Framework and Business Logic Components

— Practical and Effective Component-Based Reuse Systems —

Figures and Tables

Preface

Conventional, arbitrary software development substantially differs from development that employs the method of **'Business Logic Components'** (see **Note 1**). This difference is comparable to the one between sailing a boat and driving a car.

On the sea, there is neither obstruction nor paved road that can be seen on the ground; instead, you can see how the wind and waves make their patterns on the sea surface. To take the shortest way to the destination, the ultimate technique of yacht racing would include analyzing the changing patterns and taking the fastest zigzag course. However, if not in a race, the course would be unrestrained. Simply by letting the wind decide your course, you could still arrive at the destination.

On the other hand, driving a car on the road shows considerably different characteristics. For example, when going from one place to another, any taxi driver will take approximately the same route. This is not a straight-line connecting two points, nor is it a zigzag course depending on the wind or current; rather, it is a course determined by suitability. On the road, although traffic regulations must be respected, a driver can arrive at the destination by fully utilizing the engine power.

Many conventional software development works are likened to a yacht sailing on the ocean. If the ultimate technique of yacht sailing were pursued, the best course for each development would likely be found. Yet, it is possible to develop software without struggling to seek the ultimate technique. In fact, most software development may have allowed the developers to seemingly pursue the ultimate technique, more so for personal gratification than for the sake of efficient development.

On the contrary, the software development that employs '**Business Logic Component Technology**' (see **Note 2**) would resemble driving a car on the road. In a sense, this is to benefit from social order by imposing traffic regulations and creating systems. This way, the computer power enables any developer to take the same course without any predicament. This would lead to offering economical, convenient traffic regulations, even though it might diminish their pride in pursuing the ultimate technique.

Note 1: To avoid ambiguous interpretations of the term business logic components, this book offers a clear definition. The term is enclosed within single quotation marks when the definition is met. Also, for generalized business logic components, I use the name 'Software Component' within single quotation marks.

Note 2: I identify the technology of practical and efficient 'Business Logic Components' that meets the definition of this book by enclosing it within single quotation marks. Likewise, the technology of 'Software Components' that meets this book's definition is enclosed within single quotation marks.

This book is intended to demonstrate the effectiveness of **'Business Logic Component Technology'** in developing business application programs in the business field. How can I prove its effectiveness? It can be explained by the underlying fact of the **duplicated developments** of similar business application programs. Even if two application programs seem similar, for example, there are generally some differences between a program aimed at Company A and one at Company B. For this reason, they are usually developed independently of each other. Yet, if asked, "What are the differences," they are frequently only slight (at

least they are perceived to be so). Accordingly, application programs in the same category should be able to be developed jointly. With differences of ten percent, for instance, the remaining ninety percent can be made common. Nevertheless, this common sense used to make common parts common is still considered difficult. Unreasonably, this is commonplace in the business application program development industry.

There are causes that have formed this unreasonable situation. This book presents one methodology to technically analyze those causes and facilitate efficient development by changing that unreasonable situation. In other words, it manifests the fact that practical and effective component-based reuse is possible through 'Business Logic Component Technology.' Moreover, it points out a way to make it effective to develop business application programs in the business field.

At this point, let me ask a question before you read further. "What do you think '**Business Logic Components**' are?"

Many people typically associate them with general subroutines. Others would claim that business logic components are the objects that are based on object-orientation, which is being repeatedly talked about in the topic of component-based reuse. In addition to these, there are many different ideas of what business logic components should be. Also, others may still believe that the realization of the idea is a future issue, even with obscure expectations regarding business logic components. Meanwhile, some critical people claim that it is nonsense in the first place to ask the question, "What are business logic components," since there are no such things as software components. Those kinds of people are likely to criticize saying, "Only putting together general subroutines doesn't necessarily make a business application program. You know what I mean through your experiences, right?"

Hence, there is no standard that defines the term "business logic components." Whereas some people have ambiguous expectations of business logic components or software components, others deny their existence.

This book offers clear answers to those who are suspicious of business logic components, while demonstrating what is meant by practical and effective 'Business Logic Components.' The 'Business Logic Components' of this book are results of a few peoples' challenges in an undeveloped field, but enough activities to publicize its usefulness have not yet been held. Thus, this book is written to justly report these newly developed 'Business Logic Components.' However, since it would not be very easy to understand the concept only by explaining 'Business Logic Components' alone, I also describe related technologies in a comparative manner in this book. Therefore, this book not only explains 'Business Logic Components' but also illuminates some of the existing development support tools simultaneously.

I made this book readable for anyone who is interested in streamlining business application program development in the business field. For this reason, I made it easy to understand this book's main points at issue by inserting them as "Topics." At the same time, I intend to discuss those points at issue from various viewpoints.

I presume that this book is mainly intended for the following readers; however, any other types of readers will be more than welcome.

- Those who are concerned with the development of business application programs and those who use them in a corporate environment;
- Those who are concerned with the development of usual business packages and those who use them;
- Those who develop business application programs according to customers' orders and the customers themselves; and

• Those students who are seeking a job such as the ones listed above.

In the recent discussions on ERP (enterprise resource planning) packages, conventional business packages have been reconsidered, and competition has arisen between them and custom application programs. 'Business Logic Component Technology' of this book is intended to streamline the development process of general business application programs, including ERP packages, and its applicable scope covers the whole business field. Meanwhile, this technology facilitates to break the wall between usual business packages and custom application programs. Consequently, I believe that this book will surely be instrumental in streamlining a wide range of development processes.

April 1998
Yasuhito Tsushima

For revision

This book was originally written five years ago. Now, updating terms in order to suit the recent software environment, I changed the title from **"Software Components"** to **"Framework and Business Logic Components."** This is because I felt the need to clarify the issue of "components" since that concept has been twisted due to the recent prevailing propaganda that is described later.

Although Microsoft Corporation has promoted so-called **VBX** (Visual Basic Extension) — later renamed as **OCX** and then as **ActiveX control** — as a component, this is the GUI component used by pasting and designing a form. It is a contributor that built the "component market," but not a business logic component. Similarly, **EJB** (Enterprise Java Beans) developed by Sun Microsystems is not a business logic component, but rather it is nothing more than a module with a good interface with the middleware, called an application server, even though some people call **EJB** a component. This is because **EJB**'s reusability cannot be enhanced without painstaking effort. The reusability of most software products can be improved through continuous efforts. In that connection, **EJB** is not an exception. Only taking the form of **EJB** does not enhance reusability. Though I am happy to see the rise in awareness of **'Business Logic Components'** in response to the emergence of those would-be "components," I am concerned about the supposed capability of **'Business Logic Components.'**

In the previous version of this book, **"Software Components,"** I made clear the necessary requirements for components to be called practical and effective components. Nevertheless, it is not desirable that many "component" believers abuse the term "component" without considering those criteria. It would be understandable if they created new requirements. Yet, saying "components" only as an advertising catchphrase is not acceptable and is misleading.

Yet, it is not impossible that this kind of "component" is considered a "component" in a casual sense. Thus, I re-titled this book **"Framework and Business Logic Components"** and set a hurdle to see whether components alone can completely cover business logic.

Now, witnessing this social trend, I cannot help but thinking that the development technologies of business programs have not been improved. Also, it is seen that although taken up as a topic, "components" are yet to be analyzed in depth. Despite this slightly pessimistic view, I can recognize a steady improvement as well, so that I can now describe some concepts with widely accepted terminology that I did not even know what to call when I authored the previous version.

For example, we can now apply the term "**refactoring**" (used in Extreme Programming) to **major surgery** that separates programs related to operation specifications from those for business specifications in a settled state.

We can also express what I describe as **common main routines** in the section of "**Demarcation of Figure and Ground When Recognizing Something**" of this book, using the term "**inversion of control**."

Furthermore, the **skeleton routine** of this book is a program that covers the **frozen area** and can be a **framework,** in the narrow sense. Likewise, the **slot** of this book means the **hot spot**, and the **supplementary routine units** of this book are not different from the **hook methods**. With the long-term view, we can observe technological progress.

Apart from these widely prevailing terms, there appears to be some newly created terms such as **operation base**. Operation base means the programs for operation that are located at the front-end, on the contrary to databases at the back-end. As for its form, it is generally engineered as a framework, in the narrow sense.

Lastly, note that I designate the term "**need for creative adaptation**" as **NCA** in its abbreviated form. This is not an imitative "adaptation" but a creative one (because I needed to create the new term for what had never existed). By the way, at this time careful consideration was necessary to refine terminology, even though NCA was not greatly required.

November 2003
Yasuhito Tsushima

For third edition

Currently being translated.

May 2008
Yasuhito Tsushima

Author's Notes

In writing this book, I paid close attention to the points that are explained below. By understanding them, this book will become more instrumental.

First, let me talk about terminology. In the field of business application programs, there is an agreed-upon standard called the data-oriented approach, which cautions the use of homonyms and synonyms. This book basically follows that standard, with the exception of the following.

I basically tried to avoid **synonymy**; however, I employed shortened forms for excessively lengthy terms such as “business app” for “business application program,” “business system” for “business application system,” and “custom business program” for “custom business application program.” But, where I need to emphasize terms as “program,” I used the original terms such as “business application program” without shortening them. Along with this shortening process, I changed the term, “custom business application program development firm (industry)” to “custom business program development firm (industry).”

I intentionally did not unify some terms with similar meanings when they involve subtly different implications. “Software,” “program,” and “procedure” are typical examples as well as “gene” and “DNA.”

Different institutions sometimes use different terms for the same concepts despite similar meanings. In cases like this, although trying to show their synonymy, I intentionally did not unify the terms.

I also tried to avoid **homonymy**. To do this, I devised expressions or attached adjectives to potentially indefinite terms to facilitate their distinctions. For instance, I attached adjectives to an ambiguous term “system” so as to separate one term from another unless the meaning is obvious. Some examples are “business system,” “manufacturing management system,” “sales management system,” “component-based reuse system,” “fill-in system,” “component retrieval system,” “component management system,” “prototype system,” and “reuse systems of componentized applications.” In addition, to avoid homonymy, I used capitalization such as “case statement” and “CASE tools” and used italicization such as “*parameter*” and “parameter.”

In the cases where I assign unique meanings to terms (**assignment of a unique meaning to terms**), I enclose them within single quotation marks for distinction. They are ‘Business Logic Components,’ ‘Software Components,’ ‘Business Logic Component Technology,’ ‘Software Components Technology,’ ‘Black-box Component’ and ‘White-box Component.’

I mostly use terms in accordance with generally agreed-upon usages, but when I use unique terms that I created in this book, they are, as explained in the main text, extracted and described as **“Keywords for Understanding This Book.”** The same is true with existing terms that are assigned a particular meaning. Refer to those explanations for terms and the relationships between terms when appropriate.

Aside from the use of terms, italicized parts in this book are intended to explain how to read this book.

Next, let me mention my point of view while writing this book.

This book is aimed to make ‘Business Logic Components’ valued fairly, rather than either overvalued or undervalued. Thus, I wrote the book with much care so that no exaggeration would be committed.

Today is a time when the computer is transforming itself from the realm of the mystical to an entertainment system. When the computer was first introduced, it was almost apotheosized, just as words, letters, and paper as mediums were considered nearly mystical a long time ago. Possibly, in accordance with this bygone age, people still tend to blindly accept things as accurate and optimum simply because a computer was used. Beliefs like this mystical idea may be welcomed by computer businesses since they help popularize computers through some kind of subconscious effects. Therefore, no effort has been made in redressing it, but we should stop regarding the computer as the realm of the mystical. Used as stationery and as an entertainment tool, a computer does not deserve to be apotheosized. Showing what the computer really is, will enormously ease the understanding of it. Any exaggerated expressions that may cause illusions should be refrained from because there are already a massive number of perplexing technologies in the computer world. So, in writing this book, I tried to negate these illusions, or at least, did not try to promote them.

This book not only discusses 'Business Logic Components' themselves, but also compares them to their peripheral technologies. However, this inadvertently resulted in criticizing those technologies partly because we faced the need to redress the over evaluation on development support tools by today's commercialism and commerce-oriented computer journalism. I then remained consistent, asserting along with the little boy, that "The emperor is naked."

Generally, today's trendy high-tech keywords contain, more or less, over-expressions. To those who are accustomed to them, this book's non-sensational expressions may look rather quiet, or it may seem that the book lacks impact because of insufficient high-tech feeling. If one applies to this book a high-tech field's unwritten rule, the strange principle, "it must be gorgeous," the book might not be appreciated right away. If possible, I hope you will evaluate 'Business Logic Components' in this book with a cautious mind against those bright high-tech keywords. Also, take enough time to read this book because it opens a new perspective that has been left behind, while scrutinizing it with critical eyes to see whether I myself overstate 'Business Logic Components.'

I could have discussed 'Business Logic Components' by focusing on concrete examples of programs just as physics is clearly explained with mathematical formulas. However, usually people, even those who have a strong programming background, are likely to be reluctant to read programs written by others. Therefore, I did not involve programming examples in this book so that it could be enjoyed by anyone.

Due to this orientation, some readers may find explanations in this book abstract. If you are interested in confirming what is argued in this book with real programming samples, information is available on those samples at the address below:

http://www.applitech.co.jp/

Programming commonly takes place in each person's mind, and those who experience that process are only a fraction of the whole population; besides, few efforts are made to show its actual circumstances. In addition, it is difficult to capture the actual circumstances solely by observing programmers from the outside. So, when disclosing the substance of programming, I tried to provide readers with a solid understanding of it. Refer to **Appendix 1 "What Does Running a Program Mean?"** if you are not familiar with programming.

However, if you have ever speculated about programming simply with your own imagination, it would be better to read this book after resetting your mind. For example, if you believed that writing programs

resembles manufacturing products, I would like you to forget it. Similarly, I would like you to forget the idea that software development productivity can easily be evaluated with numbers, if you believed so.

Lastly, let me clarify some points about this book's main theme, the component-based reuse system.

Because this book begins with the topic of the usual business package, you might have the idea that 'Business Logic Components' are something related only to customization work in business packages. However, I emphasize here that that is a misunderstanding. Of course, they are effective in maintenance work, either, and more importantly, in development work for new application programs, if related to an already developed field. For instance, they are effective in the new development of business fields such as one where Fourth-Generation Languages (4GLs) exerted their effectiveness since they can reuse the results of that field that have already been developed.

Unfortunately, on the other hand, the component-based reuse system is ineffective for software development in a completely new field that is not related to any already developed field (nor is it to a business field). However, once a new field has been developed, it may be changed to the stage where 'Software Components' and others flourish. If that happened, component-based reuse systems could exercise their effectiveness in order to utilize the results of the new field. Either way, they are not useful until a development process is completed. To clarify what is meant here, note that this condition applies not only to the 'Business Logic Components' of this book, but also to component-based reuse systems in general.

Keywords for Understanding This Book

This section presents this book's unique terms and the terms to which I assigned a particular meaning.

Generally, at the beginning of a program, variables and constants used in that program are acknowledged. According to this rule, in this book let me explain here the significant keywords which correspond to the variables and constants.

These terms are explained in the main text as well, so I recommend that you glance through this section and go on to the main text, and that you refer to this section when you are confused about unknown terms.

Black-box component:

Component that is used (usable) without deciphering an inside program.

In other words, they are components that disallow program customization, and the source programs are not usually disclosed.

In this book, the black-box components that have generality are noted within single quotation marks as 'Black-box Component.' 'Black-box Components' are a type of 'Business Logic Component.'

General subroutines and general main routines are examples of 'Black-box Components.'

The argument for this term appears in **5.2-f "Generalized Construction Technique for a Reuse System of Componentized Applications."**

Bloat:

Redundant portion of a program. Although many programs are constructed for a similar outcome, the most compact of them are regarded as the standard with no bloat. With this kind of standard, it is revealed that other programs have the same amounts of bloat as the difference between the most compact one and others.

A typical example of bloat is a redundant portion of a program (removable parts). Another example of bloat is a portion that can be compacted by replacing it with common subroutines but has not yet been compacted. Also, if you re-develop the program stored in the library without reusing it, then that is also bloat.

The argument for this term appears in **4.1-c "Minimum Information Content of a Program."**

Business logic components:

The component sets used for the software component-based reuse system.

General subroutines are a typical example of business logic components.

In this book, the business logic components with particular qualities are noted within single quotation marks as 'Business Logic Components.' 'Business Logic Components' are categorized either as 'Black-box Components,' which have generality, or as 'White-box Components,' which have all the four qualities of retrievability, locality, suitable size (suitable granularity), and readability.

General subroutines, general main routines, and data item components are other examples of 'Business Logic Components.'

The argument for this term appears in **5.3 "Meaning and Significance of 'Business Logic Components'."**

Business logic component technology:

The technology that utilizes ‘Business Logic Components.’

In this book, the technology that utilizes ‘Business Logic Components’ with the prescribed set of qualities are noted within single quotation marks as ‘Business Logic Component Technology.’

With ‘Business Logic Component Technology,’ customization work can be made extremely easy through “component customization” that can improve the productivity of program customization work.

The argument for this term appears in **5.3 “Meaning and Significance of ‘Business Logic Components’.”**

Business Packages with Special Customization Facilities:

Business packages that can cope with special customizations, such as custom business programs. Because business packages with special customization facilities must accurately match a customer’s special order, program customization needs to be done in some way. If customization costs can be controlled, a substantial cost reduction can be achieved through business packages with special customization facilities, compared to custom business programs.

SSS and **RRR** families are examples of business packages with special customization facilities. For these packages, customization costs are controlled through component customization.

The argument for this term appears in **1.3 “Business Packages with Special Customization Facilities.”**

Common main routines (framework in the narrow sense):

The routines that consist of common portions extracted from a program that function as main routines rather than those portions that function as subsidiary works. In the majority of development projects, there is the convention to extract common subroutines from programs that function as subsidiary works. So, from a slightly different point of view it should be possible to extract common main routines in the same way.

Some examples of common main routines are the models of a primitive program, the skeleton routines of a classic fill-in system, **4GL** operation bases, GUI operation bases, and frameworks in the narrow sense, many of which are mainly used to carry out operational processes in the common process.

In development projects where only common subroutines are the focus, it could be possible to find common portions that have been overlooked by trying to find common main routines that have not been found. If common main routines are discovered, productivity can be improved by reusing them. Employing the object-oriented programming (OOP) language easily facilitates writing common main routines because it can better manage the inversion of control.

The argument for this term appears in **3.2.2-n “4GL and Fill-In Systems.”**

Compensated productivity:

The value that is approximated to “true productivity or the value of productivity after completely removing bloat” by revising plain productivity.

Although plain productivity is easily obtained, if one depends on this value, a compact program loaded with dense substance is likely to be undervalued, while one inflated with redundancy is likely to be overvalued. As a result, it could suffocate the effort to compact a program and endorse the

attempt to inflate it with bloat. Compensated productivity is a measure invented to avoid harmful effects like this.

To revise productivity, each time bloat is found in a program, subtract the amount of the bloat from the total value to re-calculate productivity.

The argument for this term appears in **4.2-d "How to Compensate Productivity that is based on Number of Program Lines."**

Componentized event-driven system:

The system created by improving general event-driven systems so that event procedures are equipped with the qualities required of the 'White Box Component.'

An example of the tool that employs a componentized event-driven system includes **MANDALA**, which is the core of **RRR tools**. Event procedures of general event-driven systems have not shown clear partitions among data items. For the componentized event-driven system, it became obvious that these partitions were important, so **MANDALA** adopted the structure of update propagation to realize this goal.

The argument for this term appears in **3.2.3-s "Second Improvement of Partitioning Guidelines for Compartmentalization of Components."**

Component-based reuse system:

The system by which software assets are reused in the form of a component or of components.

Component-based reuse systems usually consist of component sets and component synthesis tools.

In this book, I discuss only the component-based reuse system in the form of the reuse system of componentized applications.

It can be said that the call mechanism slotted into a subroutine and the general subroutine library are classic examples of the component-based reuse system. If expanding this classic system with 'Business Logic Component Technology,' we can transform a business app into the component-based reuse system that can be solely comprised of 'Business Logic Components.'

The argument for this term appears in **2.2 "Reuse Systems of Componentized Applications and Object Orientation."**

Component customization:

The customization for application programs that apply 'Business Logic Component Technology.' In other words, it is the program customization using 'White-box Components' practically.

Although the component customization is categorized into program customization, in the broad sense, it is capable of extremely reducing its workload, compared to general program customization.

The reason for the workload reduction is that program codes which should be modified are nothing other than a hundred percent 'components' (worthy to be called so). This means, 'components' have the favorable qualities that are effective in reducing workload. For example, 'White-box Components' require the work to modify its internal codes, even though 'White-box Components' have the four favorable qualities of retrievability, locality, suitable size (suitable granularity), and readability - that are effective in reducing workload. By the way, this book insists it is not a "component" if it has these favorable qualities.

An example of the component customization is the business program customization made of data item components.

The argument for this term appears in **1.3-j "Custom Business Program or Business Package? (Part 3: Conclusion)."**

Component set:

The set of components that constitutes a component-based reuse system. Also, a component-based reuse system usually consists of component sets and component synthesis tools.

Examples of a component set include a general subroutine library, and RRR component sets for sales management and financial accounting.

The argument for this term appears in **3.2.3-q "Improvements for RRR Family."**

Component synthesis tools:

The tools that compose a component-based reuse system, which in turn consists of component synthesis tools and a component set.

Component synthesis tools function to elaborate an application program by synthesizing designated business logic components.

Classic fill-in systems and **RRR** tools are examples of component synthesis tools.

The argument for this term appears in **3.2.3-q "Improvements for RRR Family."**

Data item components:

The components into which a business app is divided according to corresponding data items. Since in the business field specification change requests are expressed by data item names, if we let data item component names begin with each data item name, we can retrieve the components effortlessly without a sizable retrieval system.

Possessing all the four qualities of retrievability, locality, suitable size (suitable granularity), and readability, data item components are considered a representative example of a 'White-box Component.' Also, because the 'White-box Component' is a kind of 'Business Logic Component,' data item components are a type of 'Business Logic Component.'

Examples of data item components involve product code components, product unit price components, and important customer code components, and so forth.

The argument for this term appears in **1.3-i "Program Partitioning with Data Item Association."**

Fourth-Generation Language operation base (4GL operation base):

When you think that something may be calling a 4GL event procedure of an event-driven type, that something is the 4GL operation base. Put in another way, the **4GL** operation base is something that calls the event procedures and determines when each should be carried out.

