for each class tree and are stable over time, i.e., during the evolution process, so they are recognized as metrics that characterize class trees.
5. The frequency distributions of the number of lines of code and the number of methods, which represent the size of a class, are stable not only in average but also in distribution shape during the evolution process. It was statistically tested that the distribution shape fits a negative binomial distribution. The negative binomial distribution is characterized by two parameters, and analyzing the fluctuations of these parameters makes it possible to characterize the properties of a system and the development process.

## 11.3 Reuse and reconstruction

It has long been said that reuse technology is the key to increasing the productivity of software development. However, it was only in the 1990s that many practical examples were accumulated, and discussions of methodologies and reports of support tools and examples began to be actively conducted at academic conferences and other venues.

The first thing that comes to mind as an object of reuse is program code, and the most reported examples are those. However, requirements specifications and designs can also be reused, and the software development and maintenance process itself is also an important object of reuse. Reuse of design and development processes has naturally been carried out in the form of the experience and knowledge of individuals and teams. However, the idea of clearly describing and reusing them in a planned manner is relatively new, and is in a state of underdevelopment compared to program reuse.

### 11.3.1 Reuse

**Modularization**

When reusing a program, if the target is the entire existing system, then it is merely ”using” the system, so any reuse requires some part to be considered as a unit of reuse. This unit of reuse standardized according to certain standards is called a component, and a collection of such components is called a component library.

The following issues exist in modularization for organizing components and the technology to use them.

1. What kind of components should be prepared?
2. How should components be searched?
3. How should the searched components be converted and further assembled as necessary?

Here, let us consider point 1 in particular: what kind of components should be prepared. This issue is related to the classification of components. First, classification by field such as clerical work, scientific and technical calculation, and further subdivision into accounting, inventory, structural calculation, graphics, is possible, but this is generally not a big problem, as it will be decided automatically once the application field is decided. In fact, the old libraries of mathematics software are examples of collections of components created by field.

Second, classification by the size of the program unit that will be the component, or by the semantic hierarchical level, is possible. With the subsystem level as the largest unit, modules (although what is meant by this varies depending on the design method and programming language), procedures, functions, macros, etc. are likely to be considered. Deciding which of these units should be used to compose the components is a fairly big decision.

Classification by the degree of completion of the components, or conversely, the degree of processing required for reuse, is also possible. Here, the degree of completion means that the degree of completion is highest when the part is used as is, and the degree of completion is low if the part is modified. Therefore, it is important to note that a high degree of completion means that it is difficult to adapt to individual reuse requirements and lacks flexibility. Processing performed during reuse can range from setting parameters to writing specified parts to meet individual conditions. Methods for setting parameters can also range from macro-like substitution to a generator that is given parameters and creates a specialized program. In addition, the method of writing specified parts is called a skelton type that is when only a processing pattern is given as a part.

**Reuse in object-oriented programming**

From the perspective of software engineering, object-oriented technology was expected to have a great effect on reuse, especially. In the case of object orientation, there is an object (or class) that is a perfect unit for making parts. By making objects parts, many of the problems mentioned above are solved. For example, the problem of what size and level of parts should be is naturally eliminated. When using components, you can use them as they are, or you can use class inheritance to further specialize or modify them by using data (attributes) and processing (methods) that have already been defined. It is also effective to prepare components by defining only the specifications for reused class components and then specifying them when creating individual classes or instances. When generating an instance from a class (or even generating a class from a metaclass), it is natural to specify parameters to create something that meets the purpose. In other words, the various cases listed above can be easily realized within the same object-oriented framework.

Object orientation also provides a partial, though not complete, solution to the problem of searching and assembling components. The class hierarchy structure is useful for searching, and components are linked by exchanging messages between objects, so as long as you have the necessary components (objects), assembly is possible in a sense naturally, and introducing new objects to add functions can be done by following the message exchange interface. A wide variety of object-oriented libraries have been created.

Not only are there general-purpose component-based libraries, but such libraries are also being created and distributed for specific application fields. However, the following problems have become clear:

1. Class-based reuse is too small, so the effect of reuse is limited.

2. If the inheritance structure created by a class library does not match the structure derived from the application system to be developed, even if there is a possibility of reusing individual classes, they cannot be used effectively.

In the former case, the reuse of frameworks and design patterns as larger units has been promoted. In the latter case, research has been conducted on aspect-oriented software development, which separates groups of objects that work together in a certain way and specific concerns such as security and privacy, and reuses these independently described units.

### 11.3.2 Re-engineering

Reconstruction is the process of rebuilding a system. Inspired by the concept of software re-engineering, BPR (Business Process Re-engineering), which means re-engineering of business processes, has been attracting attention in the field of business administration since the 1990s. When it comes to software, re-engineering of the system itself has been practiced for a long time, rather than the process, and the technology for this has been developed.

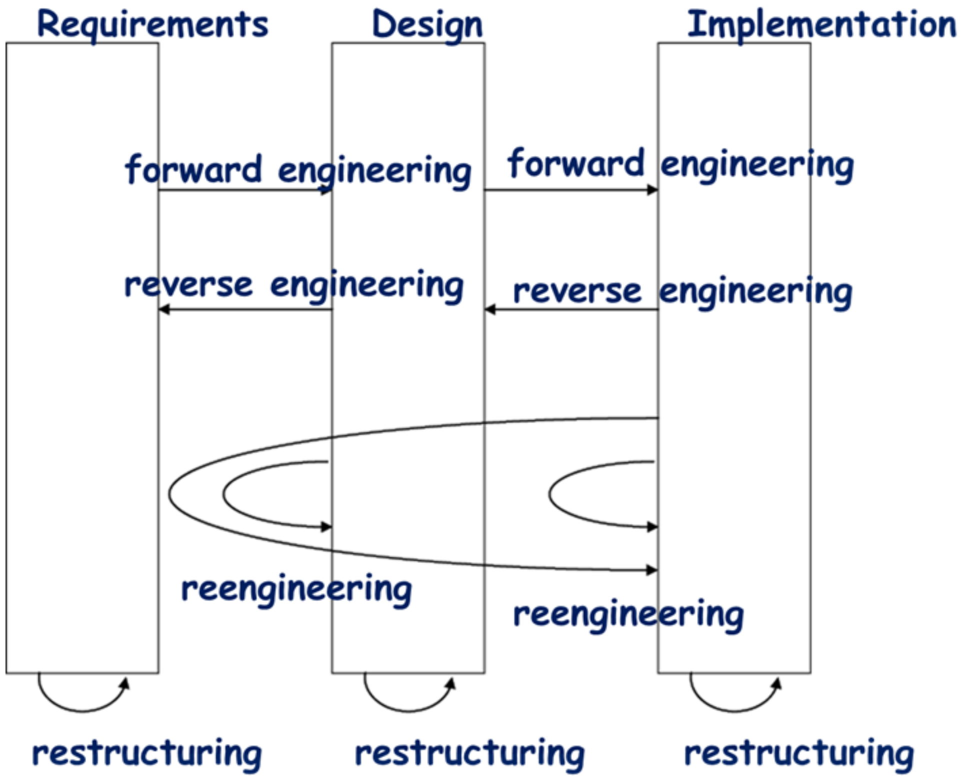


Figure 11.1: The position of re-engineering according to Chikofsky

### The relationship between re-engineering and reverse engineering

Figure 11.1 shows a diagram by Chikofsky that depicts the relationship between reconstruction and de-construction [32].

As this diagram shows, reverse engineering is inextricably linked to re-engineering.

In the 1980s, expectations for CASE tools rose, and one of the claims was that they were effective for maintenance. However, this was because when CASE was used to develop a new system, the repository (a storehouse that stores the products of development, such as requirements specifications, design specifications, data models, programs, and test cases, along with information about their structures and relationships) created was useful for maintenance. However, it could not be directly applied to software that already existed and was in operation, and for which no reliable information existed other than the source programs. This created a demand for tools that could analyze existing software, register it in a repository, analyze its structure and behavior, and support changes and regeneration.

### Reverse engineering

Reverse engineering is a process that reverses the input-output relationship of engineering, i.e., normal manufacturing and development. In software, a typical example of reverse engineering is recovering design specifications from a program. Reverse engineering can also be used to obtain source program code from machine code, obtain a general design from a detailed design, or obtain requirements specifications from a design.

A problem that often occurs during maintenance work is the absence of reliable documentation other than the source program. Even if documents such as requirements specifications and design specifications created during development remain, it is not certain whether they correctly reflect the current state of the system and correspond to the source program.

The original solution to this problem is to always keep the specification documents up to date during development and maintenance, but in reality it would be useful if specifications could be reverse-generated from a program by reverse engineering.

A compiler generates a machine language program from source code, but a decompiler works in the opposite direction, generating source code in a high-level language from machine code, and there have been various attempts to do this since old days. However, in the case of Java source programs that are converted into Java bytecode, a code for a virtual machine, the distance between the compiled code and the source code is close, so decompiling is not that difficult.

Even so, there are many technical difficulties in generating specifications from a program. Essentially, it is impossible to recover a complete design without adding design information that is not written in the program. However, there are various ways to provide information to understand the structure and meaning of a program, even if the specification cannot be generated perfectly. Such methods are generally called program understanding, and research and some practical applications are being advanced.

In the figure, ”restructuring” means, at the implementation level, restructuring the code of an existing program without changing its functions. In agile models such as XP, refactoring is often performed. This corresponds to the restructuring in this figure. From another perspective, the agile process can be said to be a model in which forward engineering and re-engineering are repeated from the initial development stage.

# Chapter 12

# Development Environments and Tools

Looking back at the software development process, we create requirement models and design models using a modeling tool, write specifications using an editor, write programs using an editor, compile using a compiler, test using a tester, and debug using a debugger. Although various tools are used in this way, historically they were created separately for different purposes. However, from the perspective of the software engineers who use them, it is desirable to have an environment where various tools are integrated, can be handled with a similar interface, and the results are stored as a consistent repository. Integrated development environments were born against the backdrop of such needs.

## 12.1 Development Tools

The culture of easily creating and freely using tools for program development was created by Unix. Unix is based on the idea that files with ASCII characters arranged line by line are the basis for all data, and programs are created in small units with single functions, and these are combined using pipes and filters. The OS itself is written entirely in a high-level language C, and the source code can be freely viewed, copied, and modified by users.

A variety of tools have been created against this backdrop. A symbol of this is the book *Software Tools* by Kernighan & Plauger, published in 1976[78]. However, the programs in this book are written in Ratfor, a programming language that introduces structured language constructs into Fortran. In 1981, the same authors published a book called *Software Tools* in Pascal"[79].

## 12.2 Integrated Development Environment

The Unix environment, which is equipped with a variety of tools, could have been called an "Integrated Development Environment (IDE)" for C but this term was used much later than the publication of books such as Unix and Software Tools. In the 1990s, Microsoft released Visual Basic, whose distinctive feature is that it allows you to edit, compile, debug, execute, and manage files in an integrated manner, and this was called an "integrated development environment." After that, Eclipse, Netbeans, IntelliJ IDEA, and other integrated development environments were released and became widely used. Microsoft, which released Visual Basic, developed and sold Visual Studio as an IDE.

### 12.2.1 Tools included in IDE

These IDEs include the following tools.

**Editor** The editor is at the heart of any IDE. In Unix, the character-based editor vi was the standard, but later, the screen-based Emacs created by Richard Stallman was also widely accepted. Vim, the successor to Vi, is also widely used today.

Editors in IDEs work in conjunction with compilers and debuggers, and have functions such as highlighting the relevant parts of the editor when an error occurs and providing hints on how to fix it. In addition, many IDEs have a completion function as a function for editing. It estimates the entire string from the beginning of the input string and displays it as a candidate, and checks and completes elements that form a section in pairs, such as parentheses, begin and end, and XML tags. Furthermore, by using knowledge of the language, IDEs are devised to reduce the amount of keyboard input required by displaying parameter lists and making extensive use of mnemonics.

**Compiler/Interpreter** Naturally, IDEs are equipped with either a compiler or an interpreter, or both, depending on the language, and an IDE is unique in that it can execute programs on the spot.

**Debugger** Debuggers are used in conjunction with editors, including a breakpoint function that allows you to set a point at which to stop during execution and check the values of program variables at that time or check whether assertions are true.

**Code Search** The function to search for classes, functions, variables, etc. is more convenient than the simple string search of an editor.

**Build Tool** The process of creating an executable program from a set of source code is called a build. In Unix terms, it is the make tool.

**Version Control** Version control tools such as Git are often used in conjunction with IDEs as plug-ins. We will discuss version control tools in another section.

### 12.2.2 Applications that IDEs target

Netbeans and Eclipse were initially targeted at Java, but have since expanded to support many more languages. Visual Studio has supported many programming languages since its inception.

For interpretive languages such as Ruby, Python, Lisp, Scheme, ML, and OCaml, interactive development environments are built into the distribution packages of the processing systems, which in a sense play the role of an IDE. However, other dedicated IDEs are also provided by other distributors, and general-purpose IDEs such as Eclipse and Visual Studio have plugins for those languages. For Python, Jupyter Notebook is provided as an open source development of the interactive development environment, which plays the role of an IDE as well as creating well-organized documents like Mathematica's Wolfrum Notebook.

The fact that IDEs have become familiar is probably due in large part to the fact that general programmers have begun to use IDEs for developing web applications and applications for mobile devices. For example, in addition to Eclipse, Visual Studio, and IntelliJ IDEA, which have already been mentioned, there is PaizaCloud, which supports Ruby on Rails, Node.js, Django, MySQL, WordPress, Java (Tomcat), PHP (LAMP), and more.

The following IDEs are used to develop apps for mobile devices such as smartphones.

**Android Studio** An IDE for Android provided by Google. It is based on the open source version of IntelliJ IDEA.

**Xcode** An IDE for macOS or iOS provided by Apple.

RStudio is an IDE for the R language, which has once again attracted attention due to the spread of data science, and is widely used by R users.

### 12.2.3 Document Creation Support Environments

Program development environments and document creation support environments such as LaTeX, HTML, and e-books (EPUB) have many things in common. A document is created in an editor, typeset using a processing system, corrected if errors occur, and version management such as saving the revision history if necessary. There are various tools for this purpose, but here are some representative ones.

**TeXLive** A support environment for creating TeXdocuments centered on the TeXworks editor. It is open source, originally designed for Unix, and compatible with MacOS and Windows.

**Calibre** An open source development environment for creating e-books. It supports a variety of e-book formats.

## 12.3 Version control tools

A version control tool manages system revisions, specifies the elements that make up multiple versions, and manages which versions developers and users can and cannot refer to. For example, when a developer is changing a design specification, whether other developers are allowed to refer to or change the version being changed is determined in advance as a version control policy. In particular, when multiple developers are developing over a network, they must deal with the problem of conflicting changes. If we are pessimistic about the potential confusion that may occur when multiple developers modify a file at the same time, we should not allow any changes to be made to the document being modified. However, if we are optimistic that overlapping changes will not cause major confusion, we can allow people to freely modify the old version. In this case, the new version can be integrated with the old version when it is registered in the repository, or multiple versions created from a single version can be integrated at some point.

Early tools for version control of programs and source code include SCCS (Source Code Controle System) developed by Marc Rochkind in 1972 [113] and RCS (Revision Control System) developed by Walter Tichy in 1985 [134]. RCS was designed to accommodate the situation in which multiple developers work together to develop a single program. Because RCS was developed under a pessimistic policy, files are not updated by multiple developers at the same time. Later, CVS (Concurrent Versions System) was developed as a version control tool for use when multiple developers work together on a network. This tool was developed under an optimistic policy, and when a new version is registered in a shared repository, it is integrated with the old version. In addition, Subversion, an improved version of CVS, has been used, but the most commonly used tool today is probably Git.

Git is a distributed version control tool developed in 2005 by Linus Benedict Torvalds, the creator of Linux. It is ”distributed” because it uses P2P (peer to peer) consistency, which is what makes it different from CVS and Subversion. Github is a web application that uses Git.

# Chapter 13

# Software for a Safe and Secure Society

The infrastructure of modern society, including electricity, gas, water, roads, railways, and communications, is all operated and managed by software. Furthermore, the information network on which the software runs is itself an extremely important piece of infrastructure. The smooth operation of that software is a fundamental prerequisite for creating a safe and secure society.

## 13.1 Threats to a Safe and Secure Society

In order to ensure the safety and security of software in society, several important qualities are required of the software. From the software product quality model shown in section 3.5.2, the following two types of quality are selected that are directly related to safety and security.

1. Reliability
   (a) Maturity
   (b) Availability
   (c) Fault tolerance
   (d) Recoverability
2. Security
   (a) Confidentiality
   (b) Integrity
   (c) Non-repudiation
   (d) Accountability
   (e) Authenticity

”Reliability” refers to the property of a system or component correctly performing the functions defined in the specifications under the defined conditions. On the other hand, ”security” which is the property of a system or product appropriately protecting information and data from people and other products and systems according to the access conditions allowed for them.

In the field of systems engineering, the term ”dependability” is used as covering almost the same range of meanings in accordance with IEC60050-191 defined by the International Electrotechnical Commission (IEC) (JIS uses the term ”dependability” in ”Z 8115: 2000”). This includes availability, reliability, recoverability, maintainability, and in some cases sustainability, safety, and security.

In any case, if these qualities are not guaranteed, it will be a threat to the safety and security of society. Typical threats are software failures and malfunctions and malicious attacks against software.

## 13.2 Social Impact of Software Malfunctions

Software and information systems make the news only when their malfunctions have caused major negative effects. This shows that, although software is widely used as the foundation of society, its existence is not usually taken into consideration. In this section, we will first give examples of social impacts caused by software malfunctions, summarize the causes, and provide an overview of countermeasures.

### 13.2.1 Cases of Information System Malfunctions

Peter G. Neumann, known for his research on operating systems such as Multics, has been collecting computer risk cases since August 1985 and publishing them in ACM Software Engineering Notes and Communications of the ACM. The database can be accessed from "Forum On Risks To The Public In Computers And Related Systems"[4]. Many other cases have been reported in various magazines and reports, and the number is increasing year by year.