Also, it is conventional that when an operation base is developed, it is engineered in the form of the framework in the narrow sense. Furthermore, if developed proficiently, it can become the framework with the function of reuse by referencing, a trump card to improve productivity.

The argument for this term appears in **3.2.2** "**Fourth-Generation Languages (4GLs).**"

Framework:

In the narrow sense, it is almost the same as **common main routines** or **general main routines**.

In the broad sense, on the other hand, it involves not only **common main routines** and **general main routines** but also the results of module partition (component partition), including **common subroutines** and **general subroutines**.

The argument for this term appears in **3.2.1-j "Second Branch in a Fill-In System."**

General main routines (framework in the narrow sense):

The common main routines altered for versatile purposes. Just as the common subroutine becomes the general subroutine through generalization, the common main routine is changed to the general main routine in the same way.

Also, generalization means to give the ability to meet all envisioned requests in areas that must be covered by selecting components from component sets and specifying parameters. Alternatively put, generalization is not different from equipping a routine with the qualities required of a 'Black-box Component.'

With object-oriented programming (OOP) language, writing common main routines becomes easier because of the improved facility to manage the inversion of control.

The argument for this term appears in **5.2-g "Comparing RRR Family Construction Technique to a Generalized Construction Technique."**

GUI operation base:

When you think that something may be calling some of the event procedures of a **GUI** application program, that something is the **GUI operation base**. Put in another way, the **GUI operation base** is something that calls the event procedures and determines when each should be carried out.

A **GUI** (graphical user interface) is a method of visual operation using GUI controls (also known as widgets), such as buttons, menu items, list boxes, or text boxes laid out in a window.

The argument for this term appears in **2.2.2-m "Object Orientation and GUI Operation."**

Improvement rate of productivity by reuse:

The index that expresses how much productivity is actually improved by reuse. An approximate value of the improved rate of productivity by reuse can be obtained by dividing plain productivity by compensated productivity.

The utilization of 'Business Logic Component Technology' is a typical example of improving productivity by reuse.

Reuse in this sense can be classified into two categories; reuse by copying and reuse by referencing.

The argument for this term appears in **4.4-l "Improvement Rate of Productivity by Reuse."**

Need for Creative Adaptation (NCA):

The extent to which creative adaptation is needed. Creative adaptation means managing various requirements in a certain applicable area or field by creating new programs when it is not feasible to depend on the passive means of specifying existing parameters. Requirements are met by parameter customization in areas in which the need for creative adaptation (NCA) is low; however, they are not in high NCA areas. In areas like that, the adaptation by creating new programs, or in other words program customization, is needed. Parameters here mean declarative information with clear-cut meaning when viewed externally, like parameters of parameter customization.

Generally, NCA is high in the business field in which creative innovations frequently occur. Yet, as an exception, NCA is not very high in financial accounting since laws and regulations restrain creative innovation.

The argument for this term appears in **5.1-b "Second Requirement for Practical and Effective Component-Based Reuse Systems."**

Reuse by copying:

The reuse method that utilizes the copies of original software. For software reuse methods, there is reuse by referencing, besides reuse by copying.

An example of reuse by copying is the development in which a program model is distributed, and then some modification is made to the copy of it. In addition, the development that employs pre-generators (software tool) is a type of reuse by copying.

Reuse by copying superficially seems to accelerate development processes and improve plain productivity; however, it tends to generate programs inflated with bloat and swell maintenance loads.

The argument for this term appears in **4.4-n "Two Methods for Improving Productivity by Reuse."**

Reuse by referencing:

The reuse method that uses original software by specifying and referring to it, but without copying it. In software reuse method; there is reuse by copying, besides reuse by referencing.

An example includes building and utilizing general subroutine libraries to their fullest. The development by post-generators (software tool) is a type of reuse by referencing as well.

With reuse by referencing plain productivity is not improved since it does not increase the program size. Nevertheless, it can achieve the same outcome as when productivity is improved. Then by getting rid of bloat assets, it reduces the amount of software resources that need maintenance.

The argument for this term appears in **4.4-n "Two Methods for Improving Productivity by Reuse."**

Reuse systems of componentized applications (RSCAs):

The component-based reuse system that reuses componentized applications. For the componentized application, an application program is cut into pieces, each of which functions as a 'Business Logic Component.'

In this book, I mainly focus on the component-based reuse system exclusively in the form of reuse systems of componentized applications (hereafter RSCAs).

SSS and the **RRR family** are examples of the products composed of reuse systems of componentized applications.

The argument for this term appears in **2.2 "Reuse Systems of Componentized Applications and Object Orientation."**

RRR family:

A successor of **SSS** that uses 'Business Logic Component Technology.' It adopts the componentized event-driven system, a new system that did not exist when **SSS** was developed. It also has advanced to ERP packages that are used for sales management, financial accounting, and manufacturing management. **RRR** consists of **RRR tools** and **RRR component sets**.

The argument for this term appears in **3.2.3-q "Improvements for RRR Family."**

Seeds:

The policies, facilities, structures, reasons, ideas, and so forth that improve software development productivity. To improve productivity, we should take up the most effective seed after evaluating the effectiveness of each of them that have been raised as candidates. In order to do this, it is required to calculate the improvement rate of productivity for each of the seeds by a certain method, for example, the build-up method.

Some examples of seeds, that have been around for a while, include the supporting functionalities for address calculations by an assembler, the conversion from a programming language to a machine language, the supporting facilities for the task of writing diagrams, and the extraction of information possibly usable as documents out of programs, to name but a few.

The argument for this term appears in **4.2-e "Implementation Verification for Determining Improvement Rate of Productivity."**

Software components:

The elements that constitute a component set used for software component-based reuse systems.

A general subroutine is a representative example.

In this book, the software components with the particular qualities are noted as 'Software Components.' 'Software Components' involves 'Black-box Components,' with generality, and 'White-box Components,' which possess all of the four qualities of retrievability, locality, suitable size (suitable granularity), and readability.

Some examples of 'Software Components' are general subroutines, general main routines, and data item components.

The argument for this term appears in **5.3 "Meaning and Significance of 'Business Logic Components'."**

Software component technology:

The technology that uses 'Software Components.'

In this book, the software component technology that uses the 'Software Components' with the prescribed qualities are expressed as 'Software Components.'

With this technology, component customization can be used that improves productivity of program customization, so customization work becomes extremely easy.

The argument for this term appears in **5.3 "Meaning and Significance of 'Business Logic Components'."**

SSS:

The first business packages with special customization facilities aimed for sales management operations (sales management activities) developed with 'Business Logic Component Technology.' Consisting of about 2,000 pieces of data item components, it was characterized by its customization facilities. **SSS** is comprised of **SSS tools** and **SSS component sets**. The **RRR family** is the successor of **SSS**.

The argument for this term appears in **1.3-i "Program Partitioning with Data Item Association."**

Two-stage customization:

The customization method made by taking up only the best traits of both parameter customization and program customization.

With two-stage customization, ordinary customization requirements are managed by parameter customization, whereas program customization is used as the trump card only when the parameter customization method fails.

In this way, regular customization tasks are easily done by parameter customization, while at the same time any kind of customization requests can be handled by program customization.

The argument for this term appears in **1.1-b "Customization Methods."**

White-box component:

The component that has the characteristics that the decipherment of its inside program may be needed. Alternatively, a white-box component is one for which program customization in the broad sense may be needed, and its source program is disclosed.

In this book, the white-box component with all the four qualities of retrievability, locality, suitable size (suitable granularity), and readability - is expressed as a 'White-box Component.' It is a kind of 'Business Logic Component.'

The data item component is a representative example of a 'White-box Component.' Also, the program customization for 'White-box Components' is especially called component customization.

The argument for this term appears in **5.2-f "Generalized Construction Technique for a Reuse System of Componentized Applications."**

CHAPTER 1 Custom Business Programs and Business Packages

This chapter will contrast **custom business programs** and **business packages**, while looking at the actual state of business program development in the business field. It will then introduce an actual example of a business program, developed using **'Business Logic Components'** and clearly show that this is positioned between custom business programs and business packages.

The discussion on business packages in this chapter will focus on **component-based reuse**, the main theme of this book, due to the pioneering role of business packages in reuse. The unit of reuse implemented so far has normally been large products, such as business packages or small functions like general subroutines. In addition to these, this chapter will introduce 'Business Logic Components' that cover an area not covered by general subroutines. Unlike large business packages, 'Business Logic Components' offer the benefit of being able to be made into a finished product by being combined with various small components.

1.1 Differences between Custom Business Programs and Business Packages

By focusing on what sort of business they support, business systems in the business field can be classified into many types, including financial accounting, sales management, inventory management, customer management, production management, and so on. However, even when you are talking about a production management system, there is a major difference between a business system in charge of semiconductor production management and a business system in charge of automobile production management. Consequently, production management systems are sometimes seen as different things depending on the industry. Based on such industry and business differences, if you were to count the number of types of such business programs, the figure could reach several hundred types depending on the counting method.

However, the number of business programs that are actually being developed far exceeds this number. This is because in many cases they are developed as custom business programs for various enterprises. This is due to the fact that even enterprises in the same industry, which are conducting the same business, differ in how they think about business processes and their procedure details.

Custom business programs are frequently likened to tailor-made clothing, as are **business packages** to ready-made clothing. This is because customer business programs are developed to fit in perfectly with the concepts and procedures for the business processes of a specific customer (single enterprise). On the other hand business packages are developed in an attempt to fit in with the concepts and procedures for common business processes of many enterprises. This is the major difference between custom business programs and business packages.

1.1-a Customization Required by Business Packages

Enterprise business procedures generally have a lot in common, but they also have company-specific portions. There is a great deal of variation in actual business procedures due to such factors as the **enterprise characteristics** of each company, including its historical background, executive policy, interdepartmental dynamics, and special methods of conducting business; special **industry characteristics** including the types of products and services handled by each industry, industry practices, and other matters of concern; and creative efforts to differentiate the company from another. Consequently, each enterprise desires a special business program that fits in with its own concepts and procedures concerning business processes.

However, since business packages are developed in order to satisfy the greatest number of users, they cannot always support all variations. As a result, when you try to introduce a business package to a specific client (single enterprise), you will find certain parts that do not fit in. In other words, among special business packages focused on a specific business, as are sales management system packages, it is usual to find some sort of inconformity arising from differences in **industry characteristics** (same business but different industry) and **enterprise characteristics**. In addition, among industry packages focused on a specific industry, as are hospital packages, it is usual to find some sort of inconformity arising from differences in **enterprise characteristics**.

This is why adaptation work (likened to customization or fitting, and hence, known as tailoring) of a business package to the business procedures of a specific customer (single enterprise) is required. Even though there are **hard-wired packages** that cannot be customized, most business packages were designed with some sort of customization in mind. In short, a customization mechanism is built in to adapt the business package to a new customer environment.

1.1-b Customization Methods

Customization methods can be essentially classified as one of the following two types:

- **Parameter customization**
- **Program customization**

Parameter customization is simple because customization work concludes merely by specifying **parameters** with clear-cut meanings when viewed externally. To make parameter customization possible, however, the extent of customization must be established in advance, and then what is required to support this must be **built in**. For instance, if you assume that either the last invoice cost method, moving-average method, periodic average method, or estimated cost method will be used as the appraisal method for inventory, what is required to support it must be built in. It is also possible to specify the appraisal method for inventory that a customer actually employs by the parameters you give to a business package.

By building in what is necessary, customization work becomes extremely simple because all that needs to be done is to specify **parameters**. Nevertheless, customer requests that can be satisfied by this method are limited to the extent that the business package development firm can only specify such requests as parameters by making assumptions ahead of time.

This method could be called a sort of true/false or alternative form exam question.

Program customization is the satisfying of customer requests by changing or creating new parts of a program module within a business package. Consequently, any request can be met in principle. However, customization work tends to be a big job because it involves cutting deep into the procedures within a program, rather than simply specifying declarative information with clear-cut meaning when viewed externally like parameters. It is sort of like surgery for reconstructing internal organs, and the amount of work required for this is from 100 to 10,000 times, that of parameter customization.

If all customer requests could be satisfied using parameter customization, work would be extremely simple. Unfortunately, the extent to which customer requests can be met with parameter customization alone is limited. In a manner of speaking, it is impossible to create and add new functions to internal organs simply by adjusting parameters. As a result, program customization’s raison d'etre lies in its ability to meet any sort of request.

This method could be called a sort of written exam problem.

Two-stage customization applies the best portions of the two-customization methods. This two-stage method uses parameter customization to meet frequently found customization requests, and when this

method cannot meet such requests; it uses program customization as a sort of secret weapon. In short, it is a sophisticated method that allows some customization to be easily completed while at the same time conferring flexibility that enables any sort of customization requests from customers to be met.

There are some things worth noting about the terms used here. There are business packages out there that are said to be able to meet almost all customization requests simply by specifying *parameters*. It must be noted that the term *parameters* used in relation to such business packages, and the term "parameter" as used in parameter customization, specify completely different things. The term "parameter" in this book means declarative information with a clear-cut meaning when viewed externally. In contrast, the other *parameter* is information that depends on the internal structure of a business package, and it is usual for procedural information other than declarative information to be included. In a manner of speaking, these kinds of *parameters* are an internal language (a kind of programming language) for writing business packages; hence, this book regards the work for setting such *parameters* as program customization. Normally, as previously stated, amount of such *parameter* customization work is actually 100 to 10,000 times that of parameter customization in this book. Consequently, even though both terms use the same word, they are actually two different things. *Parameter* customization, which some vendors are claiming to be simple and easy, is another kind of program customization, which is not easily completed.

1.1-c Custom Business Program or Business Package? (Part 1: General Discussion)

A business package is intended to be sold to many enterprises rather than a specific customer (single enterprise). Hence, its development requires unique know-how and considerations, and also runs up development costs beyond a custom business program. Therefore, business programs that can only be sold to a specific customer (single enterprise) are normally developed as a **custom business program**, rather than choosing the business package style.

In contrast, a **package style** is advantageous for development of those industries and business field business programs, which can be sold to many customers (enterprises), and have few variations in customization requests. If you were to estimate the total cost of hypothetically developing individual custom business programs for each customer (enterprise), a business package that can be developed once and used by many customers (enterprises) clearly results in a dramatic cost reduction.

Topic 1: Dreaming of the"Golden Egg"Business Package

Even in the business field, financial accounting is special for reasons discussed later, and its variety in connection with customization requests is limited. This sort of application field is suitable for business packages and can meet most requests simply through parameter customization.

To raise the parameter customization rate (percentage of customization that can be met simply by setting parameters) of a business package, it is essential to collect sufficient information about the customization requests. For example, in the process of evolving into an organism that can adapt to a diverse range of environments, genetic information for adapting to each environment must be obtained. In the exact same manner, the evolution of a business package that can adapt to a diverse range of environments requires information about customization requirements.

In reality, there are also exceptional cases where the parameter customization rate for a financial accounting package reaches nearly 98 percent by using information-gathering tactics. Such information-gathering tactics are based on the principle of having a package customized simply by setting

parameters, without passing the source program to the person in charge of customization. Whenever program customization is absolutely necessary, the source program will be disclosed in the exchange of detailed information about that customization case. Doing this makes it possible to widely collect detailed information about customization requests. If information can be collected, the later application of development funding and development time will enhance the business package and enable it to evolve into something that meets the newly understood requests by means of parameter customization.

Business packages with a high parameter customization rate can be thought of as equipment that requires a large capital investment. As it is like developing an automatic sewing machine for tailor-made custom clothing, a large initial investment is required. However, the equipment will produce a large profit as long as it is operated effectively. This is because like the goose that laid golden eggs, the simple setting of parameters will allow the production of business programs one after the other, as long as the business package can be applied.

Creating such a goose that lays golden eggs is not possible for just any business program. Actually, it is only possible for business programs for application fields in which the **need for creative adaptation** (NCA) is low. (Chapter 5 discusses NCA in detail, but for now, refer to the explanation given in **"Keywords for Understanding This Book."**) With business programs for application fields in which the NCA is high, new customization requests that cannot be met, no matter how many parameters you add, will arise, and hence, it is impossible to adequately raise the parameter customization rate. Each time you encounter a new customer (enterprise), what is required must be built in, and maintenance for responding to continually arising needs is constantly required. Therefore, the creation of a goose that lays golden eggs is generally difficult. It is only possible in the case of application fields for which NCA is exceptionally low, such as with financial accounting packages. In the financial accounting field NCA would be low, because there is the constraint of legal provisions that must be obeyed and creative activities are restricted.

Note that the use of 'Business Logic Component Technology' enables the creation of business programs (business packages) that come close to a goose that lays golden eggs, as described in Chapter 5, even in business fields with high NCA.

1.2 Custom Business Program and Business Package Development Firms

This book has so far focused on technical matters, but now let's change the viewpoint and expand the discussion from the perspective of development firms that are making an all-out effort to raise profit. This point of view is crucial to the understanding of what is really happening.

1.2-d Business Package Development Firms

The profit of business package development firms is largely affected by the number of business packages they sell. Consequently, they focus their efforts on selling as many business packages as possible and position those that will likely sell well as their flagship products. In addition, since they can lower the price if a business package sells well, they can expect a further increase of users through a low-price strategy.

Under such circumstances, it is thought that the flexibility of being able to meet all requests is crucial to increasing the number of business packages sold, but in reality, this is not always the case.

Employing the two-stage customization method is the best way to create business packages that can meet the various customization requests from a variety of customers. However, business package development firms almost never employ this method for their flagship business packages. This is because the requests of

the majority of enterprise clients can obviously be met using parameter customization only, and hence, the remainder is the minority. Even if they were to perform program customization for this minority, the expected increase in units sold would in most cases be miniscule compared to the amount of time and effort required, which is why the two-stage customization method is not employed. The author would like to remind you, that the time and effort for program customization is of a different order of magnitude, anywhere from 100 to 10,000 times, compared to parameter customization. In light of this gap, you can understand why business package development firms shy away from program customization.

Now if you were to say business package development firms exhaustively build in what is necessary to meet all requests through parameter customization, you would be wrong. Necessary development stops at a happy medium. A look at this situation reveals that these firms build in what is necessary, starting from areas in which there seems to be the most enterprise customers that will benefit from the additional development (in other words, areas in which there seems to be a chance to increase the number of units sold); hence, they can only get so far before the returns start thinning out. That is the time when the work of building in what is necessary stops. Rarely do firms purposely expend the time and effort to build in what is necessary to meet requests for which they can only expect a miniscule increase in profit.

Rather than such a long explanation, you might find the following shorter one easier to understand. **The work of building in** what is necessary to enable parameter customization takes more time and effort than program customization, and that is why it stops at a happy medium.

Note that there is another reason why the work of building in what is necessary is not carried out exhaustively to enable parameter customization. This is described in **5.1** "**Requirements for Practical and Effective Component-Based Reuse Systems**."

As described so far, business package development firms do not strive to be able to meet all requests through customization. Rather, they usually place an emphasis on sales techniques and customer guidance that leads the customer to a position in which they can meet requirements through parameter customization. In reality, this guidance policy has a major effect on the number of business packages sold.

This should have been mentioned earlier, but the discussion so far assumes that customization service is included in the business package price. If business package development firms were to separately charge for customization service, the discussion would likely head in a different direction.

However, once business package development firms start strongly advocating a switch to fee-based customization services, they may not be able to avoid fostering customer expectations for customized business programs. Once this happens, the switch to fee-based customization services will become incompatible with the guidance policy of business package development firms that are attempting to sell without customization. As a result, you almost never find cases of business package development firms making a business out of customization service. Business package development firms are oriented towards business that does not require constant monitoring, which is an area that has nothing to do with customization.

1.2-e Custom Business Program Development Firms

Custom business programs for enterprises are sometimes developed in-house by the enterprises themselves, but usually it is the computer manufacturer or dealer that develops and delivers them along with a computer. In this case, the fee basically depends on the labor cost for development. In short, the fee for contracting the development of a custom business program is usually decided based on an estimate in man-months, a hypothetical unit of measure for how long it would take one person to develop a program.

However, since there is a vast difference between the amount of work each developer can manage and the quantitative analysis for business program specifications is difficult, the actual number of man-months required for development varies widely. Consequently, contract work for developing business programs is very risky. Losses may be incurred, but there are also instances of unexpectedly high profit. In addition, averaging out a large amount of contract work mitigates variability, resulting in the difficult business of being able to produce a profit one way or another. Nevertheless, computer hardware sold along with business programs, was previously more than feasible as a business because it was possible to make a good profit in a stable manner, at least until the open movement started gaining momentum.

Under such circumstances, increasing the number of developers and the amount of contract work for developing business programs is considered crucial. By doing so, variability is statistically mitigated, and this leads to increased sales of hardware, which has a large profit margin.

In addition to the above-mentioned approach of increasing both developers and work, there is also a work saving strategy of increasing only the amount of work without increasing the number of developers, due to the rationalization of development work for custom business programs. However, rationalization is not advancing in environments where estimates in man-months directly link with sales. You could easily say that custom business program development firms do not feel that they have to implement cost reductions, no matter what. For instance, even when they did try to come up with a rationalization plan, it was conspicuously lax. Wagering a company's fate on rationalization is extremely rare.

Generally, rationalization does not readily progress without strong external pressure. There was once no pressure whatsoever, however, with the advance of the open movement and price reductions for hardware, both of which exert downward pressure on profits, even custom business program development firms are being forced to rationalize.

Now let's take a more in-depth look at custom business program development firms by categorizing them either as system integrators or firms that carry out development work on a subcontract basis under such integrators. In addition to computer manufacturers and dealers, custom business program development firms also include the software development companies that actually develop the programs that the others order. It was once usual for computer manufacturers and dealers to have software development companies develop custom business programs, and then take on the role of system integrator (put together a system comprised of hardware and various pieces of software) themselves.

Before the advance of the open movement, the information necessary for development flowed from computer manufacturer to dealer, and then to software development companies, which put software development companies in a weak position. However, control by information weakened after the advance of the open movement. That is why powerful software development companies are increasingly receiving direct orders from customers. We have entered an age where the various abilities of both system integrators and software developers are increasingly coming under question.

1.2-f Reuse

Unexpectedly high profit can be made in the contract development of custom business programs. For instance, there are cases where a software development company receives orders for similar business programs. In such cases, cleverly allocating the same developers to that work enables the adoption of a measure referred to as the conversion or appropriation of programs. Appropriation in this case refers to a developer reusing a custom business program developed for Company A to develop a similar custom business program for Company B. This is also known as duplication, but because of its negative

connotations, as is the case with many other words, I will hereafter use the term **reuse**, which does not harbor such a negative meaning.

Reuse is generally carried out on a personal basis. It is usual for a certain developer to reuse a program he or she has developed. If reuse could be carried out systematically, it would serve beautifully as a rationalization plan for development work. However, when you consider the time and effort it takes to decipher a program so that it can be reused, it is usually better to develop programs from scratch. One theory holds that it is best not to reuse a program written by someone else unless eighty percent or more of it can be used exactly as is without having to be deciphered. For that reason, rationalization plans for systematically implementing reuse have not been exhaustively pursued.

1.2-g Custom Business Program or Business Package? (Part 2: Cost of Customization)

From the standpoint of business program users, the question arises, which is more advantageous, using a business package or ordering a new custom business program? If they could find a business package that exactly fits their needs, the answer would certainly be a business package. However, that is not always possible to find. That is why some form of customization becomes necessary, and the resulting expense is a villain of sorts.

There is a funny, yet unfortunate anecdote related to this. A certain consulting firm (not mine) once said that it was more expensive to customize a certain business package for one of their customers (an enterprise) than it would have been to develop a custom business program from scratch.

Generally, the applicable scope of a business package is fixed, and the moment you try to deviate from that scope, the cost of customization will start increasing. A typical example of this is subsequently switching to program customization after parameter customization could not adequately meet customization requests.

Accurately estimating the cost of customization is vital to the application of a business package, and failing to do so will lead to a mistake being made in choosing between a custom business program and a business package.

In conclusion, when deciding whether a custom business program or a business package is advantageous from a business perspective, making an assessment that takes into account the cost of customization is crucial. This is a conclusion from a user standpoint. At the same time, this is also the reason why the markets for business package development firms and custom business program development firms are separate.

1.3 Business Packages with Special Customization Facilities

This section will introduce the products of Woodland Corporation, which has now been merged into FutureArchitect, Inc. This company occupies a unique position among custom business program development firms. Woodland Corporation was a medium-sized enterprise that delivered over 10,000 business systems as an office computer dealer (a mid-range computer dealer), but unlike other custom business program development firms, it was completely involved in the rationalization of development work. The company has also developed business programs that use ‘Business Logic Components.’ Such programs should be referred to as business packages with special customization facilities, which are located between custom business programs and business packages. Let’s take a look at the features of these business packages with special customization facilities.

1.3-h Woodland Corporation's Efforts

Woodland Corporation got involved in the rationalization of development work for custom business programs because they thought they had problems in the following two areas:

- **Real variation in development cost; and**
- **Labor-intensive development**

Based on their prior development experience, they knew that although custom business programs also have company-specific portions, common portions are more numerous, and hence, they pursued research based on their hunch that there must be a better way.

If they could standardize program portions related to common specifications and then adjust only the program portions related to each company's variety of standards, development work for custom business programs could be rationalized. However, this is easier said than done. There is no clear demarcation between common portions and company-specific portions, and such demarcation is by no means easy to do. (To use the latest terminology, such demarcation is the compartmentalization of **frozen areas** and **hot spots**.) Consequently, the ability to cope, no matter where company-specific portions are, was necessary, and they had to rethink the **correspondence between programs and specifications** from scratch.

For reference sake, let's take a look at the current state of reuse being spontaneously carried out by individuals. For instance, when reusing a custom business program originally for Company A to remake a new one for Company B, we know that by focusing on the portions wherein specifications differ between Company A and B, all that is required are program changes related to those portions. Therefore, development becomes markedly easier than having to develop all new business programs.

It would be wonderful to enable systematic reuse on a personal basis. However, it is not easy to make reuse systematic. Only the developer knows the correspondence between a program and its specifications. Therefore, it is not easily understood by others. Comprehending this correspondence requires you to become the student of the developer and decipher the program from inside out. This enables you to figure out where you should make changes just as if it were a program you developed yourself. However, since programs cannot be as easily deciphered as reading a novel, it takes an extremely long time to get acquainted with them. Knowing this, investment towards getting acquainted with programs may be another rationalization plan.

Woodland Corporation concluded that this would not be an essential solution. Even if they were to apply this rationalization plan to a so-called "spaghetti program," a disorganized mass of code, it would inevitably lead to failure. Accordingly, they headed in the direction of making the correspondence between programs and specifications easier to understand by devising a structure for business application programs.

1.3-i Program Partitioning with Data Item Association

In reality, the company faced many detours through trial and error. However, since it would be difficult to follow that account, let's set it aside so that we can get to the heart of the matter.

In the 1980's, they decided to start partitioning programs in accordance with **data items**, not only because of the focus on data-oriented approaches (concept of emphasizing data items), which was also garnering attention in Japan, but also because it made good sense based on past development experience. Past experience made them realize that over eighty percent of each company's variety of standards was related to data items in one form or another.

They were inclined towards partitioning with data item association because they thought it would allow not only the developer, but also anyone else to understand the relationship between a program and its specifications. This is due to the fact that in the business field, specification change requests are represented using data item names. Please also refer to **Appendix 2** "**General Features of Business Applications in the Business Field**."

For example, a data item named "product unit price" appeared in a specification change request in the form "we want to change the method for computing 'product unit price' discounts." They focused on the fact that when implementing the customization, as in this example, they should be able to meet the request simply by changing the program portion corresponding to the data item "product unit price" that they could extract.

After pursuing this idea in search of how to partition programs, they found that there were also a few portions that would not correspond with data items, but by using their ingenuity they still succeeded in partitioning most in accordance with data items. They were able to realize their idea by gathering program fragments that extracted portions corresponding to each data item and then standardizing them (forming them so that they fit in with a certain template).

As a result, they were able to make data item correspondences in seventy to eighty percent of business programs, and the correspondence of each program fragment with specifications became clear just as they had intended. Moreover, each fragment became a closed block that could be understood by itself. In addition, almost all were one hundred lines or less. On top of that, the effect of fitting in with a template was gained, and each fragment became easy to decipher because they took on a stereotypical format. Finally, they made it easy to search program fragments by making the name of each one start with the name of the data item they contained, and no large-scale search system was required to do this.

Note that a program fragment corresponding with a data item has the qualities required of 'Business Logic Components' that will be discussed in Chapter 5, and hence, this book refers to them as **data item components**.

Woodland Corporation used such data item components and **'Business Logic Components'** to develop a synthesis system for business apps aimed at a sales management system. They named it SSS (triple S). This system is perhaps the first to be comprised of 'Business Logic Components' that fit the definition given in this book. The combination of these components enabled the production of sales management systems that were exactly what they expected.

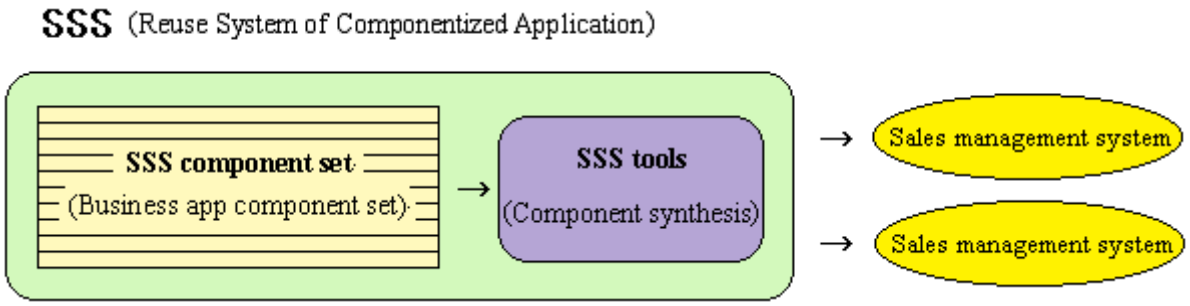


Figure 1-1: SSS Components and Component Synthesis

There are some approximate boundaries between 'Business Logic Components' and component synthesis tools, but these are not clearly separated in SSS. However, since they can be conceptually distinguished in a clear manner, I will refer to them as **SSS component sets** and **SSS tools** as shown in **Figure 1-1.**

The first time I saw this system, the only **SSS component set** offered was one for sales management operations (sales management activities). However, I was able to clearly predict that they would be able to use the same component synthesis tools (SSS tools) to produce business systems, as long as they developed component sets for other businesses. Since this component-based reuse system is actually a mechanism that effectively functions in the business field, enhancing component sets will lead to their development into **ERP packages**.

1.3-j Custom Business Program or Business Package? (Part 3: Conclusion)

If you look back over time, you will find there tends to be a polarization between custom business programs on one-end and business packages on the other. This polarization causes the following problems.

Business packages are reasonably priced because of the extensive reuse involved, but they are geared towards the avoidance of customization. Few efforts are being made to create business programs that fit exactly. If you want something that is a perfect fit, you must order a custom business program. However, since custom business programs employ almost no reuse, which means they are developed nearly from scratch, they are not reasonably priced.

Accordingly, people end up wanting what are called **business packages with special customization facilities**, which combine the best of business packages and custom business programs, are moderately priced, and fit perfectly.

If you were to try to elaborate a plan for such a system, you would realize that areas that could not be dealt with using parameter customization when adopting two-stage customization should be designed so that they could also support special orders through program customization.

The story so far could stand on its own, but then the cost of customization would be a problem, and it would not be possible to keep prices within a reasonable range. There is a problem with using program customization to deal with areas that cannot be dealt with using parameter customization. The problem resides in the fact that there is a high customization cost (two or more figures higher). Therefore, it can be seen that reducing the cost of program customization will become an issue that must be resolved.

The ‘Business Logic Component Technology’ developed by Woodland Corporation is one answer to this issue. When performing program customization using this technology, the applicable scope is limited to a number of data item components. For instance, even if a business application program had 100,000 lines in all, you would only have to focus on data item components in about 1,000 of the lines when performing program customization. Furthermore, this work is much simpler than customizing the usual 1,000-line program because the data item components are standardized.

Conventional program customization work required the detailed examination of some 10,000 lines out of 100,000, followed by touchups here and there throughout that wide region. In addition to the large number of programs lines, this sort of work is difficult because you have to be very careful because of the high possibility of adverse effects on program operation. The use of ‘Business Logic Component Technology’ enables the amount of program customization work to be vastly reduced compared with such conventional work.

Since there is at least a single-digit reduction in the amount of customization work, program customization for business packages with special customization facilities using **data item components** must be thought of as a completely different beast compared to what came before. Consequently, this book refers to it as **component customization**.

Table 1-1: Custom Business Programs, Usual Business Packages, and Business Packages with Special Customization Facilities

Evaluation Point / Type of Software	**Support for Special Orders from Customer**	**Cost Burden on Customer**
Custom business program	Can be supported in every way because the program is custom built.	Large burden because a single customer incurs all development costs.
Business package	Often results in support within the range possible with parameter customization.	Small burden on each customer because a single package can be reused for many customers.
Business Packages with Special Customization Facilities	Can be supported in every way because of component customization.	Small burden on each customer because a single package can be reused for many customers. However, the cost burden of component customization is inescapable.

In conclusion, the author would like to present the comparison shown in **Table 1-1 "Custom Business Programs, Usual Business Packages, and Business Packages with Special Customization Facilities"** and summarize this chapter using the keywords described thus far.

Since there is such a huge cost gap between **parameter customization** and the conventional method of **program customization** for business packages, business package development firms have adopted a strategy of avoiding program customization. These firms have headed in the direction of the polarization of **custom business programs** on one-end and **business packages** on the other.

However, since **component customization** can reduce this gap and in a sense enable a cost reduction in program customization, it is possible to produce business packages with special customization facilities that fuse a custom business program with a business package. In addition, a custom business program or business package can fulfill the aim of taking the best of each, being reasonably priced and fitting perfectly.

It is therefore easy to conclude that **business packages with special customization facilities** are the answer to the question "custom business program or business package?"

CHAPTER 2 Component-Based Reuse and Object Orientation

The concept of object-orientation has a long history. It is said that its origin can be traced back to the advent of the key concepts of object, class, and inheritance in the simulation language known as Simula67 in the latter half of the 1960's. Combined with the promotional activities for the object-oriented programming (hereafter OOP) language Smalltalk in the 1980's, object-orientation became a catch phrase for resolving software development-related problems.

This chapter will report as honestly as possible about how object-orientation has been perceived by looking back at the process of examining the component-based reuse system, as carried out by Woodland Corporation and AppliTech Inc. in 1994. This was the start of development for the RRR (Triple R) family, the successor to SSS.

2.1 Smalltalk System and SSS

Woodland Corporation's SSS (Triple S) incorporated the good aspects of the concept of a **data-oriented approach**, but it was developed independently from the concept of object-orientation. However, SSS did have striking similarities with software development in the object-oriented system known as Smalltalk.

2.1-a Software Development on Smalltalk System

It is more accurate to perceive the Smalltalk system as a component-based reuse system that encompasses objects (components) that form a hierarchy rather than a mere OOP language. Actually, experts that can comprehend the objects contained therein, and how they were structured, will be able to immediately develop certain types of visual application programs.

Software development on the Smalltalk system involves the modification of some objects. This choice of words accurately represents the actual state of object reuse. Development consists entirely of modifying declarations of objects and procedure portions or registering modified objects as new objects. In a manner of speaking, it is a system that makes extensive program customization extremely easy. Since object structure in the Smalltalk system was designed after careful consideration, experts who understand that will know where modifications should be made when developing certain types of visual application programs. They also will be able to transform the Smalltalk system into exactly the system they intended simply by adding, modifying, and/or removing some objects.

2.1-b Customization on SSS

SSS, on the other hand, is a **business package with special customization facilities** for sales management operations (sales management activities) in the business field. SSS includes component sets and their synthesis tools, and these two items comprise a component-based reuse system. Over 5,000 business programs developed using SSS (in short, created as a result of customization) have already been shipped up to 1995.

The strength of SSS lies in the application of a new idea for making customization easy. This is crystallized in the nearly 2,000 SSS component sets in the form of the previously mentioned **data item components** and their synthesis tools.

The foundation of customization using SSS tools is the selection of data item components. Although I used the term "select" here, it is not a matter of specifying 2,000 components one by one. Since standard components are selected from among component sets in advance, resulting in a business program that

actually operates, you need only to replace a number of the standard components, which do not fit customer requirements with appropriate data item components. This is the core work in customization.

The special feature of the data item components comprising SSS component sets lies in the fact that each one corresponds with a single data item. Since specification change requests are represented by data item names in the business field, it is easy to find the data item component you want from among the 2,000 provided. This is due to the fact that each data item component is assigned a name that begins with the name of the corresponding data item. Please also refer to **Appendix 2** "**General Features of Business Applications in the Business Field**."

The use of such a naming method using data item names clearly points out which data item components should be replaced or removed when some sort of modification is required. Furthermore, you need only make a single selection from among several candidates categorized by data item names when deciding which data item components to add or use as a replacement. The simple selection of appropriate components in this manner enables the synthesis of a sales management system as intended.

Customization in a business program, which is better fit for specific needs, requires both the internal modification of data item components and new development. Therefore, this involves extensive program customization. During such work, the development of new data item components normally consists of selecting similar data item components from a component warehouse, partially modifying them, and then registering them as new data item components. Obviously, reuse is fundamental to this sort of customization work as well.

In short, the registration of new components and modification of some others strongly resembles program customization work in the Smalltalk system. As previously discussed in **1.3** "**Business Packages with Special Customization Facilities**," this book has specially designated this sort of customization as **component customization**.

2.1-c Ingenuity of SSS Focused on the Business Field

If you take a look at the reuse phase of componentized software, you will see component customization is taking place both in the Smalltalk system and in SSS, and the work method in both is nearly the same. It should be noted, however, that SSS has specialized ingenuity only for parts limited to the business field to which it is applicable.

First of all, the ingenuity of SSS tools lies in making selection possible from among several candidates prepared in advance because it makes modification work for 'Business Logic Components' even easier. Of course it is easy to establish a similar selection mechanism for objects in the Smalltalk system as well. However, you cannot focus on which objects to make candidates unless you limit the applicable fields. To make selection possible from among candidates, a collection of object and "business logic candidates" is crucial. Since the selection mechanism consists of nothing more than minor ingenuity that almost anyone could think up, which the Smalltalk system will eventually have, the selection mechanism is easy to develop. In contrast, a collection of candidates can only be created through experience, knowledge, and insight related to the kinds of customization that are actually possible.

Secondly, the ingenuity of SSS component sets lies in the extreme ease of identifying components that must be modified. This is because it adopts 'Business Logic Components' corresponding with data items as its main components. In the Smalltalk system, it is not easy to identify the objects that must be changed. Identifying these objects requires an expert who comprehends the structure of objects forming the hierarchy contained therein and the contents of each one. This means it will never be as easy for non-experts as SSS is.

In SSS, the good aspects of the shock therapy-like concept of a data-oriented approach deemed to be effective in the business field were incorporated, and 'Business Logic Components' corresponded with data items. As a result, the work for identifying the 'Business Logic Components' that must be modified became easy. Note that the methodology of "associating such components with data items" is not guaranteed to work as well in non-business fields, as you can imagine from such a background, but at least it works effectively in the business field.

As a general trend, it seems the Smalltalk system and object-oriented approaches overlook individual improvements focused on specific fields, most likely because they emphasize universal principles and rules. SSS, on the other hand, delves deep into any one of many fields.

2.1-d Applicable Fields for Smalltalk System

Let's explore the kinds of fields to which Smalltalk is applicable.

If you look at Smalltalk as merely an OOP language, you will see that it aims for generality rather than targeting specific fields. Conversely, if you look at Smalltalk as an object-oriented system that includes objects forming hierarchies, it has an orderly, systematized structure based on the partitioning guideline of Model-View-Controller (MVC), a three-way partitioning method. The applicable field is highlighted by the structural method and the objects that conform to it. Although it is difficult to represent what this field should be called, it is possible to immediately develop certain types of visual application programs.

However, the present Smalltalk system cannot easily develop business programs in the business field. What do you suppose might happen if objects for the business field were to be included in the OOP language of Smalltalk? For example, we could try to make a Smalltalk system for production management business or a Smalltalk system for sales management operations (sales management activities) like SSS. Conceivably, we could end up with a new Smalltalk system for the business field. I would like to attempt something like this if given the chance.

2.2 Reuse System of Componentized Applications and Object Orientation

This section takes a look back at the investigative process before starting the development of the RRR family, the successor to SSS, while reporting how object-orientation was broken down into an easy-to-understand form at one development site.

Since SSS is a "component-based reuse system" that has similarities with the Smalltalk system, we harnessed this strength to carry out an investigation into the RRR family in an attempt to create a system that was more refined than SSS. The concept of object-orientation was a strong focus of our work. To make it acceptable to the greatest number of people, we adopted a policy of conforming to currently popular concepts and tools as much as possible. However, it goes without saying that, among the achievements of SSS, we attempted to carefully maintain and foster anything for which no previous examples existed.

The greater part of the investigation into refining this "component-based reuse system" followed the concept that perceiving the essence of things was more important than separate commercialization. Consequently, it became not only an investigation for a product, the RRR family, but also an investigation into figuring out what a practical and effective "component-based reuse system" should be like. We eventually named the system we were aiming for a **reuse system of componentized applications** (hereafter **RSCA**).

Hereafter, the product name “RRR family” and the generic term “reuse system of componentized applications” or RSCA will appear. In short, please think of the RRR family as one example of developing an RSCA into a product.

In addition, a fair number of pages hereafter are devoted to a general description of object-orientation and its peripheral technologies, but this is not the real purpose of this chapter. I would like to once again emphasize the fact that this chapter will report how we perceived object-orientation through an investigation of an RSCA.

2.2-e Two Candidates for Objects

The first problem in applying the concept of object orientation is what to associate with objects. Object-oriented items can be said to be a kind of ideology that exhorts the benefits of preparing a certain structure and then making software conform to a model that follows that structure. Accordingly, what to associate with objects is basically unconstrained. But this does not mean such correspondences can be arbitrarily decided. The benefits of object-orientation cannot be obtained by associating unsuitable things with objects. That is why design and analysis techniques for object-orientation are said to be crucial.

In the investigation of the framework for an RSCA, there were two candidates for what to associate with objects. One was **data items**, which played a crucial role in SSS, and the other was what is called **entities** in structured analysis technique.

A concrete example of a **data item** in the business field is elemental data, such as a “customer code,” “order (received) date,” or “product unit price.” Business programs in the business field normally deal with several hundred to tens of thousands of data items. We have already discussed the fact that a SSS component set is a piecemeal program group corresponding to over 1,000 data items.

Although there are a large number of data items, there is a clear correspondence between business terms in the business field and data item names. In addition, modifications of business content are normally represented using business terms or data item names. Therefore, a data item is a unit of crucial significance in the business field as indicated in the data-oriented approach as well.

The term **entity** is used in the structured analysis and structured design techniques. Generally, dictionaries define an entity as a being, body, or thing. For example, structured analysis takes up things that should be of interest (things which the customer would like to grasp, manage, and control), analyzes what sort of relationship there is between those things, models them in the form of an entity relationship model, and then draws the model as an ER diagram (ERD). An entity is what results from this process. A concrete example of an entity targeting the business field would be a “customer,” “order (received),” or “product.”

Topic 2: What Does Structured Mean?

The descriptive term “structured” is often used when writing about software programming methods. Ever since **structured programming**, which was advocated by E. W. Dijkstra, gained popularity in the 1970’s, the term “structured” has been extensively used as a catchphrase in solving problems related to software development. Note that later, the status as a catchphrase of this sort was passed on to the descriptive term “object-oriented,” which had an older origin.

To begin with, in order to understand structured programming, we’d like you to read the explanatory text according to the following musical notation that includes its essence. Note that before you read the

following paragraphs, I recommend you read through **Appendix 1** "**What Does Running a Program Mean?"** to gain some programming knowledge.

When you study structured programming, you will encounter the theorem of being able to get rid of the non-structural "go to statement" by using a number of patterns including conditional loop structures, such as while statements. Since this has been mathematically proven, it is fitting to call it a theorem. Later on, we will drive home the importance of using a carefully selected control structure (method of writing statements for controlling program flow), such as while statements or case statements rather than go to statements.