Below are some representative cases where information system malfunctions have brought risks to society.

#### Mizuho Bank's System Failure

Mizuho Bank's information system experienced major problems in April 2002, when the new system began operation after the system integration following a major bank merger. After the Great East Japan Earthquake on March 11, 2011, a new system failure occurred due to a large number of donations being transferred to the disaster-stricken areas, causing some ATMs to stop working and causing major delays in the processing of transfers and remittances.

The troubles did not end there, and four system failures occurred in a short period of time from the end of February 2021, and in June a "third-party committee" announced measures to prevent recurrence. Despite this, in August of the same year, it was announced that the system failure had temporarily rendered it impossible to transfer or deposit or withdraw funds at branches nationwide.

#### System Troubles at JR East

A little after 8 am on January 17, 2011, five Shinkansen trains were completely halted due to a problem with JR East's train traffic control system COSMOS, which finally resumed around 9:30am. Later, on January 18, it was announced that the problem was not a system failure, but that when a certain number of revisions to the timetable are made, the display disappears from the terminal screen. This is a planned operation, but this was not thoroughly understood at the site, so train operations were stopped because it was judged to be a system failure.

However, a similar incident also occurred on December 29, 2008. On this day, due to a system failure in COSMOS, five Shinkansen trains on the Tohoku, Joetsu, Nagano, Yamagata, and Akita lines were unable to operate for about three hours from the first train to around 9am, with 79 trains in both directions canceled and 37 trains delayed by up to three hours and 25 minutes, affecting about 65,400 people, including those returning home from their hometowns. It was reported that the cause of the problem was excessive data input into COSMOS, which resulted in the system not being able to start up in time for the morning launch. In the end, it is very similar to the Mizuho Bank case, in that it was a combination of system performance issues and human error in using the system. Another similarity is that the newspaper-worthy failure occurred again after a period of time.

#### Misorder at Mizuho Securities and the malfunction of the TSE system

This incident is introduced in the section on user interfaces in chapter 99.3.1. On December 8, 2005, Mizuho Securities placed an order with mixed-up data on the number of shares and price, resulting in losses of 40 billion yen. It was later discovered that the order could not be cancelled due to a malfunction in the system provided by the TSE, and a lawsuit for damages was filed all the way to the Supreme Court, where it took 10 years to settle.

### 13.2.2 Examples of Embedded System Failures

Embedded systems, unlike information systems, directly control machines and transportation, and therefore when a failure occurs, it can cause physical damage to equipment and harm to people, and in the worst case, even endanger human life.

The following case is a little old, but it is recorded as an example of an embedded system failure that caused great losses before cyber attacks became a major problem.

#### Failure of Ariane 5

The European satellite launch vehicle Ariane 5 experienced a failure on its first flight on June 4, 1996, exploding 37 seconds after launch. The cause of the failure was software, an error in the process of converting 64-bit floating-point numbers to 16-bit integers.

In 1962, much earlier than Ariane 5, Mariner 1, launched by NASA to explore Venus, also lost orbital control due to a software failure and was destroyed by a self-destruct mechanism five minutes after launch. The commonly held belief is that this error was caused by a mistake of typing a comma in the Fortran repeat statement

```
DO 17 I = 1, 10
```

for a period, and that Fortran's whitespace-ignoring nature led to the statement being interpreted as an assignment statement

```
DO17I = 1.10
```

but it appears that this occurred in a different NASA program.

#### Therac-25 Incident

Between 1985 and 1987, a software error in a Therac-25 radiation therapy machine caused six patients to receive excessive amounts of radiation in Canada and the United States. At least three patients died as a direct result of the excessive radiation.

The machine was developed and manufactured by Atomic Energy of Canada (AECL) and the French company CGR-MeV. The irradiation control is done by a minicomputer called PDP11 from DEC, so it is not ”embedded” in the strict sense. However, a unique OS was developed for the Therac, and the software that controls the irradiation runs under that, so in reality it is equivalent to being embedded. The problem with the software was dealing with the race conditions that often occur in parallel processing. Moreover, there was not just one software defect, but at least two of the six incidents had different causes but similar phenomena.

However, the tragedy of overexposure from radiation therapy machines did not end with the Therac incident. Between 1999 and 2000, 28 patients receiving cancer treatment in Panama were overexposed to radiation, and five of them died as a result of the overexposure. According to an investigation by the International Atomic Energy Agency (IAEA), the cause was indeed an error hidden in the control software.

#### Patriot Missile Interception Failure Incident

In February 1991 during the Gulf War, a US military barracks was attacked in Dhahran, Saudi Arabia, when the US Patriot missile bayonet failed to track and stop an attacking Iraqi Scud missile. As a result, 28 US soldiers were killed. The cause was that the software on the Patriot radar system handled time as an integer, but converted it to a 24-bit real number, which lacked precision and made the tracking data inaccurate. This error accumulated over a long period of use[52].

### 13.2.3 The Combination of Machine and System Failures and Human Error

In this way, failures in information systems can threaten the lives of many citizens and cause huge economic losses, and in the worst case scenario, failures in embedded systems can take human lives. On the other hand, the defect problem in Toyota cars that caused a stir in the United States in November 2009 was suspected to be caused by a malfunction in the electronic control system, but in February 2011, the U.S. Department of Transportation announced the results of a 10-month investigation and found that there was no problem with the electronic control.

Automation by computers has contributed to preventing human mistakes and inconsistent decisions by replacing machine operations and decisions made by humans with software. On the other hand, as the structure and operation of systems have become more complex, the operations and decisions left to humans have become abstracted, and the interface between humans and machines has become difficult to understand. As a result, people are more likely to make errors in their actions, especially when abnormal situations occur, and the impact of these errors is greater.

At the same time, even if the number of parts that are directly operated by humans during system operation is reduced and the possibility of errors occurring there is reduced, the work is being shifted to software developers, and the impact of human errors that occur during software design and development is increasing. This is why software reliability and safety are required.

## 13.3 Cybercrime and Cyberattacks

The software defects seen in the previous section are caused by mistakes made by system designers and operators. However, even if a system is properly designed and operates normally, it is common in modern times for malicious third parties to invade the system, steal or tamper with information, or set up to cause malfunctions. When developing and operating software, careful consideration must be given to defenses against such criminal acts.

### 13.3.1 Examples of Cyberattacks

**Attacks on infrastructure**

Systems that control and manage infrastructure such as electricity are also connected to networks, so they are at risk of being attacked by cyberattacks using the Internet as an entry point.

In 2010, centrifuges at an Iranian uranium enrichment plant were destroyed by a cyberattack using the Stuxnet malware. Malware is software or code designed to commit fraud or cause harm.

In July 2015, a recall of 1.4 million cars was announced in the United States due to the discovery of a vulnerability that could be remotely controlled. Before any damage occurred, experts noticed the risk and conducted experiments, which led the car manufacturer to take measures to recall the cars. In December 2015, a cyber attack using the malware Black Energy 3 on a Ukrainian power company caused a large-scale blackout. It took up to six hours to restore power, affecting 225,000 customers[64]. The following year, in 2016, similar cyber attacks on Ukrainian power companies occurred one after another.

**Attacks on IoT**

In October 2016, a DDoS (Distributed Denial of Service) attack was carried out using a large number of hijacked devices such as surveillance cameras to generate a huge amount of communication, reaching 1.2 Tbps, paralyzing the target of the attack. The target of the attack was a DNS service, and many websites that used it were unable to connect. The malware used at the time was named Mirai.

Mirai is a malware that turns computers running Linux into remotely controlled bots that can be used to attack networks. Its main targets are home IoT devices such as network cameras and home routers.

The following March 2017, Hikvision network cameras were hijacked in the United States, the United Kingdom, New Zealand, and other countries, and were also used in a large-scale DDoS attack. The device passwords were left at their default settings, which was exploited. The malware used at this time was different from Mirai.

**Unauthorized access and leaking of personal information**

In April 2011, Sony's PlayStation Network was invaded from outside, and the personal information of 77 million people around the world was leaked. As a result, Sony was forced to take measures to suspend the PlayStation Network for more than a month. A hacker group called Anonymous was suspected of being the culprit, and several members of the group were arrested in Spain, but the group denied the crime, and the facts are still unknown.

Those who use IT technology to break into computers are sometimes called ”crackers” or ”black hat hackers” to distinguish them from ”hackers,” which means genius programmers. In this case, hackers with good intentions are called ”white hat hackers” to distinguish them. In 2011, Sony was not the only company to suffer some kind of damage from black hat hacker attacks; the main targets included the FBI, CIA, US Senate, Fox.com, Citigroup, and Sega. The fact that the FBI, CIA, and US Senate are listed here suggests that these crimes are not just the work of pranksters, but are also cyber terrorism or cyber warfare between nations, and that organized crime groups are secretly operating behind the scenes.

In Japan, the Unauthorized Computer Access Prohibition Act (officially known as the ”Act on the Prohibition of Unauthorized Computer Access”) was promulgated in 1999 and came into effect the following year. This law prohibits unauthorized access, sets penalties for unauthorized access, and provides for assistance measures by prefectural public safety commissions for access administrators who have been subjected to unauthorized access in order to prevent recurrence.