Structured programming has actually had an enormous impact. It can take credit for the standardization of program control structure in an easy to view pattern.

A **go to statement** is a visibly unpleasing statement that resembles references in a book that tell you to skip ahead to read a certain paragraph in a certain section of a certain chapter. If such statements appear everywhere, the overall book will obviously be difficult to read. A **while statement**, one of the visibly pleasing statements, instructs a program to, for example, repeat the following calculations every day until the end of the month. It is a control structure that clearly defines under what conditions to repeat and what to repeat and then instructs such repetition. Like the musical notation that instructs repeated performance that appears above, a while statement is an easy to view control structure. Since a **case statement**, another one of the visibly pleasing statements, is a control structure resembling itemization, using it will clearly make a program easier to view. In this manner, structural programming has increased the visibility of programs.

However, the control structure known as subroutines, which are easily readable statements, is effective for the most part, but it is not always easy to view in every case. Subroutines define words packed with meaning so to speak, and then correspond to the use of such words. As a result, it becomes necessary to thoroughly perceive the meaning of those words, and there will likely be resistance to overdoing this. In short, since subroutines for the sake of visibility alone enter the realm of disorganized semantics, the assignment of easy-to-understand meanings and the ability to fully use words packed with meaning become issues.

Note that this book distinguishes between visibility and readability. Readability will be discussed later.

Structured programming was effective, but of course it alone was not a solution for all problems related to software development. There was still a mountain of issues to be dealt with. Under such circumstances, the preprocess responsibility becomes crucial, and so from the mid 1970's, the general design carried out in advance of programming had to be structured. The **structured design technique** was advocated and the revision of analysis work before structure analysis (**structured analysis technique**) was said to be important. The revision of analysis and general design certainly was necessary, but it did not reveal how analysis and general design could be skillfully structured so that anyone could understand. Consequently, it is not remembered as a loud declaration of victory. The descriptive term "object oriented" emerged during such a period.

2.2.1 Associating Entities with Objects

When the investigation of an RSCA commenced, it was thought that associating **entities** with objects was closer to the concept of object-orientation that attempts to honestly model the real world. Incidentally, the significance of object-oriented entities does not reside in purpose-orientation, but rather object or entity-orientation. In simpler terms, it emphasizes the correspondence with things in the real world when designing software structure.

There is also continuity from associating entities with objects, structured analysis, and structured design techniques. In addition, it could also be said that entities are similar to normalized tables known to people who know relational database (RDB) theory. Based on these reasons, the association of entities to objects was thought to be safe.

2.2.1-f Object and Instance Variables

When associating entities with objects, attributes — a structured analysis or structured design technique term — are equivalent to instance variables (called private member variables in the OOP language C++) in object orientation. They are also equivalent to what are known as columns or items in normalized tables and even data items that play a crucial role in data-oriented approaches and SSS.

Specifically, if you considered a “product” as an object, then the data item that represents an attribute of the product object, such as “product name,” “product unit price,” or “product size” would be an instance variable (or private member variable).

A large variety of terms have suddenly sprung up. The use of synonyms is troublesome because of the different origins of terms despite similar content. It is probably good to rely on familiar words to gain an understanding when unfamiliar words appear. Each and every term differs slightly in emphasis, nuance, and level of abstraction. However, if terms have nearly the same meaning, they can be thought of as being the same. The correspondence between such terms is shown below.

Object-Oriented Terms:		**Non-Object-Oriented Terms**:
Object	<===>	Entity ≈ Table ≈ Format of a record in a file
Instance variable ≈ Attribute ≈ Private member variable	<===>	Attribute ≈ Column or field ≈ Data item or item

2.2.1-g But is This Progress?

In one sense, the correspondence of entities to objects borrows from the fruits of structured analysis and structured design techniques, and therefore, one might think that no progress has been made, but that is not entirely true.

Early structured programming had a **procedure-oriented** concept that was most concerned with structure of the procedures, but it took a major swing to a **data-oriented approach** emphasizing data analysis, organization, and naming conventions among other things within the later structured design and structured analysis techniques. This shocked those who were interested only in procedures. However increasing interest in **object-orientation** later forced the pendulum to swing back to a central position where both the procedures (methods in object-oriented terminology) and data portions had to be considered. As a result, we must have advanced one step ahead.

In object-orientation, the meaning of the basic units known as **objects** within the hierarchical **class** structure is clearly defined. In addition, objects become highly independent capsules containing data and procedures, i.e. all relevant information, in capsules (know as **encapsulation**). Encapsulated data undergoes **information hiding** so that it is not visible externally, allowing the use of objects without having to know their internal details. In short, the boundary between the internal and external tended to blur unless considerable care was taken in traditional programming methods, but with object-oriented programming (OOP), there is no longer the need to clearly define the internal and external, thereby avoiding ambiguity and making programs easy to understand. This had the effect of gathering methods (i.e. procedures) concerning encapsulated data into highly independent blocks. In the past, similar procedures concerning a single piece of data would end up scattered about here and there if you were not careful, but with object-orientation, a procedure and its data are concentrated into a single capsule.

A specific example would be taking the entity "product" as an object, and then applying information hiding to the data item "product name" or "product unit price," either of which are an attribute of the object, so that it cannot be seen. Then, by preparing methods for the product, such as "product purchasing" or "product shipping," all processing for the product will be carried out through these methods.

A mere entity contains data items that represent its attributes, but it does not contain a procedure. On the other hand, objects contain both data and procedures, both of which are encapsulated. You could say that comprehensively perceiving data and procedures was progress. This aspect of an object differs from a mere entity.

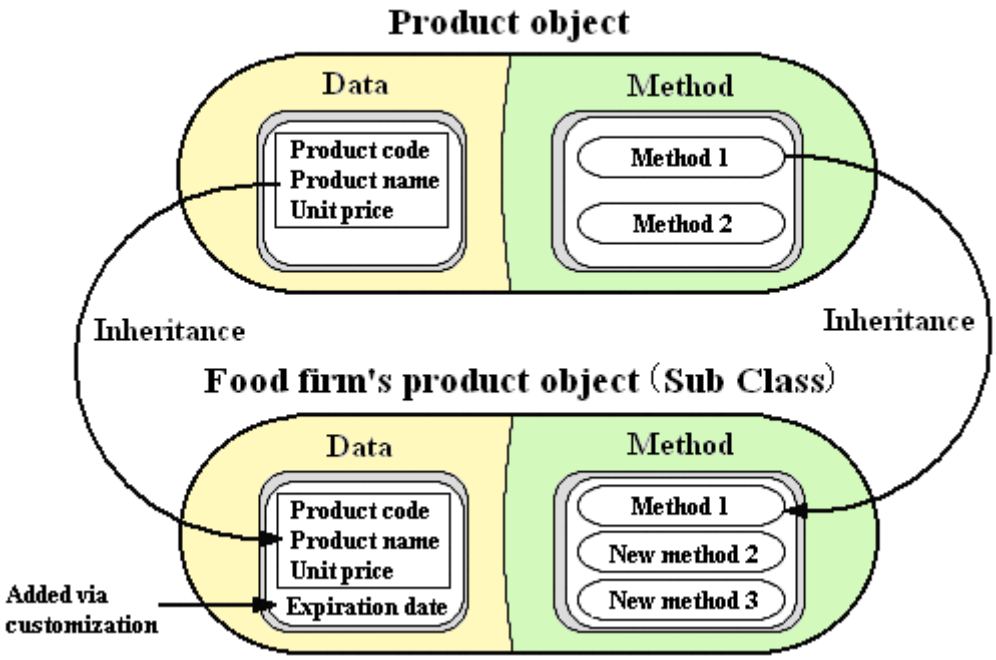
Here is some supplementary information for those who found it difficult to understand **2.2-e** "**Two Candidates for Objects**." The correspondence of data items with objects is a point of view that focuses on data items for which methods are associated. The correspondence of entities with objects is a point of view that is focused on entities for which methods are associated, and it means that data items are not exposed (i.e. they are turned into private attributes). Exposing data items weakens the effect of information hiding because it results in the ignoring of object-oriented structure.

*Those who are not very interested in object-orientation or find it boring should skip forward to **2.2.2 "Associating Data Items with Objects**." Doing so should present almost no difficulties in the understanding of later explanations.*

*The above paragraph is equivalent to a **go to statement**, which is contrary to structured programming, but such paragraphs are extremely rare in this book, so I would like to ask for the reader's understanding in this matter.*

2.2.1-h Effects of Object Orientation on a Reuse System of Componentized Applications

The change in point of view from what was once perceived as entities to objects is undoubtedly progress, but what are the resulting effects on an RSCA? During the investigation of an RSCA, the interest of the members conducting the investigation focused on this point, while keeping in mind the concept of object-orientation.



Customization work turns into a refined procedure.

Figure 2-1: Effects of Object Orientation on a Reuse System of Componentized Applications

Object-orientation is generally described as having the effect of improving reusability, increasing efficiency of large-scale development, or increasing reliability, but we cannot gauge the degree of effect from such abstract wording. Often, exaggerated explanations of productivity improvement really only mean an improvement of about 0.1 %. Such a meager effect does not give one much to be thankful for. Object-orientation for the sake of object-orientation is meaningless. A harsh point of view would be that if conventional concepts and models can produce an adequate degree of effect, object-orientation should not take the credit. Consequently, the matter of what really is the advantage of an RSCA compared to those in the past became an issue.

During the investigation of an RSCA, only one effect of object-orientation was found. After eliminating what was formally achieved by the component-based reuse system SSS and what has no effect visible to the naked eye, only one effect could be found, but it was of considerable value.

The effect was the ability to transform commonplace customization procedures into a refined specification for an RSCA by using an object-oriented structure. Let's use an example of a product entity, or better yet an object, which is one level higher, to take a specific look at what customization procedures can be transformed into refined specifications. Refer to **Figure 2-1**.

To characterize objects during design in object-orientation, each object is given one or more attributes (i.e. instance variables). This decides the scope of what sort of processing is possible for each object. Serious consideration is also given to the product object to decide what sort of attributes to give it. For example, attributes such as "product name" or "product unit price" are required in almost all processing, and so they should be built into the product object from the outset as mandatory attributes. The majority of attributes that should be given to the product object in this manner can be brought to light ahead of time. However, since it is not possible to predict all attributes that will be necessary, customization work will be required for adding attributes when you discover that they are necessary. For example, the attribute "product expiration date" might be specially required for a product carried by a food firm, while Company A, a mass retailer of electric appliances, might have customers in 50 Hz and 60 Hz power-supply regions, and therefore, specially require the attribute "power supply type for product" that defines what the default power supply type is (or whether the product is for both types or does not have to be switched). When a special attribute is found to be necessary in this manner, customization work is frequently required to add it to the product.

This sort of customization would be refined in specification when an object-oriented structure is used. Ideally, it would be possible to add/modify/delete methods (procedures of a program) when adding/modifying/deleting product attributes. This ideal can be achieved by using a truly object-oriented structure because object-orientation encapsulates attributes and methods together in one bundle.

In addition, providing a hierarchy, as described hereafter, conveniently opens a road for utilizing the object-oriented reuse mechanism known as inheritance. In short, the definition "food firm's product object" or "Company A's (mass electronic appliance retailer) product object" as a subclass (child class) of the "product object" that includes attributes, such as "product name" and "product unit price," enables the inheritance (reuse) of its attributes (instance variables) and procedures (methods).

Using this sort of object-oriented structure makes it possible to transform customization methods into refined specifications. Note that since SSS tools are poorly organized, this sort of customization work can only be carried out manually through laborious efforts. Issues requiring improvement include not only the above-mentioned problem with SSS tools, but also the ability to deliver refined specifications.

2.2.1-i Reuse System of Componentized Applications and Object-Oriented Technology

It is often asked, "What does it mean to apply object-oriented technology?"

There are those that take the harsh view that since Smalltalk is object-orientation in its pure form, object-oriented technology will only be truly applied when a program is written in Smalltalk language. There are also those that say object-oriented technology will be applied as long as some kind of OOP language is used. These are the opinions of people who focus on programming.

In contrast, people who focus on object-oriented analysis and design techniques carry out analysis and general design along the lines of the object-oriented concept, and they are of the opinion that an OOP language does not have to be used to apply object-oriented technology, as long as programming is carried out in the spirit of such analysis and design. Also, people who focus on analysis and general design claim that those who focus on programming are wrong. They say that the use of the OOP language C++ indicates

a program that was developed without using any object-oriented technology. As you can see, there is not always agreement on what constitutes the application of object-oriented technology.

Analysis and general design were carried out along the lines of the concept of object-orientation in the investigation of a framework for an RSCA, but the product version (RRR family) did not use any OOP languages. (Note that the latest version does use OOP languages, such as Java, VB.NET, and C#.NET.) Consequently, it should be said that whether object-oriented technology has been applied to an RSCA, or the RRR family, changes depending on the previously mentioned standpoints. Accordingly, this book says, "The concept of object-orientation was a strong focus in the examination of a framework for an RSCA." Note that this wording is used because this book attempts to accurately and honestly evaluate object-orientation.

Incidentally, the author has nothing against adding the epithet "object-oriented" in an advertisement for products, such as the RRR family. This is because the use of such descriptive terms in advertisements has already become commonplace.

It seems our discussion is returning to square one. The reason no OOP languages and object development support tools were used in the development of the RRR family was that no appropriate ones were discovered. If it were merely okay to transform customization work for adding/modifying/deleting attributes into refined specifications, then an OOP language could have been used. However, doing so would lead to rough-going in a variety of areas. It would be just like being alienated like a foreigner in an object-oriented system. Although this is the way it is supposed to be in the business field, it is unacceptable in the world of object-oriented systems.

For example, using an object-oriented inheritance mechanism to carry out customization work for adding/changing/deleting attributes makes changes to their related files and databases, but unfortunately, this does not result in a design that enables access with good performance. Since customization is impractical if it leads to a decline in access performance, it is crucial for this sort of customization to not cause any performance problems. However, no object-oriented development support tools that satisfied this requirement were found.

I am not asserting here that object-oriented systems neglect performance. If you take a general look at hardware performance, you will find that compared to the progress made in **CPU** speed, memory capacity, and disk space, progress in disk access speed has been vastly lower, and therefore, this cannot avoid being a performance bottleneck. This has resulted in an introduction of a variety of software ingenuities to deal with this bottleneck. For example, to improve the performance of object-oriented databases, disk access is decreased by loading all related data into memory ahead of time, processing at high speed without writing to disk each time data is modified, and then by writing to disk all at once, such as when a day's worth of processing is finished. A variety of such performance considerations are being made.

However, in the business field, the above-mentioned method cannot be adopted because data must be written to non-volatile memory, such as disks, each time a form is processed. When investigating methods to improve the performance of object-oriented systems, it would be desirable to focus on their usage in the business field and then adopt a method that was befitting to it, but that has yet to be done.

Since appropriate object-oriented development support tools were not found, an inheritance mechanism that took into consideration access performance to files and databases had to be developed in-house. Note

that a completely object-oriented implementation was not developed, but rather the characteristics of the business field were taken into consideration to develop a special mechanism that reduced the places where object-orientation was thought to be unnecessary. There are many things we would like to try, but there is no point doing so when cost and effect is considered.

2.2.1-j Extended Features Necessary in the Business Field

During the development of the RRR family, features unrelated to the business field were reduced, while on the other hand, those required for the business field were extended. One of the main extended features was related to the customization accompanying the addition/modification/deletion of attributes as previously discussed. In short, improvements were made to increase the inheritance rate with inheriting methods.

Understanding this mechanism requires a certain degree of knowledge about object-orientation, so let's start with a description of what it is.

Differential programming is one term for object-orientation. This type of programming means, as long as you write subclass-specific procedures (methods) as differences, parent class (super class) procedures will be inherited (can be reused), with no further work necessary. In short, differential programming is a crucial mechanism for reuse in object-orientation.

In orthodox object-orientation, a procedure in a class is the unit of inheritance, and hence, the unit of reuse. However, during the investigation of an RSCA, problems were found in the commonplace customization discussed earlier, at least in the business field. In short, if the unit of reuse is too large, the rate of reuse will not increase. Since the unit of reuse is the unit of differential programming, we must reuse the whole unit as is. Otherwise, it is necessary to write the entire program (all reuse blocks) rather than only the actual differences.

Generally, making the unit of reuse smaller decreases the number of program lines that must be written additionally and increases the rate of reuse. But that does not mean it is okay to merely make the unit of reuse smaller. Making the unit of reuse too small causes the basic unit to be nearly unidentifiable and makes it difficult to understand the program overall. Consequently, it really makes sense that the unit of reuse in orthodox object-orientation is a procedure in a class.

In the evaluation of whether object-orientation is effective, there are two crucial keys: can it model the real world in an easy-to-understand manner, and do programs (procedures) become easy to reuse? For the former, modeling, the unit of inheritance should be procedures in a class associating with entity, but for the latter, reuse, this is not suitable, at least in the business field. This is because as far as commonplace customization is concerned in the business field, the unit of differential programming in most cases is something associated with data items, which are even more detailed than entities.

For the **RRR family**, accordingly, the conclusion was reached that it would be best to make what would become procedures associated with data items, through further partitioning the procedures, in a class associating with entity, in accordance with data items, and then make them the unit of differential programming (i.e. the unit of reuse). This also dramatically increases the rate of reuse by inheritance.

The following two sections offer a rather detailed discussion for those who know a lot about object-orientation, or those who have developed an interest in it. If you are not interested, please skip ahead to ***2.2.2*** *"****Associating Data Items with Objects****."*

2.2.1-k In-Depth Look at Extended Features Considered Necessary

Customization that is commonplace in the business field is concerned with data items that are even smaller than entities. Also, those who are able to add or replace procedures corresponding to data items are able to increase the rate of reuse in the customization of data items.

For example, a method (procedure associated with an object's class) that displays a product's attributes will become something different if you carry out customization that adds new attributes to or modifies/deletes existing attributes of a product object.

As a specific example, let's look at customization wherein the attribute "product repair service center" is deleted and the attribute "product expiration date" is added for "food firm's product object" of the subclass "product object." Refer to **Figure 2-2**. When carrying out this customization, it is necessary to stop displaying the product repair reception desk and start displaying the product expiration date in the method that displays product attributes. Consequently, the method for "product object" cannot be inherited as is. The entire method that displays product attributes must be written as a new method (procedure) for "food firm's product object."

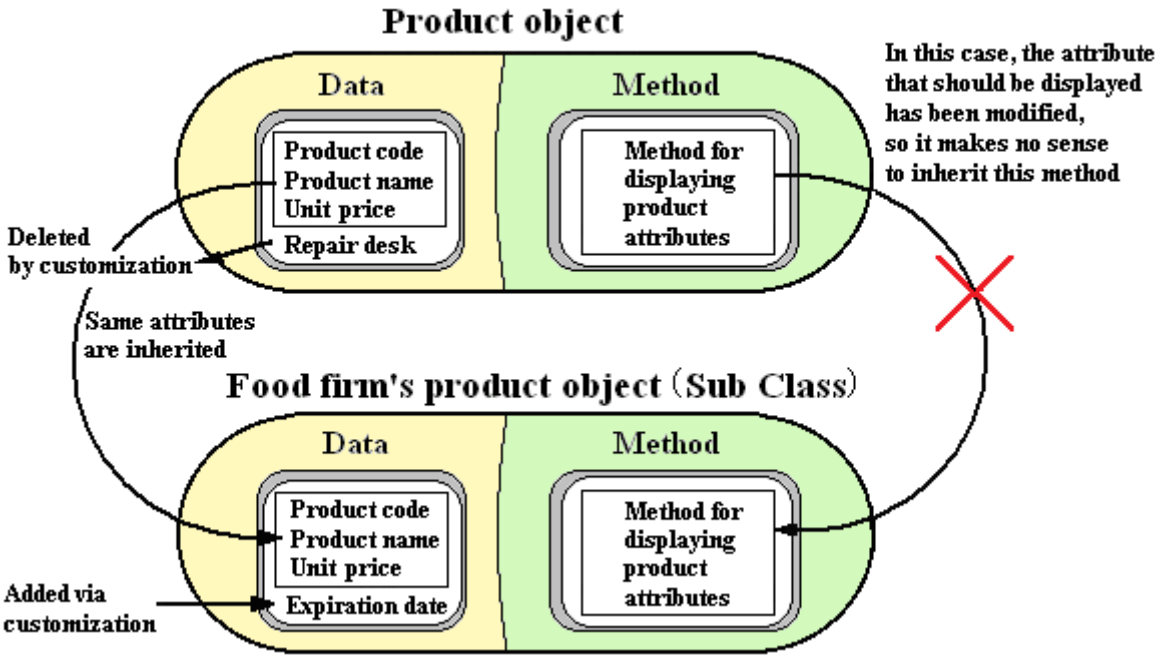


Figure 2-2: Example of Customization Adding/Removing Product Attributes

Since there is only a very small difference, it is not efficient to require that the rather large procedure including the small different portion be entirely rewritten. We would like to increase the rate of inheritance (rate of reuse). The idea adopted in response should be able to raise the rate of inheritance because small, individual blocks are not affected to any large degree, by customization work, if the large method that displays product attributes is partitioned into small methods corresponding to data items.

Since actually associating methods to data items and then partitioning them results in small methods, most of them can also be used for "food firm's product object," therefore, enabling inheritance. Consequently, the only new thing that has to be written is the small method that displays the product expiration date, which did not previously exist. This also dramatically improves the rate of inheritance.

The **RRR family extension** for the business field allows the handling of procedures which are partitioned corresponding to data items, i.e. “small methods,” by using visual and simple indication.

If you were to change your perspective in the investigation of the problem of increasing the rate of inheritance, you would find that it is essentially related to what you should make a method. When you figure out what you should make a method, you end up returning to the first problem of what to make an object. Let’s take a close look at this.

In object-orientation, generally a procedure associated with an object’s class is said to be a method. Accordingly, if you associated a data item with an object, then a method would be a procedure corresponding to a specific data item from the start. This means there is no need for partitioning a large method in accordance with data items.

If we were to further develop this discussion, you would probably come to understand how it would return to the first problem of what to make an object. We will investigate the correspondence of data items with objects in **2.2.2** “**Associating Data Items with Objects**,” while in this section we will simply consider the correspondence of **entities** with objects. If we did not do this, the discussion would end up going in circles, and we would like to avoid that. Despite our saying this, the question of how the following two actions differ might come up, even given the premise of associating entities with objects.