The Personal Information Protection Law, which was promulgated in 2003 and went into effect in 2005, is particularly relevant in terms of protecting personal privacy.

### 13.3.2 Methods of Attack Exploiting System Vulnerabilities

There are various methods that black hat hackers use to exploit system vulnerabilities. The most common methods are listed below.

#### SQL injection

SQL is a widely used query language for relational databases. In particular, in application fields such as web services, databases are used for user management, product management, etc., and the basic processing is to reference and update data according to user input, and in this case, user input from the web is translated into SQL commands and queries are executed on the database.

For example, consider the following SQL statement.

```
SELECT * FROM table name WHERE column name = '(input value)';
\end{vervatim}

This is a command to extract all records from the specified table where the value of the colu
\begin{verbatim}
SELECT * FROM table_name WHERE column_name = 'any' OR 'x'=x';
```

but since the condition is always true, all records in the table will be selected. In a similar way, by making full use of quotation marks in the input value and injecting various commands, it is possible to extract information that would not normally be leaked.

The information leak from Sony PlayStation was caused by SQL injection. The countermeasure is simple: if quotation marks are used in the string entered by the user, escape characters are added to it to prevent the SQL command from being delimited and interpreted in the middle.

#### Buffer overflow

When a program receives input from the user and writes it to its own buffer area, if data arrives that exceeds the size of the memory area reserved for the buffer, it overflows and rewrites the adjacent memory area. This can be used to cause a buffer overflow, destroying the system or misdirecting its operation. C and C++ programs in particular are prone to this attack because they do not have a memory protection mechanism built into the language.

A typical example of an attack that uses a buffer overflow is one in which an overflow in the stack area is used to change the control of the program and direct the execution control of the computer to the part of the program where malicious code is embedded.

Figure 13.1 shows an image of an attack that exploits a buffer overflow in the stack area. Suppose the data structure like this is created on the stack in response to a procedure (method) call. For example, a pointer to a string is passed as an argument to the procedure, and when that string is copied to the local variable buffer, if it overflows the buffer area, the return address will be corrupted. If the input string is appropriately tampered with, control will be passed to another code set up by the intruder when the procedure returns.

The simplest countermeasure is to use a programming language with a memory protection mechanism, such as Java.

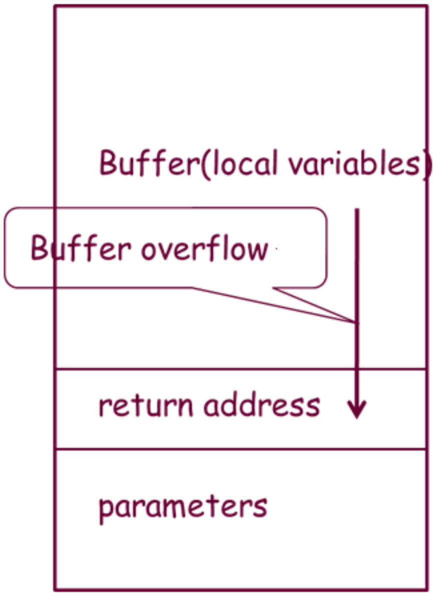


Figure 13.1: Buffer overflow

### Cross-site scripting

Similar to SQL injection, this is the act of entering script code such as Javascript into a user's input field and executing it to make the attacker perform arbitrary actions.

This can be prevented by escaping the input string when outputting it, for example by changing < to `&lt;`.

### Denial of Service attack

This is an attack that mechanically and intensively accesses services provided on the Internet, causing servers and communication lines to become overloaded and malfunction.

Another type of DDoS attack is a distributed attack that uses many computers as springboards.

### Phishing

This is a fraudulent act that leads users to fake websites of well-known companies, etc., and steals personal information such as passwords or money.

### Ransomware

Ransomware is a type of malware that invades a computer, holds data, and threatens to make the data public or prevent the owner of the computer from accessing it unless the ransom is paid. Sodinokibi, LockBit, Darkside, and other highly malicious types are well known.

There are many other methods of attack, and an arms race is underway between the methods of attack and the countermeasures. In any case, it is essential to give sufficient consideration to this during the software design stage, just as to ensure safety.

### 13.3.3 Black Hat Hackers

The term hacker originally meant ”someone who is knowledgeable about computers and finds joy in creating creative programs,” and was created at the Massachusetts Institute of Technology (MIT) in the United States in the late 1950s. On the other hand, especially in newspapers, the word hacker is often used to mean ”someone who illegally breaks into a large system and destroys it from within.” To distinguish between the two, the former are sometimes called white hat hackers and the latter black hat hackers.

Here are some of the most famous black hat hackers in history. All of these are from the United States, but of course there are many other examples outside the United States, even if they are not as famous.

1. **Kevin Mitnick (1963-)** Kevin Mitnick made headlines after breaking into DEC, IBM, Motorola, NEC, Sun, and other companies, but was arrested in 1995. Tsutomu Shimomura, a researcher at UCSD, made a major contribution to his pursuit. Shimomura is a typical example of a white hat hacker, and is the eldest son of Osamu Shimomura, who won the Nobel Prize in 2008. The pursuit was made into a movie called Takedown in 1999. Mitnick served five years in prison before returning to society.

2. **Robert Tappan Morris (1965-)**
In 1988, Robert Morris released a self-replicating piece of software onto the Internet, called a worm, which spread rapidly to many university computers, causing them to malfunction, and became a major incident. This was the beginning of the countless computer worms and viruses that followed. This incident also became a big topic of conversation because his father, Robert Morris, was a well-known computer security scholar. He (the son) is currently a professor at MIT.

3. **Jonathan James (1983-2008)**
In 1999, at the age of 16, Jonathan James hacked into the systems of organizations related to the US Department of Defense and NASA. He was a precocious genius. He was arrested in 2000, and died in 2008 after his release. It is said that he committed suicide.

4. **Kevin Poulsen (1965-)**
Kevin Poulsen took over radio stations in Los Angeles in the 1980s and was arrested in 1991. He later became a journalist.

### 13.3.4 Cyberwar

Many of the hackers listed above later became consultants. Although they were all talented, it is problematic to view them as heroes from the perspective of the impact that system security has on society. However, if they become white hat hackers and work to prevent cybercrime, it will be a contribution to society.

On the other hand, these examples are all individual actions, and in the sense that they are not organized crimes, it can be said that they do not pose a great threat to society. What is truly frightening to society is organized crime. Many recent cybercrime cases are suspected to be committed at the national level. In that sense, they are more like cyberwar than cybercrime. Therefore, each country has created defense units within its military or police organizations to deal with cyber attacks, and has invested large budgets in them.

### 13.3.5 Risk Countermeasures

There are various countermeasures to the risk of safety and security being violated.

#### Risk management

Risk management is a series of processes that identify risk factors, predict the probability of their occurrence, estimate the damage that will occur if the risk materializes, and prioritize avoiding risks with a large product of the probability of risk occurrence and the damage that will occur, by conducting advance evaluations and monitoring the actual situation, and also taking measures to minimize the damage if the risk becomes a reality and an accident occurs.

One method for identifying risk factors during the design and development of a system is to analyze data on accidents that similar systems have caused in the past. However, software does not break down due to changes over time, so the probability of failure in this sense cannot be estimated from past data.

However, various methods for applying systems engineering risk analysis to software are being developed. On the other hand, while software can be a risk factor, it is also useful to incorporate measures to monitor and avoid risks.

### Fault-tolerant and fault-safe systems

A fault-tolerant system is one that avoids the effects of a fault even if it does occur. One effective way to achieve this is duplication. Duplication involves using two or more processing systems that perform the same processing. Common examples include duplication of computers, duplication of communication equipment and lines, and duplication of databases.

There are several patterns for using redundancy through duplication. One method is to perform the exact same processing on two machines, so that if one fails, the other continues the processing. In this case, comparing the results of the calculations of both machines can also be used as a check. With two machines, the problem of which to use when the results differ remains, but a method of using three or more machines and taking a majority vote is also possible, and it is well known that a corresponding majority vote logic is built into the control system on the Space Shuttle. This is also mentioned in the N-version programming section of the 10.2.8 section.

Instead of performing the same processing, there is also a method of distributing the load to two machines, and if one fails, the other machine is concentrated and the processing continues. Of course, this method can be generalized to three or more machines. There is also a method in which only one main machine is usually in operation, and the other machine is on standby, and if the main machine fails, the standby machine takes over the function. Furthermore, especially in the case of processing units such as software, there is a method of time duplication in which if a process fails, it goes back and tries again in a different way.

What is common to almost all of these methods is that it is meaningless if two or more processing systems fail simultaneously or in the same way, so it is assumed that there is a certain independence between the processing systems. This requirement is especially essential in the case of a method that uses majority voting. Another common problem is that even if a duplication system is set up in preparation for failures, the backup device does not work at a critical time. One reason for this is that tests for such cases tend to be insufficient, but it is also common for people involved in such situations to make mistakes or misread the situation due to inexperience.

When the target is software, if the same program is used, errors will exist in the same place, so independence cannot be recognized. Therefore, a measure may be taken to have two teams independently create programs that achieve the same function. However, some research has shown that humans tend to make similar mistakes, and independence cannot be guaranteed even if programs are created by different teams.

A fail-safe system is a system designed to always operate safely when a failure occurs in the system. A similar concept is failsoft, which minimizes damage even when a failure occurs and allows the system to land softly without immediately or completely stopping, and is a concept that is somewhere between fail-safe and fault-tolerant.

# Chapter 14

# Project Management

The themes covered in the previous chapters were theories, methods, and tools for each phase of the software process, such as analysis, design, verification, and maintenance. The theme of project management in this chapter transcends these phases.

## 14.1 Project Management Failures

In Section 3.3.1, we introduced the Standish report published in 1994, which stated that 84% of the projects surveyed were over budget, overdue, or canceled midway through the project. There have been many other reports of project failures. For example, Meilir Page-Jones visited dozens of private software companies and interviewed dozens of managers, and pointed out how many nightmarish projects there were, facing significant delays in delivery and severe complaints from users about quality [**?**]. Edward Yourdon called such projects ”death marches,” and the fact that his book *Death March* was widely read speaks to how many projects have failed.

When a project fails, it means that project management has failed. Why does project management tend to fail? Are there any measures that can prevent failure?

## 14.2 Objectives and Methods of Project Management

A project is an undertaking in which a series of tasks is planned, carried out, and evaluated within a certain period of time in order to achieve a set goal. In most cases, it is carried out by a team of multiple people. A project has a start and an end, and the development team is not permanent and usually disbands after the project is completed.

Throughout history, various civilizations have carried out huge projects, such as the construction of the pyramids in ancient Egypt and the Great Wall of China. A representative modern project from the 20th century onwards would be the Apollo program in the United States. Thus, when we think of projects, we think of construction, space development, plant development, etc., but software development and maintenance projects also have much in common with general projects, especially when it comes to management issues and management methods.

### 14.2.1 Project management process

It is widely recognized and practiced that the implementation of a project is a cycle in which a plan is made, the project is executed based on that plan, the status is monitored during execution, and when a certain unit of work is completed, the results are evaluated and necessary improvements are made. This cycle is generally called the PDCA cycle, which stands for Plan, Do, Check, and Action. The proponent of the PDCA cycle is often attributed to W. Edwards Deming, known as a pioneer in quality control, but it seems that the term was popularized in the field of quality control in Japan, especially in TQC activities, based on a lecture that Deming gave at the invitation of the Japan Science and Technology Agency in 1950.

### 14.2.2 Objects of Management

The objects of project management are as follows:

1. Quality
2. Cost (budget)
3. Time (delivery)
4. People (personnel)
5. Resources
6. Risk
7. Constraints from management, organization, market, system, society, etc.

QCD, which is an acronym of the first three elements, is a catchphrase often used in manufacturing production management. In software projects, people play a particularly large role. Quality management is discussed in Chapter 10, and risk management is discussed in Section 13.3.5.

## 14.3 Project planning and estimates

### 14.3.1 Systemization planning as a decision-making process

Before starting the PDCA cycle for a project, there is a stage where you decide what kind of project should be started and for what purpose. In the process actually used by many companies, this phase is given a name such as ”systemization planning.” In fact, in addition to the results of the requirements analysis, which is what functions are required, the following are important factors to consider when planning software development:

The objects of project management listed in the previous section are also factors to consider when starting a project. In other words, what should be done at the start of a project is to understand the status of these elements and grasp their constraints. After making a comprehensive judgment of these factors, a decision is made as to whether or not to carry out the project. This planning process usually involves not only managers but also various users related to the system developers. In Japan in particular, this phase is often seen as a consensus-building process among such stakeholders [157, 127].

The results are compiled into a document called, for example, a ”system plan” that describes the following items:

- Background
- Development objectives
- Target area
- Current status of business
- Current status of parts that have already been systemized
- Outline of the new system to be developed
- Effects
- Constraints
- Development and implementation plan
- Estimated development costs

- Required resources

Many companies use manuals that summarize the methods and guidelines for such work. Tools may be used in conjunction with the manuals, but their role is auxiliary. The work is often carried out in a training camp format that brings together the relevant parties, and an instructor usually accompanies them to provide guidance and guidance.

Spending time building consensus is often effective in making the next step proceed smoothly, but spending too much time on planning is a tendency often seen in software development projects, particularly in Japan, and may hinder productivity [11, 129]. Other possible problems include the following.

1. Group work, such as at a training camp, depends heavily on the skill of the instructor. Even if the same techniques are used, the quality of the results varies greatly depending on the instructor. In general, there is a shortage of good instructors, and training them is time-consuming.

2. Group work creates a sense of participation among participants and is useful for building consensus. On the other hand, since it is not possible for all users and stakeholders of the system to participate, it is necessary to communicate the results of the work to non-participants, but it is difficult to convey subtle information, including the decision-making process.

3. It is not possible to carry out the same planning work for all projects, regardless of their size. The various methods proposed are quite time-consuming, so how to handle small-scale systems remains a separate issue.

4. Even if the work results in a brilliant organization of problems and the construction of an ideal system, it is often difficult to link this to subsequent development or realization.

One of the starting points of agile development is to avoid spending so much time and people on planning. Plans are expected to change frequently, so new goals are constantly set, planned, and evaluated and improved at weekly meetings, for example.

Below, we will look at cost estimation and scheduling that are carried out during planning.

### 14.3.2 Cost Estimation

Development costs have long been estimated by all development organizations, but there is no definitive method.

The usual method is

1. Estimate the size of the software to be developed in some way.

2. Calculate the required man-months from the size of the software using some estimation formula.

3. Determine the development period and the number of personnel to be deployed from the man-months.

First of all, estimating the size of software is difficult. There are various issues regarding how to measure the size, but the most common measure is the number of lines of the source program. However, since the software has not yet been created, it must be predicted based on some kind of data.

Counting the number of function points is a method that is used to some extent. There are many ways to count function points, but the most common method is to calculate them by weighting the number of kinds of input data, the number of kinds of output data, the number of kinds of processing that the system responds to user requests, the number of kinds of files accessed, and the number of interfaces to other systems that need to be considered. Assuming that this function point number and the number of lines of a program are proportional, the proportionality constant is estimated from past projects, and the number of lines of source program is estimated. An improved version of the function point method for object-oriented systems has also been proposed [8].

Next, we look at the formula for calculating the required effort from the size of the software (the number of lines of source program). However, this relationship is not linear, and it is empirically known that the required effort $E$ and the program size $S$ have an exponential relationship, as follows:

$$E = aS^k \tag{14.3.1}$$

The exponent $k$ is greater than 1, with some data showing it to be 1.5, while a survey by B. Boehm has given values of 1.05-1.2. In any case, the coefficients $a$ and $k$ are estimated from past projects. Boehm first proposed this type of model under the name COCOMO in 1981[22].

This formula is further multiplied by a cost driving factor that takes into account the following factors:

1. Product attributes
   - Required reliability
   - Product complexity
   - etc.
2. Hardware attributes
   - Real-time performance constraints
   - Storage capacity constraints
   - Response time requirements
   - etc.
3. Personnel attributes
   - Analyst capabilities
   - Software engineering capabilities
   - Domain experience
   - Programming language experience
4. Project attributes
   - Tool use
   - Application of software engineering methods
   - Schedule tightness

Set five levels for these elements and adjust the correction factor values accordingly.

Boehm et al. further improved the model using more data and published it in 2000 under the name COCOMO II [21]. There are reports of its use in a considerable number of projects.

The units of labor required estimated here are man-months or man-days, calculated by multiplying the number of personnel by time, but this is also problematic. This labor is based on the implicit assumption that the development period will be shortened in proportion to the number of people doing the same work, but it has long been pointed out that this does not match the reality. Indeed, if a task that takes 10 people 10 months can be completed in 5 months with 20 people, the feeling in the field is that this is impossible. If we consider the relationship between man-hours and development time, where E is the amount of required laborl, P is the number of personnel, and T is the development time, then we should have the relationship shown in Figure 1. However, Frederick Brooks, Jr. argues that increasing the number of personnel actually extends the development time due to communication overhead and training (see Figure 1). He describes it as ”adding manpower to a delayed software project will make it even later,” and calls it ”Brooks' Law.”

### 14.3.3 Scheduling

If a probable number of man-months is estimated by cost estimation, even if there are problems with the man-months scale, it is possible to obtain a rough estimate of how long time will be required to complete the project (in months or days). In addition, for many projects, time constraints are set in advance for management or policy reasons, such as when the project must be completed and the system must be operational. Therefore, the project's work is broken down, the steps between the broken down work units are clarified, and the time required for each work unit is estimated, and the schedule for the entire project is determined. When work is carried out based on this, progress is monitored and progress management is carried out to measure deviations from the plan.