(D) Partitioning a large method in accordance with data items.

(S) Perceiving “small methods” corresponding to data items, for example, displaying product names or product unit prices, as methods from the start.

As discussed above, it is possible to increase the rate of inheritance through carrying out **(D)**. If that is the case, the question is whether it is really the same thing or not, to make “small methods” corresponding to data items as methods (unit of inheritance) from the start, as described in **(S)**. To answer this question, the people who put methods into context must clearly define what stage of development they are in.

If they are in the analysis and general design stage, improving the ability to see the whole picture is crucial. Doing **(S)** would be completely out of the question because in the business field it would result in an enormous number of methods, thereby worsening the ability to see the analysis and general design. Since methods that display or change individual data items will emerge, there is no choice but to grapple with methods several times the total number of data items. The number of data items handled by business programs in the business field ranges from several hundred to several tens of thousands. This results in having to remain aware of this magnification while conducting analysis and general design.

Since this is very troublesome, we turn procedures associated with entities into methods (unit of inheritance) when conducting analysis and general design. However, since entities are normally a bigger unit than data items, methods too end up being bigger blocks, and so the answer to the previous questions is that the rate of inheritance during customization work cannot be raised without partitioning these big blocks.

2.2.1-l A Number of Mismatches with Business Fields

In trying to figure out what should be made a method, the problem arises, whether models that follow an object-oriented structure are a perfect fit for the business field. At present, object-orientation refers to a procedure in a class as a method, and this is used as the unit of inheritance. As a result, the fundamental problem of whether this is appropriate arises.

Although we are not trying to defend object-orientation here, it is appropriate that the method, which is the unit of inheritance, belongs to an object's class. Since object-orientation emphasizes correspondence with things in the real world, methods relating to such things are made the unit of inheritance. If some pieces smaller than an object were to be arbitrarily made up and then methods were associated with it, the meaning of an object would probably become ambiguous.

However, the situation in the business field differs slightly. In some cases, there are those who want to ignore data items that are smaller than entities (to which objects are associated), but in other cases that has a clearly defined meaning. In short, improving the ability to see the whole picture is crucial, and therefore, the correspondence of entities to objects is easy to understand. However, the rate of method inheritance in customization work will only increase if we associate methods with data items that are smaller than entities. This means that in the customization phase, we require not only methods associated with entities, but also methods associated with data items.

Topic 3: Talking about Instances

The following paragraphs will introduce two topics related to the concept of object-oriented instances.

The first is a discussion about the secondary benefit of object-orientation that was discovered while thinking about what an **instance** of a product object was.

For example, if you were to consider a business program for the picture gallery industry, instances of a product object would normally indicate individual pictures. If prints were also handled, there would be only a slight difference from paintings. With prints, a number of copies are made from a single work, and therefore, there are two possible ways to handle them as follows:

- Make each copy of the work an instance. (The focus is on each copy.)
- Make the title of work an instance. (The focus is on each work.)

If you wanted to handle print copies that are numbered and signed as individual products, you would probably make each copy of the work a separate instance. If you perceived the prints as a single mass without distinguishing between individual copies and you were only interested in how many you had in stock, you would probably make the product type an instance. In short, you would regard copies of a single work all together as a single product.

It is usual to decide in this manner what should be made an instance based on how much of a distinction you want to make. Consequently, a business program for a stationary wholesaler would normally associate a product classification or type (or even a group of special products), such as "pencil B123 manufactured by Company A" as an instance. Such a program would almost never make each pencil a separate instance of the product object.

Generally, when you leave an instance ambiguous during business program development by a large number of people, the results of development would be confused and misunderstood. If developers knew object-oriented terminology, they could communicate using special terms like instance, thereby helping to establish a mutual understanding between them. If we were to evaluate this effect in a slightly exaggerated manner, we would say that establishing an object-oriented structure and clearly defining the meaning of its terminology is a secondary benefit.

Another discussion we could have when talking about instances is about the importance of **tuning object-oriented features** for the business field.

When you use an OOP language, unique identifiers are applied for identifying each instance in a perfectly natural manner. For example, an identifier would be applied to each instance of a product object to distinguish them from other product instances. This may seem quite convenient, but that is not so. Since code systems, such as product codes, have been traditionally used as unique identifiers in the business field, we end up with two of the same thing.

The continued use of product codes as in the past would lead to the existence of two types of identifiers, and such duplication would be a waste. In fact, not only would it be a waste, it would also inevitably lead to confusion because there would be two things charged with the same role. This makes one want to abandon product codes in favor of using only instance identifiers.

In most cases however, instance identifiers are arbitrarily applied by an object-oriented system, and therefore, cannot be systematized ahead of time like product codes can. Also, identifiers are normally not persistent, which means different identifiers might be applied each time you start an application program. For these two reasons, instance identifiers are difficult to use in the business field.

An in-depth look into the inner workings of an object-oriented system reveals there are many cases in which, for some reason, instance identifiers are made out of the memory addresses in their internal tables. This is a problem because it ignores such things as usage in the business field. If we were to recommend object-orientation in the business field as well, then we should by all means make instance identifiers convenient to use just like conventional product codes. Alternately, we should allow most processing to be done using product codes only and not identifiers, even if they exist.

Note that making identifiers persistent in an object-oriented database is a good thing, but identifiers cannot be systematized ahead of time like product codes can. In addition, it appears there is a problem due to the lack of consideration of usage in the business field. For example, there are many problems with consistency in exclusion control and transaction control, which are necessary in the business field.

2.2.2 Associating Data Items with Objects

The previous section associated entities, which have traditionally been the focus of the business field, with objects, and discussed the investigation and analysis of an RSCA while keeping in mind the concept of object-orientation. This section reports the investigative results of associating data items rather than entities to objects.

Associating data items to objects is convenient because it fits in with the concept of assigning a crucial role to data items, which is a special feature of SSS. However, since this differs from association by the usual object-oriented analysis, we were left wondering what it is. We were worried this was too realistic and might have a weak theoretical basis.

However, in general, we learned that in visual development support tools, there is something close to data items corresponding with objects. Therefore, we were relieved that the correspondence of objects to data items was not unprecedented. In addition, since we wanted to try to use some sort of visual development tools in the development of the RRR family, we gathered the support of the members conducting the investigation for associating objects with data items based on the fact that good conformance with visual development support tools was convenient.

2.2.2-m Object Orientation and GUI Operation

A graphical user interface (**GUI**) is a method of visual operation using **GUI controls** (also known as widgets), such as buttons, menu items, list boxes, or text boxes, laid out in a window. The GUI was born at Xerox's Palo Alto Research Center, further developed by Apple Computer in their Macintosh computers, and finally enjoyed commercial success with the masses in Microsoft Corporation's Windows operating system. GUI software is successful because it perceives GUI controls as objects, and it is like a child of object-orientation born in answer to our prayers. This section will introduce a GUI program as a successful example of object-orientation.

GUI operation is said to be object-oriented. This is said due to the fact that GUI operation and object-oriented structure have a perfect correspondence with each other as described below.

GUI operation is represented in the form "**perform x operation on y**," such as "perform click operation on the menu item," "perform double-click operation on the icon," or "perform click operation on the button." If we associate these GUI controls (menu items, icons, and buttons) with objects, then the operation of "perform x operation on y" becomes "**perform x operation on the object**." Furthermore, if we associate this "perform x operation" with a method that has an object, then we can represent all of the above examples as "**perform the operation prescribed by the method on the object**." Refer to **Note 3**.

Note 3: Generally, when this sort of easy-to-understand correspondence exists, i.e. program fragments and things in the real world correspond exactly; the program fragments corresponding to those things become easy-to-identify blocks and are easy to make into what can be called components.

For example, since the various components in the SSS component set can correspond with data items, we can partition a program into easy-to-understand units (data item components). In addition, since GUI operation programs can correspond with GUI operation in the form "**perform the operation prescribed by the method on the object**," we can partition programs into easy-to-understand units (components).

GUI operation corresponds perfectly to object-oriented structure as just described, and that is why the application of object-oriented technology is becoming a fixture in GUIs.

2.2.2-n GUI Operation Base and Processing Programs

Generally, programs related to GUI operation use object-oriented technology, associate each GUI control with an object, and associate the operation of such controls with a method. When an operator performs an operation on a GUI control, the result is a mechanism in which the method corresponding to the operation on the GUI runs.

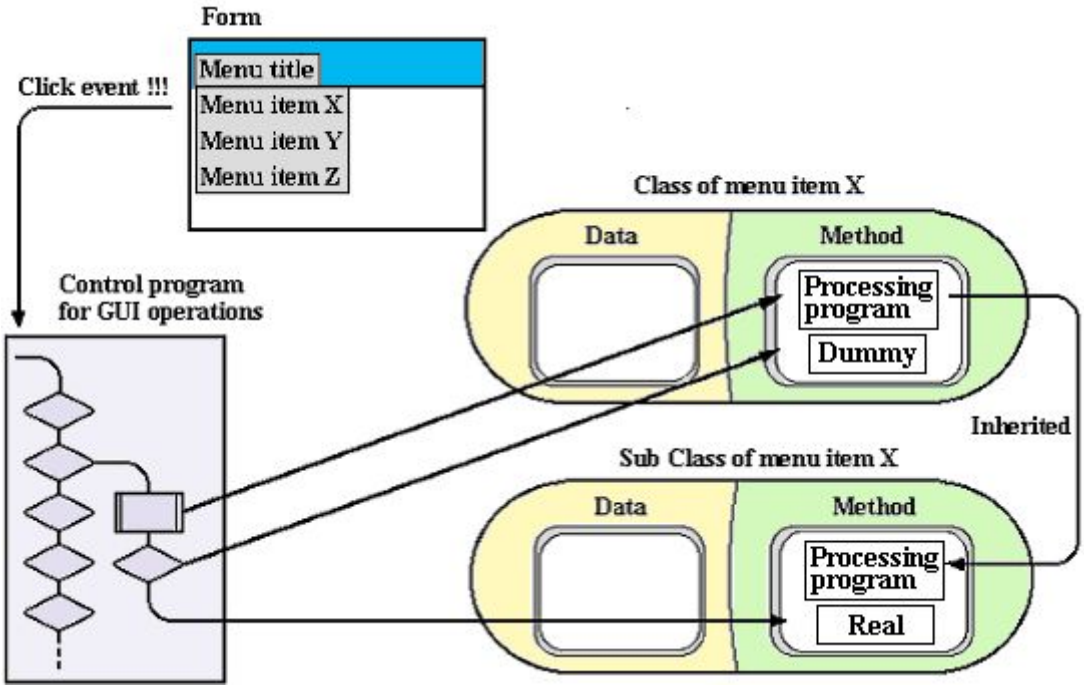


Figure 2-3: GUI Operation Base and Processing Programs for GUI Operation

If you take a look at the development of programs related to GUI operation, you will find that you can classify them into those that form correspondences with objects (GUI controls) and those that do not (those that take the roll of passing on control for example). We know that we can partition the former, i.e. programs that form correspondences with GUI controls, into individual methods for an object.

As shown in **Figure 2-3**, this book calls the former **processing programs for GUI operation**, and the latter, which does not correspond with objects, a **GUI operation base**. Generally, application programs that run on an operating system (OS) are called processing programs, but let's try to classify GUI operation programs into two categories similar to what was described above.

Classification in this manner results in **processing programs for GUI operation** that correspond exactly with the methods of the corresponding object. In addition, a **GUI operation base** can be referred to as something that passes control (calls) to a method (processing programs for GUI operation) that corresponds to the operation performed by an operator on a GUI control. Moreover, you can also say that a GUI operation base is a program that implements a mechanism upon which methods run.

2.2.2-o Reuse of GUI Operation Base and Processing Programs

Let's say that when developing a program to achieve a certain objective, we decided to adopt a GUI because we were concerned with operation characteristics. However, developing this sort of program from scratch is a big job due to the fact that we have to develop the program itself (a business processing-related program), and on top of that, develop both a GUI operation base and processing programs. To make this development work easier, it would be convenient to reuse a GUI operation base and processing programs that have already been developed.

Generally speaking, the reuse of programs involves decipherment, which makes reuse very tedious work. However, GUI operation bases and processing programs that use object-oriented technology can be easily reused.

A specific example would be relying on a GUI operation base and click method (processing program for GUI operation that performs standard processes) for most of the processing that has to be performed when the user clicks menu item x. The common processing (required when the user clicks menu item x, or when he/she clicks menu item y) is performed by the same GUI operation base and click method already developed. Consequently, we only need to create a program related to the special processing. In short, we only need to create the business-processing program.

Let's take an in-depth look at this. After establishing menu item x as menu item subclass, we would write the business processing portion of the program as a click method for the business processing of menu item x. By doing so, the method for menu item x would be inherited in principle as the method for other menu items, and a program written for the difference would be used only for the click method for business processing. In short, we would reap the benefits of having only to write a program for the difference (differential programming).

Another way of looking at this is to say that the benefits of object-oriented technology make it easy to perform program customization on processing programs for GUI operations.

This has already been stated earlier, but the use of object-oriented technology makes reuse easier. Developing programs from scratch has some tedious aspects including the need to provide a design of what to associate with objects, but reuse certainly makes the work easier.

2.2.2-p Visual Development Support Tool

A number of types of visual development support tools are currently available. They provide mechanisms for the above-mentioned differential programming and make it extremely easy to develop programs that provide GUI operability.

Visual development support tools often use the terms **event procedure** or **event handler** to refer to programs that are written as differences, such as a click method for business processing. An event, for example, is something that happens like "clicking," and an event procedure is the program that runs when such an event occurs. In addition, visual development support tools have a convenient mechanism for automatically generating subclasses simply by writing an event procedure. Note that an in-depth discussion of event procedures is provided in **3.2.2** "**Fourth-Generation Languages (4GLs)**."

As you can see, visual development support tools are in almost complete accord with object-oriented structure, but they have fewer features than the object-oriented structure of Smalltalk, which is said to be orthodox object-orientation. They also have portions that have been extended. These portions should be viewed as a favorable adaptation for visual development support tools.

2.2.2-q GUI Objects Corresponding with Data Items

A GUI can be viewed as a stage director (operator) assigning a performance to actors (GUI controls) on a stage (window). In that sort of world, object-oriented structure fits in perfectly, and object-oriented technology has achieved great success.

During the investigation of an RSCA, we considered what we should do to capitalize on this great success. In other words, since sixty to seventy percent of components in the RRR family run when an operator interacts with forms, we felt we were in a similar situation, and the incorporation of GUI operability into the RRR family was essential. To do this, we associated the forms that the RRR family handles with windows, and then we thought it would be good to establish a design that associates input fields within forms with text boxes (a type of GUI control).

Upon actually trying to create a prototype using a certain visual development support tool, we found that although functional tuning was required, there were basically no problems designing in that manner. In

short, this was the greatest strength of SSS. We discovered a way for associating data item components with the event procedures of a visual development support tool, and this discovery assured us that we could inherit all of the good aspects of SSS. More information about this is provided in **3.2.3-s "Second Improvement of Partitioning Guidelines for Compartmentalization of Components"** within **3.2.3 "From SSS to RRR Family."**

The creation of such a design agrees with the policy of conforming to current popular concepts and tools as much as possible without losing the strength of SSS, which we adopted in the investigation of an RSCA. It also allows the relatively easy development of the RRR family because off-the-shelf visual development support tools could be applied to sixty to seventy percent of RRR-family components.

However, upon seeing the results of such a design, we realized that data items in the RRR family were regarded as objects. In short, data items in the RRR family basically corresponded to data items in business program forms, and therefore, visual development support tools considered them as objects (GUI controls).

We previously thought that regarding entities as objects was good, but over the course of our investigation that included visual development support tools, data items ended up being objects.

2.3 How Has Object-Orientation Been Perceived?

This section will report how object-orientation has been perceived through the investigation of an RSCA. It will also state our conclusion as to whether we should associate either entities or data items, which we have investigated thus far, with objects.

2.3-r Object-Oriented Structure

We have already provided an overview of object-oriented structure in "**But is This Progress?**" under **2.2.1** "**Associating Entities with Objects**." The following paragraphs provide a slightly more in-depth description of object-oriented structure, in preparation for an evaluation of object-orientation. Note that readers who are not interested in object-orientation may skip this part.

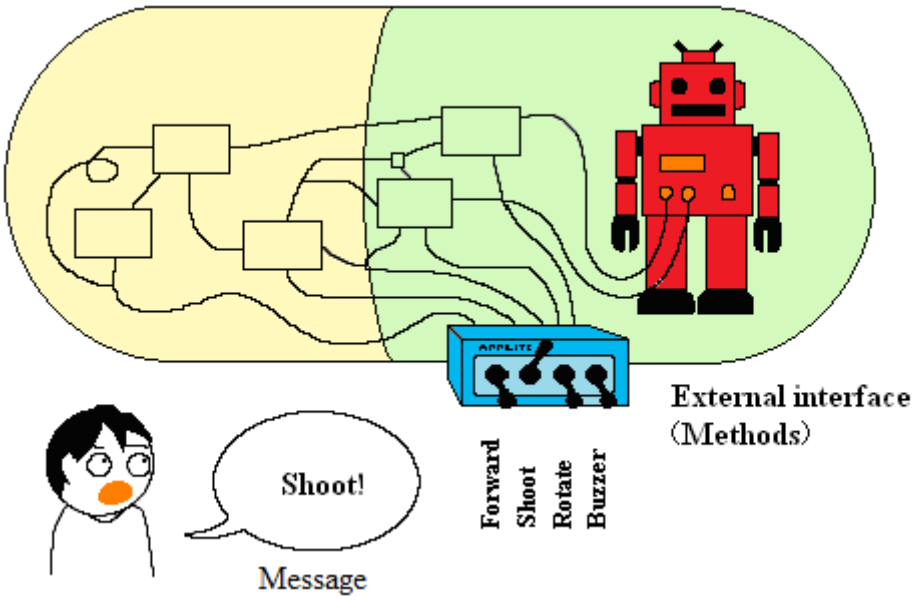
Object-oriented structure is frequently described using the example of a system that controls a robot's actions using levers on a remote control box, much like the control of a marionette. This seems to be an easy-to-understand description. Therefore, I would like you to think of a system that controls a robot. For example, you could think of the robot contests held between technical colleges and universities (events in which robots compete in ball games and so on) that are currently so popular.

It is a crucial point in this sort of system to design the robot's behavior depending on its purpose. In this case, we should start by considering the intended behavior. In object-oriented terminology, the lever operation required to produce behaviors is known as a method, and thus, we could also say "start by bringing methods to light." In addition, each method can correspond with a behavioral element we want the robot to produce, such as shoot the ball or move forward. In short, the operation of levers on a remote control box enables the production of such basic behavior.

Incidentally, in the business field, it is usual to associate fundamental processing, such as the registration of a product or the displaying of attributes of a product, as a method for product objects of a sales management system.

To use a word familiar in the programming world, a **method** is a procedure that performs certain processing (or produces a certain behavior) for an object. In simpler terms, you could think of it as a subroutine that performs some sort of processing for an object. Strictly speaking, a method is something that defines an object's **behavior**.

The strength of object-orientation lies in its encapsulation of data and methods so that they can be comprehensively perceived. This can also be seen as a means for preventing data and procedures from being haphazardly scattered about a program. In addition, this imposes constraints so that data referencing and modification are only possible via methods. This is none other than an established mechanism for information hiding that resulted in developments even in non-object-oriented areas. With the inclusion of these matters, object-oriented structure can be discussed in the following manner.



Information hiding is applied so that you cannot see the complex structure within the object

Figure 2-4: Object-Oriented Structure

When you clearly define and transmit an object name, method name, and parameters as a message (an object-oriented term that means a form of correspondence like a letter), they are conveyed to the target object, resulting in the implementation of the stipulated operation by the specified method. For example, transmitting a message for displaying a product's attributes that clearly defines the object name "product" and method name "display product attribute" will result in the implementation of the stipulated display processing by the specified method.

There is no way to manipulate an object other than by using a method. As a result, it is not possible to directly manipulate or reference an instance variable within an object. When designing a robot system, you consider how you want to manipulate objects, and the operations required to make this work as you intended must all be built-in as methods ahead of time. Consequently, when you view an object from the outside, it will look like the previously mentioned remote control box because you will only see the methods on the outside, not the complex content within the object. The end result is skillful information hiding. Since parameters for each method are predetermined, the object's external interface can be said to be only its methods and their parameters. In short, the scope of an object's behavior is prescribed by the methods and parameters you provide.

You could probably understand this better if you were to consider the following example of a robot. The remote control box is for controlling the robot features equivalent to methods, and whenever the operator changes lever settings, messages are conveyed to the robot, causing it to produce a certain behavior. The only thing that can be done to the robot externally is to change the settings of the control box levers, and it is therefore clear what the external interface is. The direct manipulation and observation of the robot's internal workings are prohibited. Therefore, the operations that can be performed on the robot are clearly determined by the types of levers provided on the control box.

What we have discussed here is the main structure for object-orientation. Refer to **Figure 2-4**. Note that there is also the reuse mechanism known as inheritance, as previously mentioned.

Object-oriented structure can also be represented using the following extremely familiar terms in the field of conventional software. However, the following representation eliminates special items called actors or active objects, which operate asynchronously, and targets only normal sequential processing.

An object is comprised of common subroutines and structures that group together its internal variables (instance variables). The common subroutines can be called externally, but the direct referencing or manipulation of the internal variables is never permitted. In other words, information hiding is applied so that only common subroutines are visible from the outside. The application of such a mechanism clearly determines the object's external interface as being the common subroutine and its parameters. In addition, object behavior is clearly determined by the type of common subroutine that is provided.

There are some who will criticize such a representation as desecrating the abstract definition of object-orientation, which is rich in meaning. When you use familiar terms, there is the risk of being influenced by their tone and misinterpreting them in that narrow sense. What they are trying to say is that both making the correspondence between each term and things in the real world profitable without being affected by stereotypes, and leaving some room for abstractness to allow a broad interpretation are crucial.

Although there are some who say this, there are others who defend representation in familiar terms saying that if you were to conform to the spirit of object-orientation, there would be no need to be particular about the form of implementation. For example, if they do not fixate on the form of implementation by saying a common subroutine might be a function, an inline-expanded macro instruction, message passing, or a message queue and its process (task), then they are acknowledging that even expressions using familiar terms increase understanding. Conversely, they criticize it as inadequate wording that the characteristics of object-orientation are in message passing. It is true that message passing has an advanced mechanism known as dynamic binding to people in the know, but the strength of object-orientation is not epitomized here.

2.3-s Evaluating Object Orientation

We have ended up presenting a rather in-depth discussion, but it seems that the more in-depth we go, the further we move away from the essence of object-orientation. Plenty of in-depth books have already been written. However, even if they go more in-depth than this book, it does not mean that doing so will clearly reveal the effects of object-orientation. Since in-depth descriptions frequently exaggerate small effects, they might actually make it difficult to see the true merit of object-orientation.

If you take a second look at the essence of object-orientation, we think you will find that it lies in the emphasis of the correspondence between things in the real world and programs. Assuming this, we cannot help feeling that the more in-depth the discussion, the more limited the actual system that is a perfect match

and the fewer chances there will be for reaping the benefits of object-orientation. We will discuss the above in **Chapter 6** "**Evolution of Life and Component-Based Reuse**."

It has been mathematically proven that in structured programming, the control structure of all programs can be written simply by combining recommended patterns, such as while statements. In addition, many people are experiencing the fact that standardization of this type of control structure results in easy-to-view programs.

This is not so with object-orientation. First of all, it has not been mathematically proven that the real world can be entirely represented by models conforming to object-oriented structure. Secondly, although there have certainly been reports of successful examples, such as the application of object-orientation to make the correspondence between programs and things in the real world easy to understand, this has yet to be experienced by large numbers of people, and there seems to be no guarantee of success in all cases. Even after acknowledging that proving whether this becomes easy to understand would be difficult because it involves subjective elements, we still have to report a large number of convincing examples of both successes, as well as failures, at minimum. Since object-orientation has some abstract and difficult-to-understand aspects, a clear understanding of how it is effective in specific cases (or all manner of cases) will probably only come after we have many clear-cut examples.

2.3-t Entity or Data Item: Conclusion

Amid the investigation of an RSCA (one example of evaluating object-orientation), there seems to be sufficient reason only for the associating of either entities or data items to objects.

When you associate entities with objects, information hiding is applied to data items, resulting in a tidy and improved view, but when performing customization, you must keep data items in mind, resulting in the loss of most of the effects of information hiding. On the other hand, when you associate entities with objects, customization-conscious implementations are possible and you can perform information hiding per data item, but the view when performing analysis and general design worsens and information hiding per entity becomes nearly impossible. Refer to **Note 4**. Consequently, when performing analysis and general design, associating entities with objects seems to be the better idea, and if you get to the software implementation stage, then associating data items with objects seems to be the better idea.

Note 4: There are some OOP languages that support an **object hierarchy** that allows objects within objects. Using object hierarchy enables two-way correspondence that allows objects to be associated with data items within objects associated with entities. However, even if two-way correspondences are made, usage in the business field normally requires the exposure of objects (data items), which means to make the objects public, thereby making information hiding for each entity almost impossible. In other words, this is almost no different than the correspondence of data items with objects. Therefore, in this book, two-way correspondence is considered the same as the correspondence of data items with objects.

Object-orientation has the concept of class hierarchy, and you could probably rank "product unit price" (a data item) as a subclass of the class "product" (an entity). However, doing this breaks the easy-to-understand relationship that says "product unit price" is an attribute of "product."

Thinking this way ended up making us worried that we had to choose between entities or data items.

Our ultimate conclusion was there is no need to worry because both have meaning. This is because there are many viewpoints from which business programs can be perceived, and when you look from a certain viewpoint, you can make a suitable model from it, but the resulting model will not necessarily be the best one when seen from other viewpoints.

For example, let's consider whether a virus should be perceived as an organism or a crystal. Depending on how we perceive it, the concept system (model) for gaining a deeper understanding will differ. From the viewpoint that it is an organism, we could use the model of evolution to gain a deeper understanding, and from the viewpoint that it is a molecule; we could use the model of crystals to gain a deeper understanding. Consequently, you could say that both of the perceptions are fine.

Similarly, we thought it would be good if there were suitable models for two situations. The first situation is when you are able to see from the standpoint of improving the view when performing **analysis** and **general design** for that actual state of business that will form a background for a business system in the business field in order to clarify requested specifications. The second situation is when you are able to see from the viewpoint of **implementation** so that business programs will have an easy-to-reuse format based on those request specifications. In addition, we arrived at the idea that associating entities with objects is suitable based on the former standpoint of performing analysis and general design with an improved view, and that associating data items with objects is suitable based on the latter viewpoint of implementation so that business programs are easy to reuse.

Note that when you associate data items with objects, it is thought there is no need for features that supposedly must be extended in the RRR family, i.e. features for further partitioning procedures in a class in accordance with data items. Actually, however, a feature for binding data items and entities is required instead of these features. In short, regardless of whether the focus changes to entities or data items, the need for some type of extension related to these features in the RRR family does not change. The necessary features are still required regardless of whether the point of view changes.

2.3-u Impressions of Object-Orientation Concept

Since there are many terms unique to object-orientation, we investigated an RSCA while keeping in mind those that are object-oriented after we came to correctly understand them. If this investigation had targeted another field besides business, or more precisely, if it had targeted a field in which object-orientation could easily play an active role, then we probably would have ended up with a different impression about object-orientation. However, it did target the business field. In any event, there is no disputing the fact that we got some sense of object-orientation through this investigation. In closing this chapter, we will summarize what we sensed through this investigation.

First, when talking about object-orientation itself, we think that using models that conform to object-oriented structure is generally effective. For example, the field related to GUIs is an archetype that fits in perfectly with object-oriented structure and thus achieves an effect. However, this structure alone cannot deal with the circumstances in individual fields, so might we also require creative actions for adjusting it to them? Regarding the business field in particular, there were areas ill suited to object-oriented structure, and we therefore felt that there needed to be extensions focused on usage in that field.

Next, when talking about object-oriented development support tools, such as OOP languages, it appears as though convenient features and performance tuning have been inadequate for the business field. Object-oriented technology has probably fallen into the vicious cycle of not being employed in the business

field because it has not become easy to use, and not becoming easy to use because it is not being employed. If that was not the case, object-oriented technology would be used much more in the business field based on the over 30-year history of object-orientation.

Another way to look at this is the functional tuning of object-oriented development support tools for the business field probably ends up hindering generic object-oriented structure. In short, reducing object-oriented features as was done while developing the RRR family dilutes the strength of object-orientation. We also feel that it converges toward something close to development support tools specialized for the business field, such as fourth-generation programming languages (4GL).

From the standpoint of tool vendors, selling what will be object-oriented development support tools for the business field will clearly result in competition with pre-existing development support tools specialized for the business field. However, many object-oriented development support tool vendors have appeared to avoid this by targeting other markets.

In conclusion, when you talk about the epithet "object-orientation," you need to be careful because sometimes people implement what can be implemented without using any object-oriented technology whatsoever and then rant on as if it was a great achievement of object-orientation. Using an actual development project as an example, we felt the need to debate with vendors about how well certain things improve compared to cases where object-oriented technology is not used. For example, it would be good to get to the bottom of the matter by asking questions, such as how class libraries differ from usual libraries, or how object-oriented databases differ from multimedia databases. In addition, striving to grasp the reality of this without being influenced by words is more important than anything. Alternately, ignoring the epithet "object-orientation" would be another solution.

We have thus far ended up dedicating considerable space to object-orientation. To date, there have been a variety of declarations stating," the effectiveness of object-orientation has been belatedly verified even in the business field." However, we think little has been written about concrete trials and evaluations from the user's end or from those who have gotten their feet wet and forced their success. That is why we thought it worthwhile to write honestly about what we perceived in the process of investigating an RSCA, and this caused us to go into considerable depth.

Rumors, sweet talk, and appearances alone do not deepen understanding. Pointing out a concrete example and then carrying out a discussion based on it is crucial. Accordingly, our intention was to write honestly from various perspectives on what exactly we did and felt. If you think that there are any misunderstandings or that we overlooked something, please send an e-mail message to bring the matter to our attention.

CHAPTER 3 Software Development Support Tools

This chapter will present an overview of general software development support tools, such as fourth-generation languages (4GLs) and CASE tools, and then try to compare them to SSS tools using 'Business Logic Component Technology' and general tools.

The development of SSS tools themselves was carried out without much study of worldwide tool trends. In retrospect, however, we now know that while capabilities, such as those in general tools were being built in, some capabilities where made to project out. In short, SSS tools were not so special as to be incomparable to other tools.

Accordingly, we set out to make the RRR family the best component-based reuse system in the world by studying worldwide tools and incorporating the more advanced aspects of other tools as part of the investigation of an RSCA. Note the RRR family is a refined version of SSS.

While reading this chapter, please also refer to ***Chapter 4 "Software Development Productivity."*** *After considering whether to put the content of Chapter 3 or 4 first, I finally decided to arrange the chapters as you see here. I probably should have written this book so that you could read the material in these two chapters together.*

3-a Upper Process Support Tools and Lower Process Support Tools

Tools that support software development are categorized as upper process support tools and lower process support tools depending on which development processes they support. There are also cases where three categories including middle process support tools are used, and there are even more detailed classifications.

The segmentation of development work is probably an effective means for finding good development procedures, but defining development processes too narrowly is a problem. The reason for this is even if we divide development processes into segments, work will not necessarily be easy for human beings by following such segmented processes. As will be discussed in **4.1 "What is Software Development Productivity?**" software development is entirely different from manufacturing hardware (goods) on a conveyor belt. Just because intellectual work during software development is cut up into small chunks does not mean it will become easier. To begin with, the portion of software development that can be processed automatically can be handled by computers, so there is no need to have programmers go out of their way to do it. Consequently, the work that human beings should carry out during software development is that which is intellectual, and partitioning such portions is almost meaningless.

However, since leaving the categorization of development processes ambiguous may also result in a hotbed of illusions, some sort of categorization is necessary. This book categorizes them simply as upper processes and lower processes and provides clear definitions for each category as follows.

This book defines **upper processes** as those wherein people understand the real world (and its periphery) that will be the target of computer processing and clearly articulate the image of a business system as requested specifications, and **lower processes** as those that implement the requested specifications, or in other words, processes that create software for building a business system based on the requested specifications. Furthermore, it defines **upper process support tools** and **lower process support tools** as tools that support the work in each of those processes.

These definitions always presume that human beings will be at the center of the work. Tools that did not require any human intervention whatsoever, i.e. tools that automated all development work, would be of a much higher level than the support tools defined here.

3-b Perceptions of Upper Processes and Lower Processes

Two proposals on how to perceive the relationship between upper processes and lower processes (see **Note 5**) have been made. The first is the **waterfall model**. The perception here is that processes flow from higher to lower ground like water over a waterfall and never reverse course. It is normal to reverse course from lower to upper processes in actual development work, but if it is small enough that it can be ignored, this perception is fine.

Taking this perception of development one step further, the ideal for development based on the waterfall model is making sure you never have to go back from a lower to an upper process. An example of where this would be ideal is large-scale development by a large number of developers. Since it is said that the amount of work for dealing with specification changes increases in proportion to two times the number of developers, particular care is taken not to produce specification changes in development by large numbers of people. Going back from a lower to an upper process as little as possible is sought in such cases, and perfecting work in upper processes becomes a crucial issue. This results in a focus on upper process support tools. Note the upper process support tools will be described in-depth in **3.1 "Upper Process Support Tools."**

Note 5: This book uses the terms upper process and lower process because they are widely used. However, these terms are tainted by the waterfall model and include a nuance that brings to mind the term upper class. To avoid being influenced by such nuances, it would probably be best to use terms like the **clarification process for requested specifications** and the subsequent **implementation process**.

Another way to perceive upper processes and lower processes is the **spiral model**. This perception considers development as proceedings much like going down a spiral staircase; the upper processes and lower processes go round and round. This concept holds that revision and reflection will lead to progress, but there is also an element of waste here when you have to go back from a lower to an upper process because of a development mistake.

This may seem contrary to what you would expect, but you could also consider development based on the spiral model to be ideal. People who think the waterfall model is ideal will probably object to this by saying, "How on earth can you think that going back from a lower to an upper process is ideal?" Even if going back from a lower to an upper process is sometimes unavoidable, it is wrong to call it an ideal. However, we are supporting the spiral model because there is a human engineering idea that idealizes adjusting to human characteristics. This concept too is quite feasible. For one thing, when humans create magnificent works, they are constantly revising what is being made. Consequently, even in software development, revision is what makes it possible to make magnificent products.

Development that idealizes the spiral model shifts to the lower process and creates a prototype system for the time being only after forming an image, to a certain degree, in the upper process. Prototyping

support tools (discussed later) play a major role therein. Note that this sort of development method is known as **rapid prototyping** or **rapid prototyping development**.

As for how the first prototype system made is handled, there are cases where it is created then abandoned and cases where it goes through a number of versions leading up to the full-fledged system. The following paragraphs will try to depict development for the latter cases.

Once the stage where the first version of the prototype system has been created, we return to the upper process. Next, unlike the first time, the prototype system already exists to a certain degree, so it can be used to find inconsistencies with the real world and refine requested specifications. Requested specifications are easier to clearly define when there is a system you can see with your eyes rather than one that only exists in your mind. Once such requested specifications are clearly defined, the lower process is shifted to, and then changes are made to the prototype system based on the latest requested specifications.

By going back and forth between the upper and lower processes in this manner, the system will be updated, and once it has gone through one hundred versions for example, a business system that can actually be used will be completed.

Such a depiction may give the impression that upper and lower processes are quite far apart, but since there are also cases where these processes are repeated in a short cycle much like a painter making one brush stroke only to change it, they can be seen as being in perfect harmony. With that in mind, you can probably understand why this is no exaggeration.

Topic 4: End-User Development and the Spiral Model

End-user development refers to the end user, who is the user of the business system, carrying out development work, not development specialists (custom business program development firms and enterprise information department staff). Since end users normally already have a good understanding of the real world and its periphery that will be the target of computer processing, upstream processes can be greatly reduced when end users carry out development. Consequently, it would seem good for end-user development to carry out development based on the waterfall model. However, it is development based on the spiral model that is more common because it is extremely difficult for end users to pin down an image of the business system they want at the very first stage when they have nothing developed yet. Generally, as a business system starts taking shape, various requests start bubbling up.

In one story I heard from someone who had experienced end-user development, the person was able to realize the effect of really utilizing a computer while breaking in the business system that had been created. The system was for supporting the taking of customer orders by phone and by providing the support described hereafter, it gave customers a good impression about the company and enabled high sales growth.

That business system offered support such that immediately after an order where a customer says something like, “That product was great, so this time give me twice as many as last time,” the company representative could answer, “Okay, so that’s n units of product x.” It also offered support so that the company representative could liven up the conversation by saying things like, “By the way, how did you like y? Isn’t your stock about to run out? How about ordering some now?” After some functional tuning of this business system, it was able to make customers think that all the company’s staffs were constantly thinking about them no matter who at the company picked up the phone.

Awareness and extensive use by end users who will answer the phones and operate the business system was crucial for achieving this, but the business system end also required update after update. You could say

that integrating the staff that answers the phones with the business system mandates the adoption of a spiral model development method.

3.1 Upper Process Support Tools

In this section we will attempt to take a deeper look into upper process support tools that help us to understand the real world (and its periphery) which is a target for computer processing, and support the processes that clearly define the image of a business system as requested specifications.

First, let's reflect back once more on why upper processes are regarded as important.

Think about when a demonstration held right before the development of a business system is just about to be completed. The developers realize at that point there is a gap between what the system does and what the customers want, and so they must make some changes. If there are only a few changes to be made, they will be lucky because it will only be a minor inconvenience. In the worst case, however, they will have to extensively overhaul the business program and probably will not meet the delivery date. After having such an experience, people who idealize the waterfall model will more than ever feel that the complete pinning down of the business system image in the upper process is crucial. In contrast, people who recommend development based on the spiral model will probably say that continually making revisions early is important because the complete pinning down of the business system image at an early stage of development is impossible.

There are various ways of thinking, so let's return to the discussion of the importance of upper processes and try to put ourselves in the position of a custom business program developer. When you take on the development of a custom business program, failing to completely pin down the image of the business system will have dire consequences. The customer will claim that their endless stream of requests which come after the estimate are covered under the original contract, resulting in work that exceeds the original estimate and may drive the development project into the red. That is why the clear definition of requested specifications is crucial to turning a profit.

However, the complete pinning down of the business system image is quite hard work, even for an experienced, highly skilled person. As a result, it would be nearly impossible for software tools. You will be able to better understand this if you imagine the following situation.

At the beginning, customer requests will not always be clear, and sometimes during development, customers will change their mind and want to make changes to their requests. Furthermore, requests may differ between customer (enterprise) departments (there will be differences in values), and as a result the gathering of requested specifications is never easy. Consequently, it is said that talking with the end user is far more important than sitting in front of a computer. It is also said that knowing the mechanism of real intentions (informal system) that cannot be elicited normally is crucial to the development of truly helpful business programs. Since there is a limit to the information that can be obtained through an interview for gathering requested specifications, there are some who are of the opinion that employees must be determined to learn job details and undergo OJT (on the job training). The job of gathering requested specifications is more social science than computer science.

Is it possible to use tools to support this sort of complicated work? It is impossible to approach this directly, and using tools to carry out this sort of work is mostly unthinkable. However, only a portion of

upper process work can be supported in some form. The kind of support described hereafter is actually carried out using tools.

- Support for writing documents containing requested specifications that outline the business system image (Word processors)
- Support for writing various types of diagrams (Upper process CASE tools)
- Interview support (Questionnaire based on development know-how for the same type of business system)
- Simulated-experience support (Prototyping support tools for forms and documents)

In addition, there are means of communication, such as e-mail and groupware, and they have an indirect support effect at minimum. However, we will omit them from this book to keep us from diverging too far from our subject. Let's take a look at each kind of support by the above-mentioned tools.

3.1-c Writing Support

A part of the process of understanding the real world that will be the target of computer processing and outlining the business system image is writing documentation and diagrams. Writing is necessary for leaving a record, and it is also effective in rationally outlining thoughts. The resulting documentation and diagrams may sometimes be kept as formal output, while other times merely temporarily used during work processes. In either case, a certain percentage of upper processes consist of such writing work.

Using a word processor when writing documentation offers major support because it will be easy to add, change, delete, and otherwise edit text. Using a CASE tool diagram editor when writing diagrams (a type of design drawing) also offers major support because the burden of revising tedious diagrams will be lessened.

Not only do support tools have a direct effect by means of their distinct functionality, but they also have a cooperative effect that is obtained by making work fun. No one wants to read illegible handwriting to which lots of corrections have been made, but when you see nice word processor output, polishing the writing becomes enjoyable and you want to unify the format whenever you find parts that are not aligned. For example, I could safely say that this book could not have been completed without a word processor. Rereading one's own writing is an overwhelming job for people who write illegibly and poorly, but word processors have the power to change it into a fun job.

This sort of cooperative effect is certainly present, but opinions are divided regarding the degree of effect. Remaining cool-headed is of utmost importance because there is also excessiveness in the evaluation of upper process CASE tools that we will talk about hereafter.

Note that although we discussed writing support in upper processes in this section, it goes without saying that similar writing support in lower processes is also possible.

3.1-d Upper Process CASE Tools

CASE is an acronym for Computer Aided Software Engineering, and CASE tools are a general term for tools that support software development. Consequently, CASE tools in the broad sense also include fourth-generation languages (4GLs). However, while the term 4GL gives the impression that it is native to the business field, the term CASE seems to have an academic air about it. The difference in nuance between 4GL and CASE leads many people to think that CASE does not include 4GL.

When a boom occurs, terms end up getting colored, and so when you mention CASE tools without any modifiers, it normally means upper CASE tools that succeeded in the tool business in the latter 1980's in the U.S. and Europe, i.e. upper process CASE tools that support the upper processes of development. In the following paragraphs, we will zero in on such **upper stage CASE tools** and attempt an across-the-board evaluation of them.

The objective of using CASE tools is the automation of the upper processes of software development and the automation of programming. Upper process CASE tools that held such an ideal and started supporting the upper processes of development experienced a sudden boom. However, the boom did not last even five years because upper process CASE tools were nowhere close to that ideal, resulting in too large a gap between their ideal and reality. Such a negative evaluation would have been swept away by the boom's momentum and ignored if it had been staged at the peak of the boom. However, the boom has already passed, so if we were to return to the former battlefield so to speak, and evaluate what has passed, there wouldn't be any shooting no matter what we said.

First, let's take a look at what upper process CASE tools actually are.

Starting in the 1970's in the U.S., professors of software engineering began advancing a variety of style diagrams based on such things as structure analysis. They were based on data flow diagrams (DFD), entity relationship diagrams (ERD), and so on, and were effective for analyzing the structure of business processing.

However, when you try to write a diagram, a number of revisions are required until you can complete it. Often you are briefly delighted that you have completed a nice diagram only to face the tragic reality that you have to totally revise it once you have calmed down. Doing this with a pencil and an eraser is troublesome and will not lead to an attractive diagram. Moreover, it is just not smart.

Upper process CASE tools have revolutionized the writing and revising of diagrams. They have changed it so that you can use the mouse to write and revise graphical diagrams on your PC screen. Upper process CASE tools could be likened to a word processor for diagrams or a CAD (computer aided design) system for writing diagrams, but that is a rather narrow way of viewing them. There were also upper process CASE tools that automated some of the consistency checking for diagram content. However, such checking was not at an intelligent level, but rather was something like a feature found in a word processor for checking the correspondence between the table of contents and body in a document.

Upper process CASE tools certainly are effective for writing support. However, if you evaluate them from the perspective of how much they support the clear definition of request specifications, the answer is not much. For an example of a method for evaluating the degree of support, refer to **Appendix 3 "Example Using Build-Up Method to Determine Improvement Rate of Productivity."** To begin with, these sorts of diagrams are certainly not easy to understand for end users who will be judging the adequacy of specifications. However, in the U.S. and Europe, there are analysts who specialize in analyzing requests, and these people write diagrams based on structured analysis and other techniques. These diagrams are effective when specialists are analyzing the real state of business processing. Consequently, we cannot say there is no demand for this kind of tool. Still, it is hard to imagine that this alone would lead to a boom.