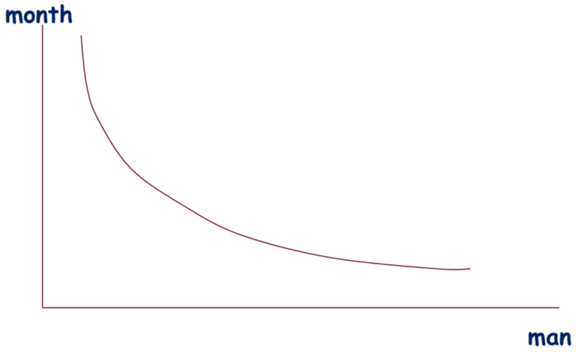


Figure 14.1: Man-months relationship –Brooks

Table 14.1: Example of PERT

| Task | Preceding Task | Working Time (Days) |
|---|---|---|
| A | - | 5 |
| B | - | 8 |
| C | B | 7 |
| D | A | 6 |
| E | A | 9 |
| F | D | 5 |
| G | C, E, F | 5 |

**PERT/CPM**

PERT (Program Evaluation and Review Technique) was developed in the 1950s and is still used today as a method for modeling the relationships between such tasks as a graph, clarifying the schedule structure of the project, and analyzing how to shorten the overall time required and where the bottlenecks in the schedule are. PERT was developed in 1958 by the American consulting company Booz Allen Hamilton Inc. as a method of OR (Operations Research), and its success led to OR itself gaining widespread attention.

In PERT, the subject of scheduling is ”tasks.” Tasks are assumed to have the following attributes:

1. Required work time
2. Possible start time
3. Desired end time
4. Preceding work

The unit of time depends on the problem, but in software development, days or weeks are usually used. The key point is ”preceding tasks.” Generally, there can be multiple preceding tasks for each task, which means that ”a task cannot start until all preceding tasks are finished.” The sequence of tasks is determined by taking this relationship into consideration.

Here is a schematic example:

Figure 14.2 illustrates the relationships between tasks in table 14.1. This is a graph called a PERT chart. The edges of the graph represent tasks, and the vertices are the connections that represent the start and end of tasks. In PERT, these vertices are called ”events.”

The event numbered 1 in this diagram is the start point of the whole project. Among all the paths from this start point to each vertex, the longest path is calculated in terms of the work time. The length of the path represents the earliest

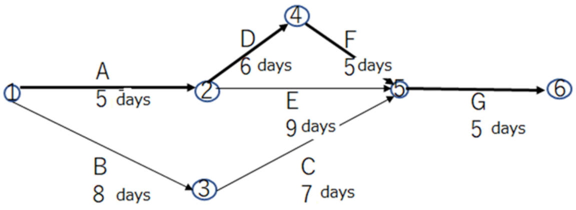


Figure 14.2: PERT diagram

possible start time of the event corresponding to that vertex. The algorithm for calculating the longest path can be the same as the algorithm for calculating the shortest path, such as Dijkstra's algorithm.

The longest path from the start event of the whole project to the last end event (numbered 6 in Figure 14.2) is called the critical path. In Figure 14.2, it is the path shown by the thick line. The critical path means that the tasks on that path determine the time required for the whole project. In other words, the time required for the whole project can be reduced by shortening the time of any task on the critical path or by partially parallelizing the tasks if possible. In 1958, the year PERT was developed and released, a method for finding the critical path was independently proposed and called CPM (Critical Path Method), and today the term is often combined with PERT and referred to as PERT/CPM.

Note that the constraints on preceding tasks in PERT often differ in real life. For example, the condition is that "a task cannot start until all preceding tasks are completed," but in rare cases, an OR condition should be applied, where it is sufficient for any task to be completed. Also, there are cases where some preceding tasks can be started in parallel even if they are not completely completed. In such cases, adjustments must be made to the schedule.

### Gantt Diagram

The Gantt diagram is a scheduling tool that was proposed before PERT and is still in use today. The Gantt diagram is a way of drawing a line chart that was invented by Henry Laurence Gantt of the United States in the 1910s. The horizontal axis represents time, and the vertical axis lists the hierarchical breakdown of the project's work units. The planned time from start to finish of each task is shown as a line segment like a bar graph. Plans and actual progress can also be listed side by side, making it useful not only for planning but also for progress management.

Figure 14.3 shows an example of a simple Gantt diagram. In this example, the tasks are not broken down into a hierarchical structure, but if they were broken down, child tasks would simply be listed under the parent task.

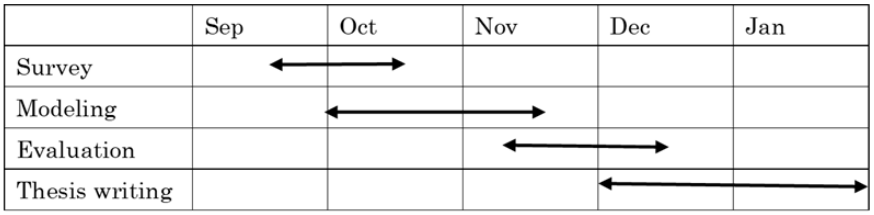


Figure 14.3: Gantt chart

### Resource allocation

Even if tasks are allocated on a time axis in the form of a Gantt chart, resource constraints may not be satisfied. Resources include equipment such as computers, workspace, and communication capacity, but one of the most important considerations in software development is personnel. It is essential to consider the technical level and experience of personnel, but we will treat this as a separate item and consider the issue of quantity, that is, the number of personnel to be allocated.

If tasks can be deployed on a Gantt chart, the maximum number of personnel required for each period can be calculated by adding up the number of personnel required for each task for a certain period on the time axis, such as a day, a week, or a month. If that amount cannot be secured, it is necessary to shift the work period forward or backward to find a feasible solution. A heuristic method called ”piling and collapsing” has been used in the field. Of course, the problem can be formulated mathematically and solved as a mathematical programming problem, and this is actually being done.

## 14.4 People Management

As long as software development is carried out in an organization, the issue of people management is unavoidable. Frederick Brooks' *The Mythical Man-Month* [27, 29], which has already been referenced several times, was written based on Brooks' own experience as a project leader for OS360, which was developed at IBM. The important message is that the most important issue in project management is people. Brooks later wrote a famous essay in 1987 called ”There is no silver bullet”[28], in which he states that there is no ”silver bullet” that has the power to defeat the difficult problem of software development, i.e. the werewolf, but lists some promising things. One of them is ”good designers.” He says that the success of a project depends on finding and nurturing good designers.

Increasing control over people in an organization often stifles the seeds of free creativity. Brooks, who was a manager of large projects at IBM, seems to have had a different view against such strongly controled.

In his discussion of ”good designers,” there are ”exciting products” and ”good products that are not exciting.” Examples of the former are Unix, APL, Pascal, Modula, Smalltalk, and Fortran, and examples of the latter are Cobol, Algol, PL/I, MVS/370, and MS-DOS. The difference is that the former were made by one or a few people, while the latter were developed by committee. In other words, products developed in an organizationally controlled system are boring, but products made by one person or a small team are original. The software examples given may be outdated now, but a similar list could probably be made today.

Agile development was born in a sense as an antithesis to the traditional idea of project management. The waterfall process is convenient for management because it allows you to clearly grasp the current progress against the plan that was made in advance. However, as many projects have experienced, things do not go according to plan, unexpected things often happen, and new ideas arise as the project progresses.

In agile development, the schedule units are short, and they are frequently reviewed before and after. The team size is small, and the level of communication between them is high. This creates flexibility that differs from project management that follows a pre-planned plan.

## 14.5 Project Management Body of Knowledge and Certification

In the United States, the Project Management Institute (PMI) (established in 1969) announced the project management body of knowledge “Project Management Body of Knowledge (PMBOK)” in 1996, and since then, it has been revised continuously, and as of 2021, the seventh edition has been released. Based on this, PMI provides a certification called Project Management Professional (PMP).

In Japan, attempts are underway to establish a Japanese project management standard using PMBOK as a reference, and a body of knowledge called P2M (Project Program Management) has been created. This is managed by the Project Management Association of Japan (PMAJ), a non-profit organization established in 2005, and certifies qualifications based on this body of knowledge. Additionally, the Information-Technology Promotion Agency (IPA) administers a project manager exam based on ITSS (IT Skills Standard, see section 1.1.4) and certifies qualifications for those who pass. There is also an academic society called the Project Management Society of Japan (abbreviated as PM Society), a general incorporated association.

# Bibliography

[1] 2019年 経済構造実態調査報告書（乙調査編）. https://www.meti.go.jp/statistics/tyo/kkj/otsu/2019/pdf/2019report01.pdf, 2019.

[2] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[3] J.-R. Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[4] ACM Committee on Computers and Public Policy. Forum on risks to the public in computers and related systems. http://catless.ncl.ac.uk/Risks.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[6] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[7] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[8] G. Antoniol, C. Lokan, G. Caldiera, and R. Fiutem. A function point-like measure for object-oriented software. *Empirical Software Engineering*, 4(3):263–287, 1999.

[9] V. R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, pages 19–25, January 1990.

[10] K. Beck. *eXtreme Programming eXplained*. Addison-Wesley, 2000.

[11] L. A. Belady. The Japanese and software: Is it a good match? *IEEE Computer*, pages 57–61, June 1986.

[12] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[13] J. Bentley. *Programming Pearls*. Addison-Wesley, 1986. 邦訳：野下浩平訳「プログラム設計の着想」，近代科学社．.