If you look into the background and causes of the boom, you will find there was a strong underlying need for improved productivity in business program development, the attractiveness of tools that employ

graphical representations, the pleasing sound of the cutting-edge technology of CASE, clever tool vendor advertising, and cooperative relationships between computer journalists and consulting firms.

When people like computer journalists were captivated by the attractiveness of CASE tools and started making a big deal about them, information department staff at companies had to obtain a certain degree of knowledge about them because top executives there would ask such things as, "The introduction of CASE tools at other companies have had an effect, but what are we doing?" That led to the purchase and study of CASE tools. However, upper process CASE tools are not easy to understand, and this consequently led to more work for consulting firms. A market consisting of a variety of interrelated people was thereby formed.

Additionally, when companies studied CASE tools, it was normally planning department personnel who were involved in establishing the company's long-term vision that carried out the job rather than those who actually did development work. Such people tend to buy into the notion that CASE tools pursue the ideal. There is nothing inherently wrong with that, but it sometimes leads them to ignore reality and to have overblown expectations, and such expectations may even take on a life of their own.

Once expectations became overblown, it was hard to convince people that the reality was actually different, and once large numbers of people started participating in the business, it became difficult to easily pull out. The only way to proceed from that point was forward, and hence the boom in upper process CASE tools was born.

In due time, many people figured out that their expectations were left unfulfilled, and they soon began joking that upper process CASE tools were neither software nor hardware but rather, shelfware (something that just sits unused on a shelf). The boom of the upper process CASE tools died down at the end of the 1980s in the U.S and Europe.

Upper process CASE tools were writing support tools for diagrams intended for request analysis pros, and they were a far cry from the ideals of software development automation and automation of programming. Furthermore, they were not something that just anybody could use in place of experts in a particular field, but rather they could only be used by people with training in structural analysis and so on. You could also say that since these sorts of tools only support some work in upper processes, we should refrain from exaggerated language. Of course it is an entirely different story if you were trying to make a profit out of such confusion.

We have so far talked about the circumstances of the upper process CASE tools' boom. There was a later movement attempting to fuel a boom by introducing integrated CASE tools, but it did not have much success because most people knew that there were not yet any tools, and so on, that were able to truly automate all development work. They may appear one hundred years from now but not in the near future, and thus there is no need to worry about them. Nevertheless, the illusion that they existed was definitely there. The fact that computers are still seen as being in the realm of the mystical and the exaggerated claims by tool vendors probably encouraged such illusions.

Topic 5: Exaggerated Tool Claims

For a period of time, the claim "using our tools boosts productivity *n* times" was being made. This is clearly an overstatement. A look into the multiplication factor reveals it is like instantaneous wind velocity so to speak and only applies to a specific case with favorable conditions. It is almost never a statistically meaningful value. There were also self-serving comparisons that cannot escape mention. Examples included cases where productivity improved because people with development experience for the same

types of business programs, or people with extraordinary skill were doing the work and comparisons with normal development where estimates tended to be low. There is no way to easily measure the multiplication factor for productivity, which gives free reign to those who say it is okay if it cannot be ascertained. However, overstatements are easily seen for what they are, and so exaggerated multiplication factors were eventually regarded as advertising propaganda. The lack of any satisfying explanations about how to measure the multiplication factor for productivity is now commonly regarded as advertising propaganda rather than an objective assessment.

For information on measuring the multiplication factor for productivity as objectively as possible, refer to **4.2 "Various Ways of Measuring Software Development Productivity."** At the same time, you should also refer to **"Topic 8: How Much Do Tools Improve Productivity?"** found in **4.3 "Is Software Development Productivity Improving?"**

3.1-e Interview Support

In this section, as in a **questionnaire** we will refer to the question-and-answer document based on customer orders and what was noticed when developing a business system. A questionnaire could be called a sort of collection of know-how for building a business system, but it could also be seen as a tool to use at customer interviews.

If one customer ordered something that another one had ordered in the past, it just might be something any customer would want. Consequently, it is quite effective to interview customers using a questionnaire created based on this. One question could be "In general [meaning other customers], there are such and such choices, so what would you choose in this case?" This will make customers realize things they had never thought of before. Also, asking such questions will allow you to discover ahead of time requests that customers would probably have made at some later date.

However, in the parameter customization of business packages as discussed in **1.1 "Differences between Custom Business Programs and Business Packages,"** it is usual to interview customers and then set parameters based on their answers. Since these parameters are information for giving certain attributes to the business package and creating a business system the customer seeks, they can be referred to as a sort of questionnaire for the pinning down of the business system image. On the flip side, you could say a questionnaire for a custom business program corresponds to customization parameters for a business package.

Since any attempt to establish new customization parameters for a business package will be accompanied by work for building in what is necessary, you cannot easily increase parameters. Since there are probably also sales policies that try to avoid customization, a parameter will only be defined after careful screenings.

At the same time, questionnaires for custom business programs aim to comprehend all requested specifications. Far more questions than the number of parameters for business package customization ought to be included, and by simply having customers answer these questions, you should be able to form an image of a business system that is a much better match for the customer than a business package.

As we will discuss later, **the need for creative adaptation** (**NCA**) is high in the business field, and so in addition to gaining an understanding through such canned questions, fulfilling various orders from customers will also be important. (NCA will be described in-depth in Chapter 5, but for now, refer to "**Keywords for Understanding This Book**" for a brief explanation.) However, I think the first logical step in approximating customer requests would be to pin down the image of the business system according to

questionnaire answers. It would then be efficient to adapt to customer requests through a second approximation stage, third, and so on.

3.1-f Clarification of Requested Specifications Supported by Simulated-Experience

The points of contact between a business program and its end user are twofold as follows. There are almost no other junctions.

- Information in forms displayed on screen and reports that are output; and
- Data input operations while viewing displayed information

At the initial stage of business program development, it is very effective to show end users the style of forms and report forms (including data input operations too if possible) to give them a simulated experience. This is effective because it allows them to verify requested specifications, which tend to be vague on paper, by putting a business system close to the real thing in front of the customer's eyes. Such simulated experiences are a means of verification that even people unaccustomed to computer systems will find easy to understand, and they are far easier to understand than the diagrams that appear in the descriptions of upper process CASE tools. Diagrams are significant in other ways, so this is a rather harsh comparison. A good analogy would be providing replicas and photographs of food rather than a written menu when customers order so that they will know exactly what they will get.

Since this enables end users to have an early simulated experience of a seemingly finished business system, they may realize many things, resulting in a large number of orders, but at least such orders can be dealt with at an early stage. Specification changes occurring in post processes will thereby be reduced.

This sort of simulated experience requires the creation of a prototype system that can be viewed and operated. To that end, prototyping support tools that help the design of form and report form style among other things are used.

As for the timing of the simulated experience, to what degree should the **prototype system** be operating before it is shown to the end user? Obviously, forms that respond when operated are better than those that are just for show. However, since doing this makes the creation of the prototype system longer, it will delay the simulated experience. That is why normally the creation of a prototype system is wrapped up at an appropriate point so that the end user can have a simulated experience.

However, if a business system close to the one planned for development has already been developed, giving users a simulated experience of that system will be effective in verifying requested specifications with no problem. An actual system feels much more real than a prototype system that lacks substance. Promoting component-based reuse will make such things feasible. However, this will indicate to the end user that there is a similar business system already in operation, which puts business program development firms in the difficult position of billing only for the cost of what was newly developed. But development by reuse is much more likely to produce a system that will satisfy the customer than haphazard new development. Shifting to reuse-based development instead of paying and billing for development costs will likely change how simulated experiences are given, and this will enable an improvement in customer satisfaction.

3.1-g Magic Applied between an Upper Process and a Lower Process

Some prototyping support tools are designated as upper process support tools while others are designated as lower process support tools, but they are essentially no different in content. Yet, if the type of the tool is left unclear, caution must be observed, lest the clever words of tool vendors fool us. Specifically, we might

mistakenly think software resources (forms, report forms, and programs) that must be developed in lower processes, after upper processes end, can be automatically generated by tools.

If tool vendors were doing this in bad faith, it could be called fraud, but perhaps they are even fooling themselves.

A description of tool vendor misperceptions will follow. Note that the correct perceptions for avoiding being fooled are contained in parentheses, but it might be fun to skip the portion in parentheses the first time.

> During development based on the waterfall model, clearly defining requested specifications by simulated experience is upper process work, and hence, creating a prototype system that is required for a simulated experience is also upper process work.
>
> (We must not confuse the waterfall model as an ideal for actual development. It is usual to go back and forth between upper and lower processes in actual development. Since the creation of a prototype system is carried out to accelerate lower process work, it should be regarded as a lower process.)

> During development based on the waterfall model, shifting from upper processes to lower processes comes after the clear definition of requested specifications by means of simulated experience.
>
> (When creating a prototype system which is necessary to verify requested specifications by means of simulated experience, there are already many temporary shifts to lower processes. Therefore, it is not the first time shifts are made to lower processes when requested specifications are clearly defined by means of simulated experience.)

> Some forms, report forms, and programs that must be developed by lower processes can be automatically generated from the products of the prototype system created by upper processes.
>
> (The conversion of forms, report forms, and programs created for the prototype system, so that they will be available for later work, should not be called automatic generation. It should be called software resource conversion. Furthermore, lower, not upper processes develop the products created for the prototype system.)

Now I would like to repeat the correct point of view as defined by this book which will help to avoid being fooled into thinking that some forms, report forms, and programs would be automatically generated.

Prototyping support tools should be regarded as lower process support tools because they are first used when creating a prototype system, and in the broad sense, they are nothing but support tools for creating a business system. There are cases where prototype systems are created and then abandoned, and others where they are created and then updated through a number of versions until the finished system. In either case, the creation of the prototype system is nothing but the beginning of business system creation, and it should therefore be regarded as lower process work.

However, work that clearly defines requested specifications and is supported by simulated experience is bona fide upper process work. This wording may give the impression that lower and upper processes are being intermingled, but I would like you to think of work that clearly defines requested specifications and is supported by simulated experience as something that is carried out by returning to upper processes after lower processes. In short, after creating the prototype system according to the spiral model, you return to the upper processes armed with that system which you then use to find inconsistencies with the real world and refine the requested specifications for the business system.

Thinking this way will prevent us from being fooled. Up to now there have been people who were fooled into being thankful for this sort of automatic generation. If some sort of conversion processing is required for midterm development products, it should be fulfilled using tools. Care must be taken because easily misleading wording, such as “automatic generation” is used in place of conversion.

If you take the levelheaded point of view of conversion processing for midterm development products, doubts as to whether such conversion processing is really necessary will spring up. One thing that comes to mind is whether conversion processing is even necessary. If it is, it should certainly be kept to a minimum. For example, if you were to use a prototyping support tool that is marketed as a lower processing support tool, there would normally be no need for conversion processing.

There are similar cases of being easily fooled by claims that conversion from pseudo code to program code is automatic generation. Care must be taken so as not to misunderstand or to be fooled.

Right from the start there is a large gap between upper and lower processes. Furthermore, there are not yet any tools that can automatically generate lower process software resources from upper process information in the true sense of the term. However, you must remain wary because the hopes for such tools create the illusion that they exist.

Saying that it’s just elaborate magic might be unfair conjecture, but there is also a mechanism by which to classify development processes in detail. This book uses the two easy-to-understand categories of upper process support and lower process support, but it establishes a large number of process classifications, such as process 1, process 2, process 3, and so on. There has also been propaganda showing off powerful tools by emphasizing the generation sequence of using the tools to generate what is necessary in process 2 from the output of process 1, generate what is necessary in process 3 from the output of process 2, and so on.

As astute readers will have already noticed, such propaganda applies magic that involves upper process support and lower process support to many processes. Furthermore, such propaganda is reminiscent of the card trick in which a specific sequence (see **Note 6**) is used to guess what successive cards are. Since software development does not proceed like a conveyor belt, dividing up the processes does not mean it becomes easy. It does not make sense to divide processes into units smaller than our ideas or thoughts. Armed with such doubts, a look into why tool vendors have established such process classifications will reveal that these classifications are convenient for them. It is something like the gerrymandering of electoral districts to get an unfair advantage in an election.

In actual software development, a variety of development work proceeds concurrently even within the mind of individual developers. It becomes all the more complex when there are multiple developers. Finding clever development procedures by classifying development work rather than development processes therein and then utilizing them is meaningful and understandable. However, trying to forcibly conform work to a specific development process sequence will not end up going well. Actual development (see **Note 7**) involving the creation of something new will not necessarily proceed according to such a process. Consequently, such development process sequences are usually just for form’s sake.

Note 6: Prepare ahead of time using a deck of cards that is face side up. Take the first card, turn it over, and then place it on the other side of the deck. Now in front of your audience, say, “I will guess cards one after the other.” Hold up the deck with the first card facing the audience so that they can see it while you memorize the card on the opposite end (the end the audience cannot see). Next, bring the deck behind you and place the card you memorized on top of the card you showed the audience. After that, bring the deck

back out in front of you and while holding it up to the audience, guess the top card. By repeating this sequence, you can guess one card after the other.

Note 7: In customization work for business programs that use 'Business Logic Component Technology' processes are easily established only for those parts wherein the work method is easy to understand, and the amount of work in that case is also easy to estimate. You could call it the effect of eliminating rehashed creation. However, even if you call it rehashed creation, in conventional development that often accompanies certain types of creation, work does not proceed as per the processes.

3.2 Lower Process Support Tools

This section will employ a study of lower process support tools (tools that support the creation of software for building business systems based on requested specifications clearly defined in upper processes) that was carried out as part of the investigation of an RSCA to report how these types of tools have come to be perceived.

3.2-h Trends in Lower Process Support Tools

This is old news, but productivity improvement by assemblers and compilers has been nothing but remarkable. The use of interactive computers has also been extremely effective. The evolution from the use of paper tape and cards to line editors and finally full-screen editors has also greatly contributed to increased productivity. At the leading edge of this evolution have come special editors for laying out forms and report forms, and word processors for writing documentation. These things greatly contributed to improving productivity as lower process support tools for a period of time following the beginning of computer history.

Here is a list of the main support functions and types of lower process support tools:

- Support for conversion from programming language to machine language (assemblers and compilers)
- Support for the creation of design documentation (word processors)
- Support for the layout and design of forms and report forms (special editors)
- Support for the development of prototype systems centered on form and report form layout
- Support for the design of files and databases (special editors)
- Support for the reduction of program lines (fourth-generation languages)
- Support for improving program visibility (chart editors)
- Support for component-based reuse (object-oriented technology and fill-in systems)
- Support for interpretive immediate execution and debugging (interpreters)
- Support for debugging/testing work (debuggers)
- Support for managing development resources

Although it cannot be called a tool, the following software resource that can be ranked as a component is crucial for improving productivity.

- A collection of functional routines (general subroutine libraries)

Although the term "design support" was used in the above-mentioned tool list, almost all lower process support tools are centered on taking over from humans the portion of design work that can be mechanically

processed, and is thus not really design-like work. For example, assemblers take over from humans troublesome address calculations as well as conversions from machine instruction names to instruction codes. Such calculation and conversion work certainly is not anything related to the main portion of design, but it is true that such work is difficult and troublesome for usual people. Best of all, these tools make computer processing easy. That is why incorporating such tool functionality has greatly contributed to improved productivity.

Since there were a number of seeds (materials, measures, mechanisms, and/or structures) that made computer processing easier and had a major effect in the early days of computer development, incorporating them into tools made it possible to greatly improve productivity. However, the seeds that had a major effect were soon exhausted, and productivity from that point barely improved at all. Compared to the remarkable progress in the early days, productivity growth by tools thereafter was stagnant due to the lack of seeds. A more in-depth discussion of this can be found in **4.3 "Is Software Development Productivity Improving?"**

By all rights, we should confront head on the improving of the productivity of intellectual work that forms the heart of design. In short, we should develop tools that have computers rather than people carry out the central part of design work. Unfortunately, there is little hope of this happening.

Accordingly, lower process support tools, i.e. tools for supporting the creation of programs for building business systems, have come to be developed for providing a pleasant software development environment more than for contributing to further productivity. This is similar to how car development proceeded from the maturation of basic functionality to the provision of pleasant passenger space.

Amid our investigation of an RSCA, we identified and studied in detail the following two types of tools from the previously mentioned list that are particularly focused on improving productivity.

- Support for component-based reuse (object-oriented technology and fill-in systems)
- Support for the reduction of program lines (fourth-generation languages)

3.2.1 Fill-In Systems

We have already discussed objected-oriented technology in Chapter 2 as being related to the support of component-based reuse. Accordingly, the following paragraphs will report the results of the study carried out for fill-in systems that aim for the componentization of software as part of the investigation of an RSCA.

Fill-in systems are composed of programs from both skeleton routine and supplementary routine units (hook methods) that flesh it out as shown in **Figure 3-1**. In other words, slots (or hot spots in the current vernacular) are provided within the skeleton routine, and by plugging supplementary routine units into them, we can finish creating a business app. This aims for reuse by using skeleton routines and supplementary routine units as components, and it can be regarded as a sort of component-based reuse system for software.

Since this fill-in system really seems like a synthesis system for components, many names have been suggested from all corners of Japan, and they all are basically the same thing. They include template system, component synthesis system, stopgap system, program paradigm, and so on. SSS has also been regarded as being a sort of fill-in system.

However, fill-in systems other than SSS, all too often ended in failure. This is because two branches can be expected in a fill-in system, and nearly everyone who developed such a system ended up heading in the wrong direction. I would like to introduce conclusions we reached after studying how they headed in the wrong direction.

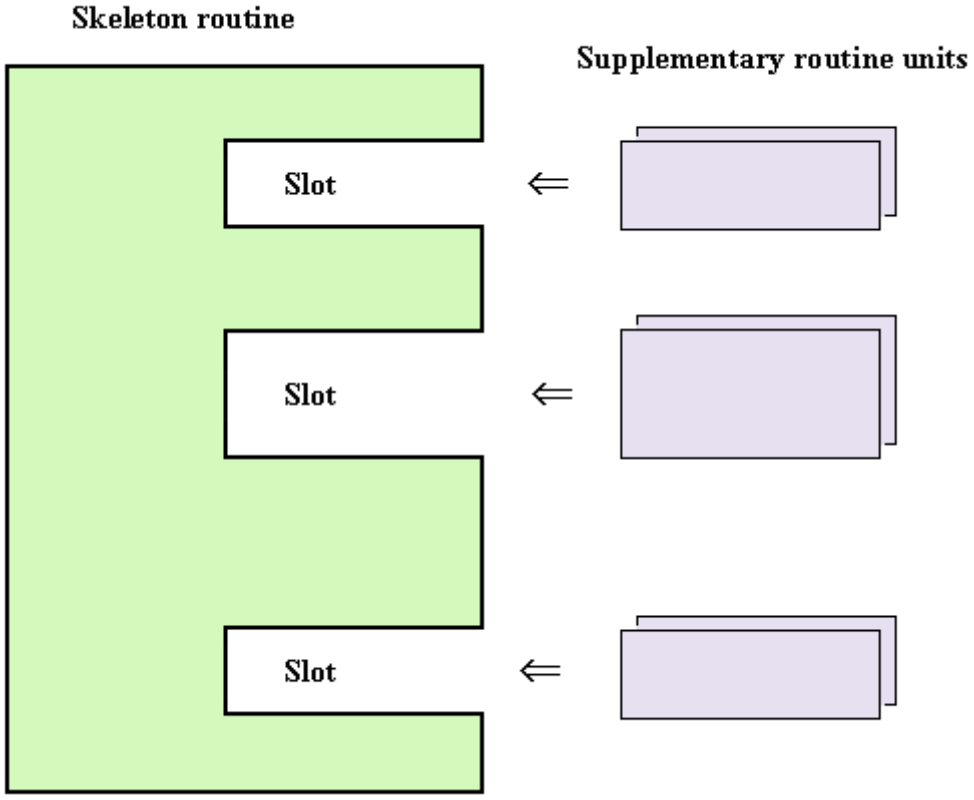


Figure 3-1: Skeleton Routine and Supplementary Routine Units (Hook Methods)

3.2.1-i First Branch in a Fill-In System

The first branch requires deciding between emphasizing plug-in component synthesis tools or the components themselves. Even if developers of this sort of system vaguely feel the importance of components themselves, once they start tool development, they end up getting sucked into the fun of development. This ends up being an error in judgment at the first branch.

Actually, tools that are plugged in and synthesized have been developed all over the place, but we have never heard of a case where such tools have been skillfully used. The reason for this is components like skeleton routines and supplementary routine units can be any number of things, and therefore, the meaning of the components becomes uncertain. It is like being confronted with a mountain of program fragments of various sizes and various purposes and trying to put them in order only to fail. Such an unordered mess is nothing but useless garbage. Trying to call program fragments components is an exercise in futility.

When that did not fly, some people who were transfixed on tool development resorted to developing a component retrieval system on top of a fill-in system. However, such component retrieval systems were nothing more than systems for retrieving garbage. Garbage can sometimes be useful, but there is certainly no guarantee that it will always be useful.

Incidentally, component retrieval systems are also sometimes developed for retrieving components, such as common subroutines, and this is done independently of a fill-in system. Let's take a look at how component retrieval systems generally end in failure.

Even if you have accumulated component sets consisting of many common subroutines and so on, it is all worthless if you cannot figure out if there is a component you want or you cannot find one you know is there. You would ideally like an easy way to find them. That is why the development of a component retrieval system becomes necessary.

The development of a component retrieval system will not necessarily solve the above-mentioned problems. Most component reuse systems, that accumulated many program fragments, provided a component retrieval system, and then started operating with the statement "Well, let's use it" have not succeeded. There are many reasons why they did not succeed, but the main reason was that users frequently could not retrieve the components they wanted and would thereafter not give the system a second glance. This is the same as tearing out a few pages from a dictionary and calling them a dictionary. Obviously, just a few pages cannot fulfill the role of a dictionary because the words you want to lookup will more than likely not be on the pages that were torn out.

The key to a successful component system is a component lineup in which the components can be found most of the time when users want them. If it is impossible to get so far, then decreasing the number of disappointing search failures by devising a means that allow users to surmise ahead of time whether the component they want is or is not in the component library would be the minimum common courtesy.

Almost no one realizes that the cause of failure lies in this area, but those who were fixated on tools and failed, as well as those who failed in such trials, will recoil at the mere mention of a component retrieval system or fill-in system.