[14] J. Bentley. *More Programming Pearls*. Addison-Wesley, 1988. 邦訳：野下・古郡 訳「プログラマのうちあけ話—続・プログラム設計の着想—」，近代科学社．.

[15] D. Bjørner. Formal software techniques for railway systems. *IFAC Proceedings Volumes*, 33(9):101–108, 2000.

[16] D. Bjørner. *Software Engineering 3: Domains, requirements, and software design*. Springer Science & Business Media, 2006.

[17] D. Bjørner. *Software Engineering 1: Abstraction and modelling*. Springer Science & Business Media, 2007.

[18] D. Bjørner. *Software Engineering 2: Specification of systems and languages*. Springer Science & Business Media, 2007.

[19] S. J. Blackmore. *The Meme Machine*. Oxford University Press, 1999.

[20] J. D. Blaine and J. Cleland-Huang. Software quality requirements: How to balance competing priorities. *IEEE Software*, 25(2):22–24, March-April 2008.

[21] B. Boehm, E. Horowitz, Madachy, D. Reifer, B. Steece, and B. K. Clark. *Software Cost Estimation With Cocomo II*. Prentice-Hall, Englewood Cliffs, N.J., 2000.

[22] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[23] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[24] G. Booch. *Object-Oriented Analysis and Design with Applications 2nd Editon*. Benjamin/Cummings, 1994.

[25] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, 1999.

[26] S. Brock and C. George. *RAISE Method Manual (RAISE/CRI/DOC/3/V1)*. Computer Resources International, 1990.

[27] F. P. J. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[28] F. P. J. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Software*, pages 10–19, April 1987.

[29] F. P. J. Brooks. *The Mythical Man-Month: Essays on Software Engineering Anniversary Edition*. Addison-Wesley, 1995.

[30] J. R. Cameron. An overview of JSD. *IEEE Transactions on Software Engineering*, SE-12(6):222–240, 1986.

[31] P. Checkland and J. Poulter. *Learning for action: a short definitive account of soft systems methodology and its use for practitioners, teachers, and students*. Wiley Chichester, 2006.

[32] E. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[33] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[34] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[35] E. M. J. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[36] CMMI Institute. CMMI Model V2.0 Quick Reference Guide. https://cmmiinstitute.com/cmmi, 2018.

[37] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, 1990.

[38] M. A. Cusumano. *Japan's Software Factories: A Challenge to U.S. Management*. Oxford University Press, 1991.

[39] R. Dawkins. *The Selfish Gene*. Oxford University Press, 1976.

[40] T. DeMarco. *Structured Analysis and System Specification*. Prentice Hall, 1978. 邦訳，高梨智弘，黒田純一郎（監訳），構造化分析とシステム仕様，日経マグロウヒル，1986. .

[41] R. Diaconescu and K. Futatsugi. *CafeOBJ Report — The Langauage, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998.

[42] E. W. Dijkstra and W. H. J. Feijen. *A Method of Programming*. Addison-Wesley, 1988. 邦訳：玉井浩訳「プログラミングの方法」，サイエンス社. .

[43] D. E. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[44] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1625–1634, 2018.

[45] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[46] M. Feldman. TSMC thinks it can uphold Moore's law for decades. *The Next Platform*, Sept. 2019.

[47] J. Fitzgerald and P. G. Larsen. *Modelling Systems*. Cambridge University Press, 1998. 邦訳: 荒木啓二郎他訳，ソフトウェア開発のモデル化技法，岩波書店，2003．.

[48] R. W. Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[49] M. Fowler. Inversion of control. https://martinfowler.com/bliki/InversionOfControl.html, 2005.

[50] D. P. Freedman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Little, Brown and Company, Boston, Mass, 1977.

[51] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[52] GAO: United States General Accounting Office. Patriot missile defense. https://www.gao.gov/products/imtec-92-26, 1992.

[53] H. H. Goldstine and J. Von Neumann. *Planning and coding of problems for an electronic computing instrument*. Institute for Advanced Study Princeton, NJ, 1947.

[54] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: a mechanised logic of computation*, volume 78 of *LNCS*. 1979.

[55] D. Gries. *The Science of Programming*. Springer-Verlag, 1981. 邦訳：筧捷彦訳「プログラミングの科学」培風館.

[56] S. L. Hantler and J. C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys (CSUR)*, 8(3):331–353, 1976.

[57] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[58] I. Hayes. *Specification Case Studies, 2nd ed.* Prentice-Hall, 1992.

[59] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[60] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[61] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, 1997.

[62] IEEE. Recommended practice for software requirements specifications. Technical report, IEEE, 1998. IEEE Std 830-1998.

[63] IEEE Computer Society and Association for Computing Machinery. Software Engineering 2014, Feb. 2015.

[64] IPA. 制御システム関連のサイバーインシデント事例 2. https://www.ipa.go.jp/files/000076756.pdf, Sept. 2019.

[65] ISO. *ISO/IEC 33001:2015, Information technology ー Process assessment ー Concepts and terminology*. ISO, 2015.

[66] R. Jabbari, N. Bin Ali, K. Petersen, and B. Tanveer. What is devops? a systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, pages 1–11, 2016.

[67] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[68] M. Jackson. *System Development*. Prentice Hall International, 1983. 邦訳： 大野・山崎監訳，システム開発法—JSD 法，共立出版，1989．.

[69] M. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudice*. Addison-Wesley, 1995. 邦訳， 酒匂寛, 玉井哲雄訳，ソフトウェア要求と仕様: 実践，原理，偏見の辞典．Kindle 版，Amazon Services International, Inc.

[70] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Reading, 1999.

[71] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM press, 1992. 邦訳，西岡，渡辺，梶原，監訳，オブジェクト指向ソフトウェア工学 OOSE，アジソン ウェスレイ・トッパン，1995．.

[72] I. Jacobson and P.-W. Ng. *Aspect-oriented software development with use cases (Addison-Wesley object technology series)*. Addison-Wesley Professional, 2004.

[73] C. B. Jones. *Systematic Software Development using VDM, 2nd ed.* Prentice Hall, 1990.

[74] M. I. Kamata and T. Tamai. How does requirements quality relate to project success or failure? In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 69–78. IEEE, 2007.

[75] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[76] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-aided reasoning: ACL2 case studies*, volume 4. Springer Science & Business Media, 2013.

[77] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, 1974. 邦訳： 木村泉訳,「プログラム書法」，共立出版，1976.

[78] B. W. Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley Professional, 1976. 邦訳： 木村泉訳,「ソフトウェア作法」，共立出版，1981.

[79] B. W. Kernighan and P. J. Plauger. *Software Tools in Pascal*. Addison-Wesley Professional, 1981.

[80] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[81] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.

[82] D. E. Knuth. *The Art of Computer Programming Volume 3, Sorting and searching, Second Edition*. Addison Wesley Longman, 1998.

[83] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change*. Academic Press, 1985.

[84] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[85] N. Maiden, A. Gizikis, and S. Robertson. Provoking creativity: Imagine what your requirements could be like. *IEEE software*, 21(5):68–75, 2004.

[86] J. Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7-8):721–781, 2012.

[87] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International, 1988. 邦訳，二木 監訳，酒匂訳，オブジェクト指向入門，アスキー，1991．.

[88] B. Meyer. *Object-Oriented Software Construction 2nd Edition*. Prentice Hall International, 2000.

[89] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psycological Review*, 63:81–97, 1956.

[90] H. Mills, M. Dyer, and R. Linger. Cleanroom software engineering. *IEEE Software*, pages 19–25, Sept. 1987.

[91] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[92] M. L. Minsky and S. A. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT press, 1969.

[93] S. Monpratarnchai and T. Tamai. The implementation and execution framework of a role model based language, EpsilonJ. In *Proceedings of the 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (ACIS.SNPD.08)*, pages 269–276, Aug. 2008.

[94] G. J. Myers et al. *Reliable software through composite design*. Petrocelli/Charter, 1975. 邦訳：国友義久, 伊藤武夫訳,「ソフトウェアの複合/構造化設計」, 1979，近代科学社.

[95] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing, Second Edition*. John Wiley & Sons, 2011.

[96] S. Nakajima and T. Tamai. Behavioural analysis of the Enterprise JavaBeansTM component architecture. In *Model Checking Software—Proceedings 8th International SPIN Workshop*, volume 2057 of *LNCS*, pages 163–182, Toronto, Canada, May 2001. Springer.

[97] T. Nakatani and T. Tamai. Empirical observations on object evolution. In *Asia-Pacific Software Engineering Conference (APSEC'99)*, pages 2–9, Takamatsu, Japan, Dec. 1999.

[98] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[99] I. Nonaka and H. Takeuchi. *The knowledge-creating company: How Japanese companies create the dynamics of innovation*. Oxford university press, 1995.

[100] D. Norman. *The design of everyday things*. Basic Books, 1988. 邦訳：野島久雄 訳「誰のためのデザイン？ー認知科学者のデザイン原論」，新曜社，1990.

[101] D. Norman. *Things that make us smart: Defending human attributes in the age of the machine*. Diversion Books, 2014.

[102] OMG. OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.5.1. https://www.omg.org/spec/UML/, May 2017.

[103] A. F. Osborn. *Applied imagination*. Scribner's, 1953.

[104] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[105] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[106] M. Pezze and M. Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[107] M. Phillips. CICS/ESA 3.1 experiences. In J. E. Nicholls, editor, *Z User Workshop: Proceedings of the Fourth Annual Z User Meeting, Oxford 1989*. Springer-Verlag, 1990.

[108] C. Potts, K. Takahashi, and A. I. Anton. Inquiry-based requirements analysis. *IEEE Software*, 11(2):21–32, 1994.

[109] R. Pressman. *Software Engineering : A Practitioner's Approach — 8th edition*. McGraw-Hill, 2014.

[110] S. J. Prowell, R. C. Linger, C. J. Trammell, and J. H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, 1999.

[111] A. Rashid, P. Sawer, A. Moreira, and J. Araujo. Early aspects: a model for aspect-oriented requirements engineering. In *Proceedings of the International Conference on Requirements Engineering (RE 2002)*, pages 9–13, Essen, Germany, Sept. 2002. IEEE.

[112] J. Roche. Adopting DevOps practices in quality assurance. *Communications of the ACM*, 56(11):38–43, 2013.

[113] M. J. Rochkind. The source code control system. *IEEE transactions on Software Engineering*, (4):364–370, 1975.

[114] W. W. Royce. Managing the development of large software systems. In *Proceedings WESCON*, Aug. 1970. reprinted in the *Proceedings 9th International Conference on Software Engineering*, 1987.

[115] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lonrensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991. 邦訳，羽生田監訳，宮追，中谷，桜井 他訳，オブジェクト指向 方法論 OMT，トッパン，1992．.

[116] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[117] S. K. Saha, M. Selvi, G. Büyükcan, and M. Mohymen. A systematic review on creativity techniques for requirements engineering. In *2012 International Conference on Informatics, Electronics & Vision (ICIEV)*, pages 34–39. IEEE, 2012.

[118] D. Sannella and A. Tarlecki. *Foundations of algebraic specification and formal software development*. Springer Science & Business Media, 2012.

[119] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging discipline*. Prentice-Hall, 1996.

[120] S. Shlaer and S. J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, 1988. 邦訳：本位田真一，他訳,「オブジェクト指向システム分析」，啓学出版，1990．.

[121] S. Shlaer and S. J. Mellor. *Object Lifecycles: Modeling the World in States*. Prentice Hall, 1992. 本位田真一，他訳,「続・オブジェクト指向システム分析: オブジェクト・ライフサイクル」，啓学出版，1992．.

[122] I. Sommerville. *Software Engineering (10th edition)*. Addison-Wesley, 2016.

[123] J. M. Spivey. *The Z Notation — A Reference Manual, Second Edition*. Prentice Hall, 1992.

[124] Standish Group International. The Chaos Report (1994). http://www.standishgroup.com/sample_research/chaos_1994_1.php, 1994.

[125] J. Sutherland. *Scrum: the art of doing twice the work in half the time*. Crown Business, 2014.

[126] H. Takeuchi and I. Nonaka. The new new product development game. *Harvard business review*, 64(1):137–146, 1986.

[127] T. Tamai. Current practices in software processes for system planning and requirements analysis. *Information and Software Technology*, 35(6/7):339–344, 1993.

[128] T. Tamai. Social impact of information system failures. *IEEE Computer*, pages 58–65, June 2009.

[129] T. Tamai and A. Itou. Requirements and design change in large-scale software development: Analysis from the viewpoint of process backtracking. In *15th International Conference on Software Engineering (ICSE'93)*, pages 167–176, Baltimore, Maryland, USA, May 1993.

[130] T. Tamai and S. Monpratarnchai. A context-role based modeling framework for engineering adaptive software systems. In *Proceedings 21th Asia-Pacific Software Engineering Conference (APSEC 2014)*, pages 110–117. IEEE, Dec. 2014.

[131] T. Tamai and T. Nakatani. Statistical modeling of software evolution. In N. H. Madhavji, J. F. Ramil, and D. E. Perry, editors, *Software Evolution and Feedback: Theory and Practice*, pages 143–160. John Wiley and Sons, 2006.

[132] T. Tamai and Y. Torimitsu. Software lifetime and its evolution process over generations. In *International Conference on Software Maintenance (ICSM'92)*, pages 63–69, Orlando, Florida, USA, Nov. 1992.

[133] T. Tamai, N. Ubayashi, and R. Ichiyama. An adaptive object model with dynamic role binding. In *27th International Conference on Software Engineering (ICSE'05)*, pages 166–175, St. Louis, Missouri, USA, May 2005.

[134] W. F. Tichy. RCS—a system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[135] K. Torii. Analysis in software engineering. In *Proceedings of the 1994 Asia-Pacific Software Engineering Conference*, pages 2–6, Tokyo, 1994.

[136] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proc. 5th IEEE International Symposium on Requirements Engineering (RE'01)*, pages 249–263, Toronto, Aug. 2001.

[137] A. van Lamsweerde. *Requirements engineering: From system goals to UML models to software*. Chichester, UK: John Wiley & Sons, 2009.

[138] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, 2004.

[139] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, 1998.

[140] H. Washizaki. An overview of the swebok guide.

[141] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. University of Michigan, 1979.

[142] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, 1990.

[143] A. Yonezawa. *ABCL: An object-oriented concurrent system*. MIT press, 1990.

[144] シドニー L ハントラー, ジェイムス C キング, and 玉井哲雄（訳）． プログラムの正当性証明入門 1. *bit*, 11(6):546–555, 1979.

[145] シドニー L ハントラー, ジェイムス C キング, and 玉井哲雄（訳）． プログラムの正当性証明入門 2. *bit*, 11(7):703–713, 1979.

[146] I. ソフトウェア高信頼化センター, editor. *共通フレーム 2013 〜経営者、業務部門とともに取組む「使える」システムの実現〜*. IPA（独立行政法人情報処理推進機構）, 2013.

[147] 中島震，玉井哲雄. EJB コンポーネントアーキテクチャの SPIN による振舞い解析. *コンピュータソフトウェア*, 19(2):2–18, 2002.

[148] 伊東暁人. 大規模システム開発におけるプロジェクト管理問題. In *ソフトウェアシンポジウム'91 論文集，D9-D16*, 1991.

[149] 吉田民人. *情報と自己組織性の理論*. 東京大学出版会, 1990.

[150] 大野とし郎. ジャクソンシステム開発法. **情報処理**, 25(9):955–962, 1984.

[151] 山崎利治. **プログラムの設計**. 計算機科学／ソフトウェア技術講座 3. 共立出版, 1990.

[152] 川喜田二郎. **発想法―創造性開発のために**. 中公新書. 中央公論社, 1967.

[153] 川喜田二郎. **続・発想法―*KJ* 法の展開と応用**. 中公新書. 中央公論社, 1970.

[154] 情報処理学会. Citp 認定情報技術者. https://www.ipsj.or.jp/citp.html.

[155] 有沢誠. **アルゴリズムとその解析**. コロナ社, 1989.

[156] 村上憲稔. ソフトウェアのライフサイクル管理. **情報処理**, 33(8):912–921, 1992.

[157] 玉井哲雄. 要求定義とプロトタイピングの現状. In **情報処理学会「プロトタイピングと要求定義」シンポジウム**, pages 103–110, 4 月 1986. 招待論文.

[158] 玉井哲雄. ソフトウェアの語源. *bit*, 33(1):54–57, 2001.

[159] 玉井哲雄. **ソフトウェア社会のゆくえ**. 岩波書店, 東京, 2012.

[160] 玉井哲雄, 三嶋良武, and 松田茂広. **ソフトウェアのテスト技法**. 共立出版, 東京, 1988.

[161] 石畑清. **アルゴリズムとデータ構造**, volume 3 of **岩波講座 ソフトウェア科学**. 岩波書店, 1989.

[162] I. 社会基盤センター, editor. ***IT* 人材白書 *2020***. IPA（独立行政法人情報処理推進機構）, 2020.

[163] 紙名哲生. 文脈指向プログラミングの要素技術と展望. **コンピュータ ソフトウェア**, 31(1):1_3–1_13, 2014.

[164] 藤本隆宏. **能力構築競争ー日本の自動車産業はなぜ強いのか**. 中公新書. 中央公論新社, 2003.

[165] 谷島宣之. **ソフトを他人に作らせる日本，自分で作る米国－「経営と技術」から見た近代化の諸問題**. 日経 BP 社, 2013.

[166] 野中郁次郎 and 竹内弘嵩. **知識創造企業**. 東洋経済新報社, 1996.

[167] 野木兼六. 最適化に基づくアルゴリズムの発見. **コンピュータソフトウェア**, 11(4):20–43, 1994.

[168] 長尾真, 安西祐一郎, 神岡太郎, 橋本周司. **マルチメディア学の基礎**. 岩波講座 マルチメディア情報学. 岩波書店, 1999.

[169] 鳥光陽介, 玉井哲雄. ソフトウェア・システムの寿命とその要因についての考察. In **ソフトウェアシンポジウム *'92* 論文集**, pages E–2–E–10, 長野, 1992. ソフトウェア技術者協会.