Incidentally, Woodland Corporation emphasized that SSS was a fill-in system in its advertisements, but it seems this ended up being negative publicity because it caused users to back off or misunderstand the product.

Heading in the right direction at the first branch requires emphasizing components themselves rather than tools. No matter how magnificent a component retrieval system or fill-in system you develop, you will not be able to retrieve, plug-in, and then synthesize components that do not exist in the component warehouse. As we already stated, a component lineup is crucial.

To that end, it is important to organize components to a certain extent, develop the necessary components accordingly, and then prepare them so that they are always available. Storing piles of components haphazardly in a component warehouse will not lead to users finding the components they want.

Machine parts and electronic components are all intentionally developed with a specific purpose in mind. The same goes for components of software. Software that is not intentionally developed with a specific purpose in mind and not regarded as a component is junk or garbage without a doubt. Many people have experienced the gathering of many program fragments that were unintentionally developed into components only to find that such a component management system is almost totally useless.

3.2.1-j Second Branch in a Fill-In System

The second branch requires deciding between whether to view skeleton routines, or supplementary routine units, as common components. Since supplementary routine units tend to accumulate, developers of this sort of system are too eager for success and end up thinking that these are common components. Such people end up making an error in judgment at the second branch.

Viewing supplementary routine units as common components is almost the same thing as letting subroutines be common components, and it is zero progress compared to conventional methods. Plugging a supplementary routine unit into a skeleton routine means that the skeleton routine will call that supplementary routine unit. The opposite way of looking at this is the supplementary routine unit will be called to do some type of task by the skeleton routine. This means that supplementary routine units are equivalent to common subroutines.

There are also nitpickers who say, "Supplementary routine units are different from common subroutines," but if that is true, then saying they correspond to the expansion of macro instructions in assembler and C programming languages is even more fitting. In other words, it is existing technology and no novelty can be found there.

The calling of components in the broad sense involves a variety of means, such as subroutine calls, expansions of macroinstructions, system calls (supervisor calls) and message passing. Plugging in is nothing more than the conventional means of calling components.

Consequently, perceiving supplementary routine units as a common component will not produce a new productivity increase effect. Even if making supplementary routine units a common component does improve productivity, it would simply be a productivity improvement effect produced by a conventional component calling method, and therefore, credit should not be given to the fill-in system. In short, that productivity increase effect is nothing more than the effect of reusing a common component, which was already known to improve productivity.

Since fill-in systems can lead people in the wrong direction at these two branches, it is all too common to end in failure without having any effect whatsoever. However, heading in the right direction at the first and second branches as in SSS, should produce a decent effect. In other words, if you are able to perceive skeleton routines as a common component, you will be able to see fill-in systems in a new light. This perception means that the main routine that calls subroutines is actually a common routine, something that was never perceived before. Furthermore, if "common main routines" (called frameworks these days) exist, making them common components will open the way for reusing portions that have thus far been overlooked. In short, since a productivity improvement effect will be achieved in a different area than in the past, we can credit it to the fill-in system.

These noteworthy common main routines are generally not very familiar, but they actually appear in a number of places and contribute to productivity improvements. Calling them a framework in the narrow sense probably makes them easier to understand. Frameworks are also used in fourth-generation languages (4GLs), which have already received much attention, and so we will describe them in detail in **3.2.2 "Fourth-Generation Languages (4GLs)."** To become familiar with common main routines or gain an intuitive understanding of them, refer to **Appendix 4 "Demarcation of Figure and Ground When Recognizing Something."**

3.2.2 Fourth-Generation Languages (4GLs)

Fourth-generation languages (4GLs) are said to be the descendants of three previous generations of languages in the order of machine language (first generation), assembler language (second generation), and compiler language (3rd generation), but the name 4GL does not always accurately represent what they are. It would be more appropriate to perceive 4GLs as tools or frameworks for improving productivity aimed at the business field than as languages that pursue generality.

There is documentation that lists the ability of end users to use a 4GL or learn it in a certain number of days as criteria for whether a language is a 4GL, but such criteria are not universally accepted. If a vendor calls its product a 4GL, then it becomes one. Because of this, the number of 4GLs is equal to the number of their vendors and easily exceeds one hundred.

Out of the various types of 4GLs, this section will focus on mission-critical 4GLs that can be used to develop custom business programs and business packages in the business field and report the results of the study carried out as part of the investigation of an RSCA.

It is incorrect to contrast cutting-edge CASE with outdated 4GLs. 4GLs are ostensibly included in CASE in the broad sense, and even today most 4GLs contribute to improved productivity. You could even say that many 4GLs were solely designed based on experience in business program development in the business field and thus do not necessarily have any theoretical backing. Another way of saying this is that there were many 4GLs designed to achieve a substantial effect without ever explaining why productivity increased.

Initial 4GLs improved upon third generation languages, such as COBOL and were reborn as languages that skillfully harnessed the power of relational databases (RDB). There was a different specification at each company, but they were all the same in the sense that they allowed the desired data to be retrieved using a one to two-line statement. Since COBOL required ten to twenty lines to do this, 4GLs were certainly effective.

Furthermore, 4GLs made not only database access, but also database definition easy. When using COBOL to do such things, you have to define both data statements in the program and data items in the database. But with 4GLs, there is no need to perform such redundant definitions.

4GLs have also added the ability to easily define forms and report forms as an extension of database definition. Note that this ability has also played a role in making it easier to develop prototype systems.

3.2.2-k Event-Driven Systems

The products of such early 4GLs were gradually incorporated in third-generation languages as well. The relational database (RDB) access method was standardized as SQL and became usable from the third-generation language COBOL as well. Thanks to the capabilities of relational database (RDB) management systems, the redundant definition of data statements in COBOL programming also became unnecessary. The gap between third-generation languages and 4GLs was thereby narrowed, resulting in the dilution of the significance of 4GLs. To resolve this situation, 4GLs attempted a comeback by adopting an event-driven system.

In the following paragraphs, we must elucidate why event-driven systems can improve productivity, but first the reader needs to understand what an event-driven system is and how things have changed compared to before and after such systems were adopted.

Let's start with a general explanation. In an event-driven system, when an event related to a certain object occurs, the system runs an event procedure corresponding to the event classification of that object. A good example of an object would be a form or a report form, or a data item within them, while a good example of an event classification would be the classification of an operator's action.

For example, an event-driven system is characterized by an event procedure running in response to an event such as an operator entering data into a data item on a form or pressing a function key on the keyboard. Associating operator actions with events allows programs to respond to such actions.

Adopting an event-driven system results in a program approach that differs from the traditional programming style for COBOL. Programs must be partitioned into a number of event procedures corresponding to each event classification of each object. Furthermore, the creation of a sluggish program that flows from top to bottom would result in a multitude of lumpy event procedures that run when called if we were to adopt an event-driven system.

This description is probably difficult to understand for anyone except those who know about event-driven systems, so I would like to describe in a slightly easier-to-understand manner how things have changed compared to before and after such systems were adopted. Note that readers with no programming knowledge should refer to **Appendix 1 "What Does Running a Program Mean?"** to gain the requisite information.

Just like with early 4GLs, programs written in COBOL execute in order starting with the first statement of the main routine and then end at the last statement of the main routine. In addition to this basic flow, COBOL programs sometimes also call and execute a subroutine before returning to the main routine, execute a loop statement, or skip statements to go backward or forward. Anyone with even a little bit of programming knowledge will probably see that this kind of program execution method is extremely common and is just like most others.

Programming in the COBOL language is nothing more than the writing of statements comprising main routines and other statement comprising subroutines with the intent of enabling this sort of execution method. Consequently, when developing a program it is necessary to develop both main routines and subroutines, except previously developed subroutines.

On the other hand, later 4GLs that adopted the modern event-driven operation style have basically the same program execution method described above, but at the same time their 4GL programs are different because they are comprised of a multitude of fragments. Another difference accompanying this is that the execution trigger for each fragment is predetermined. The fragments we are talking about here are formally known as event procedures. They are like subroutines that are called as the trigger of an event. The triggering of an event is predetermined, such as the entering of data in an item on a form, and that is what is called and executed. When such a predetermined event actually occurs, the event procedure that serves to process it is called to execute the fragment. The fact that the trigger for executing event procedures (subroutines) is predetermined is something new.

However, the fact that event procedures are being called means that something must be calling them. We will name that something a 4GL operation base. This allows us to say that 4GL operation bases call event procedures and determine the trigger for executing each one.

Assuming this, the development of event-driven programs means the development only of event procedures called from common main routines (a framework in the narrow sense) known as a 4GL operation base, or in other words, the development of only subroutines.

What I would like to stress here is that in the past it was necessary to develop both main routines and subroutines, but with event-driven programs, only subroutines need to be developed.

This change has resulted in increased productivity. In short, the adoption of an event-driven system eliminates the need for main routines that were once routinely developed, and this has raised productivity. Main routines are no longer needed because the 4GL operation base fulfills the role.

3.2.2-l Why does 4GL Improve Productivity?

The name fourth-generation language (4GL) gives one the impression that it has magnificent language specifications. However, despite having such a name, later 4GLs do not feature such specifications. An in-depth look into commercially available 4GLs reveals that their language specifications are essentially no different than those in third-generation languages, such as COBOL. For example, the statement equivalent to an IF statement in COBOL is no simpler when you use a 4GL. While in COBOL you can move a bunch of data all at once simply by writing a one-line CORRESPONDING MOVE statement, there are some 4GLs that do not provide such convenient functionality in their language specifications. This sort of comparative investigation reveals the fact that the language specifications of commercially available 4GLs are not all that different from those of third-generation languages.

Nevertheless, using a 4GL certainly does improve productivity. This was a curiosity to the author. During our investigation of an RSCA, we eventually realized that fact after we designed a mechanism similar to a 4GL as a trial. The reason for this productivity improvement, as we have already explained, is that main routines no longer have to be developed thanks to the 4GL operation base. A look into 4GL operation bases will reveal they carry out processing related to operation characteristics, and hence eliminate the need to write programs related to operation specifications for business programs. Therefore, you could also say that that is what improves productivity. In short, the 4GL operation base becomes the common main routine and takes over operability-related processing, thereby improving productivity.

While 4GLs have the above-mentioned merit, they also are plagued by the following structural problem. Specifically, it is often not easy to make slight changes to the operation characteristics of a business program whenever you choose to use a 4GL.

3.2.2-m Two Reasons 4GLs Have Not Gone Mainstream

You would think that 4GLs would be more widely used and would have gone mainstream, but that is not necessarily the case. Let’s look into the reasons why.

One problem frequently mentioned by the users of commercially available 4GLs is their frustrating lack of freedom. Any attempt to stray even a little bit from what is covered by a 4GL will end up in a complete denial of support. That is why people often get fed up with 4GLs. The following paragraph details what one developer would say.

“While testing a business program we had developed using a 4GL, the customer requested a slight change in operation characteristics. Since such a change could not be made with a 4GL, we explained that it would be impossible. However, the customer would not understand and we ultimately could not avoid fulfilling their request. We ended up having to put in weekend hours to develop an approximately

4,000-line-long program in a hurry. If we had used COBOL from the start, we could have avoided that fate."

This typical problem with 4GLs is also sometimes expressed, as "Form styles are limited to those built-into 4GL." Form patterns, layouts, and operation characteristics are included in form style, and more importantly, operation characteristics are fixed and cannot be easily changed. This can also be thought of as a good thing because it results in standardized operation characteristics. However, almost all programs developed in the U.S. and Europe using 4GLs are incapable of implementing the finely tuned operation characteristics that are encouraged in Japan, and therefore, there are many Japanese who are not comfortable with the operation characteristics of business programs developed using a 4GL.

The reason why operation characteristics cannot be easily changed in 4GLs is that the procedures related to operation characteristics (i.e. the 4GL operation base) is built into the program code generator in the form of wired logic. Changing this would require program customization of the 4GL itself. However, it is safe to say that 4GL vendors would never respond to such change requests in a timely manner.

Another major problem is a lack of standardization resulting in specifications differing with each company. You would think that at least the specifications for the language itself would be standardized, but that is not the case. As in the story of the Tower of Babel, the fragmentation of language creates all sorts of problems. When using a 4GL, you must select one language and learn its new specifications. There are many people who are ambivalent towards the non-standard language specifications of 4GLs even though they would like to use a commercially available version. No one wants to use a confused language up to the point of suffering the punishment of the Tower of Babel.

As you have seen thus far, the crucial point about 4GLs is the operation base that carries out operability-related processing. Consequently, we can think of the specification differences resulting from vendor competition to make 4GL operation bases even better as a necessary evil. However, there is no need for each company to have its own language specification for writing such a simple thing as an **IF** statement. To the suspicious eye, it would seem that vendors are establishing proprietary specifications in order to enclose users so that they cannot escape.

The open movement is making waves in the computer world, and we have entered a period in which vendors cannot do things simply because they are convenient for their business. Therefore, a continued focus on proprietary language specifications will be the demise of 4GLs. An effort is required to make 4GLs open along the stream of open movement. The pursuit of such openness should lead to the conclusion that we only need to create common main routines (a framework in the narrow sense) that carry out processing related to operation characteristics on top of standardized language specifications.

A blunt opinion concerning this matter is that the name 4GL is no good because it gives the impression that it has superior language specifications. I think 4GL development should cease and we should instead quickly replace it with "common main routines" (a framework in the narrow sense). Even if this were to happen, the name 4GL would remain in history as a former cutting-edge technology producing substantial results in light of the experiences in business program development in the business field. Consequently, I hope that the old name will be decisively abandoned and replaced by "common main routine" which is open and easy to understand. Such a frontier spirit is exactly what must be respected.

3.2.2-n 4GL and Fill-In Systems

Now I would like the reader to recall the previously mentioned fill-in systems. 4GLs and fill-in systems that employ an event-driven system are very similar even without a detailed comparison. A comparison of

their call-related specifications reveals a 4GL operation base corresponds to a skeleton routine, and an event procedure corresponds to a supplementary routine unit.

The most important similarity is that either one is a scheme for reusing common main routines (frameworks in the narrow sense). Any program written in a 4GL that has adopted an event-driven system treats the 4GL operation base as main routines. Therefore, this sort of 4GL can be seen as a scheme for reusing the common main routines known as a 4GL operation base. The same goes for a fill-in system. As we have already said, a fill-in system perceives the skeleton routine as the common main routines (common component), and by reusing it productivity can be improved. In short, a fill-in system is nothing but a scheme for reusing common main routines.

If you look at the role and function fulfilled by a common main routine, you will notice that a 4GL operation base mainly carries out processing related to operation characteristics. If you look into what is widely and commonly used as a skeleton routine in a fill-in system, you will find that as you might expect, they are mostly programs that carry out processing related to operation characteristics. In short, 4GLs and fill-in systems are similar in terms of their role and function.

Note that there are a number of other common main routines besides those that carry out processing related to operation characteristics. For example, processing that calls event procedures and supplementary routine units is without a doubt a role of a common main routine. There are also those that carry out database access. Those that are called database tools alleviate the need to create database access programs manually as long as a certain form of usage is followed.

However, visual development support tools for GUI application programs normally adopt an event-driven system. An examination of this matter reveals that a GUI operation base fulfills a role similar to a 4GL operation base.

Consequently, the following three programs are “common main routines,” and we can say that their main role is processing related to operation.

- Operation base related to operation specification within SSS
- 4GL operation base
- GUI operation base

3.2.3 From SSS to RRR Family

During the investigation of an RSCA that was carried out before the development of the RRR family, we reconsidered the fill-in system SSS and reevaluated where it was superior. We also compared the event-driven systems of 4GLs and visual development support tools and used the best aspects of each. This will be described in-depth hereafter.

3.2.3-o SSS as a Fill-In System

We have already discussed how most people end up heading in the wrong direction and failing at the two branches found in fill-in systems. We also mentioned how heading in the right direction should produce some degree of success. The fill-in system known as SSS fortunately was able to head in the right direction, and it was also able to rack up a huge success, not just a meager one. Let’s take a look at what went right.

In research that Woodland Corporation conducted to develop SSS, the company, as a dealer of office computers which had shipped 10,000 business systems to customers, used the systems as research material.

Specifically, they carried out their research using the sales management system with the best proven track record as their subject. Let's discuss the circumstances of the research and the ideas and so on that were produced therein. First, I would like to restate the following paragraph contained in **1.3 "Business Packages with Special Customization Facilities,"** and then continue the discussion from there.

If they could standardize program portions related to common specifications and then create only the program portions related to each company's variety of standards, development work for custom business programs could be rationalized. However, this is easier said than done. There is no clear demarcation between common portions and company-specific portions, and such demarcation is by no means easy to create. (To use the latest terminology, such demarcation is the compartmentalization of frozen areas and hot spots.) Consequently, the ability to cope no matter the location of company-specific portions was necessary, and they had to rethink the correspondence between programs and specifications from the bottom up.

During this rethinking, they performed an in-depth study of a sales management system that actually operated rather than investigating component synthesis tools. They demarcated and partitioned the program, turning it into components. This allowed them to head in the right direction of emphasizing components themselves rather than tools at the first branch of the fill-in system.

This book applies the general term **reuse system of componentized applications** (RSCA) to the component-based reuse system represented by the SSS and RRR products family. This term includes the meaning of a system that reuses componentized applications. If you were asked what a componentized application is, you could say it was an application program that was demarcated and split into pieces. In other words, it means an application program that was broken down into little pieces (i.e. componentized). The important thing here is componentized applications are easy to reuse if the splitting up of components was done skillfully. This is the origin of the term "RSCA," reuse systems of componentized applications.

The idea of **demarcation** arose within the rethinking of the correspondence between programs and specifications that was carried out in 1988. That was an idea that sprung from the desire to demarcate between programs related to business specifications and programs related to operation specifications. If it were possible to do this, the outlook for research on SSS development could certainly be expected to be brighter.

Since operation specifications are those related to the operation characteristics of a business program, it should be fine to apply the same ones for both Company A and Company B. Therefore, it should be possible to use the same program which is used for operation specifications. In contrast, business specifications have company-specific portions as well as common portions. This means that programs related to business specifications are what require customization.

Putting all this together leads to the conviction that portions with the highest potential for customization when such demarcation is possible can be narrowed down to only a program of a certain scope.

However, if you look at actual programs, you will see that those related to business specifications and those related to operation specifications are all scrambled up. This means that separating them requires major surgery (**refactoring** in the current vernacular). Such surgery uses techniques, such as shaping, sorting, and generalization. Since it is something like making numerous changes to the chapter structure of this book to make it even a bit more lucid, talking about the details therein would be too much information and not be very interesting, so we have elected to omit it. In any case, once you look at the post operative-state, you will see that the program related to operation specifications will be a skeleton routine and the program related to business specifications will be a supplementary routine unit.

Since a skeleton routine is a program related to operation, it corresponds to components that we should be able to make common. Consequently, SSS just happened to head in the right direction at the second branch in the fill-in system where either a skeleton routine or a supplementary routine unit is viewed as a common component.

In this manner, SSS, which was one kind of fill-in system, provided a scheme for reusing **common main routines**. In other words, as with 4GLs that adopted event-driven systems, the reuse of main routines that carry out processing related to operation characteristics became something that improved productivity. Furthermore, portions with high customization potential could be narrowed down to only programs related to business specifications written in supplementary routine units.

3.2.3-p Importance of Partitioning Guidelines for Compartmentalization of Components

Amid the emphasis on the need for a component lineup where you can "find the component you want most of the time," this book had discussed the importance of systematizing components to a certain extent, developing the necessary components accordingly, and then maintaining them so they can be used at any time. In other words, haphazardly saying that this or that is a component without any thought will make it impossible to figure out how to perceive a component and lead to nothing but confusion. It will not be possible to know whether a component you want exists, and for most people, such components will not be easy to use. Making components usable in a methodical manner requires the establishment of strong partitioning guidelines for the compartmentalization of components, systemization of components to a certain extent according to those guidelines, and development of the necessary components according to that system.

In relation to this, I would like to make clear in this book my understanding of how component-based reuse systems are perceived. For example, when designing a component-based reuse system that is able to synthesize components from a sales management application program, this book takes the point of view that what will be the source of those components must exist within the program that actually runs.

Care must be taken here not to misunderstand. Since application programs for sales management that actually run may be in a spaghetti state, even considerable demarcation to produce compartmentalized blocks will not produce anything that could be called a component. Therefore, we can say without a doubt that surgery (refactoring) is necessary to organize the internal organs through techniques, such as crafting, preparation, shaping, sorting, and generalization rather than mere demarcation and partitioning. If such surgery on a sales management application program is done skillfully, it will result in clean partitions, and each partition will be a component itself or the source of a component. In short, we are thinking about what 'Business Logic Components' are. Additionally, the reasons for using such vague terms, such as component itself or source of a component are these; additional components may be required, functionality will need to be generalized, and some sort of adjustment processing may be required during component synthesis. Basically, as long as the surgery goes well, each partitioned block will become a component itself, and each of these blocks is what should become a "Business Logic Component."

What this boils down to is the idea that skillful surgery on application programs for sales management that results in clean partitioning will produce componentized applications for sales management, and this will make it easy to customize them into a sales management system for Company A and another for Company B. In short, reuse will become easier. SSS is nothing more than an actual example of this and it backs up this idea.

At the same time, skillful surgery on an application program for production management to create clean partitioning will result in componentized applications for production management, and this will make it

easy to customize them into a production management system for Company A and another for Company B. In short, reuse will become easier.

This is the idea of my book, but how about the perceptions of other component-based reuse systems? For example, there may be other forms different from the previously discussed failures, such as emphasizing component synthesis or emphasizing component retrieval. Such pursuit may have some sort of end result, but it is difficult to predict. Due to this unpredictability, this book ignores these systems and takes the standpoint that there is no other component-based reuse system other than RSCA. The fact that this concept practically has an effect has been proven by the actual example of SSS.

We could sum this up by saying the perception in this book at least has meaning, and based on this we would not be exaggerating by saying that strong partitioning guidelines for the compartmentalization of components are the crucial key for influencing whether a component-based reuse system is or is not possible.

Accordingly, let's evaluate SSS from the viewpoint of partitioning guidelines for the compartmentalization of components.

The SSS component set is partitioned into individual 'Business Logic Components' based on two partitioning guidelines. The first partitioning guideline is the demarcation of business and operation discussed previously in "**SSS as a Fill-In System**." The other partitioning guideline is partitioning with data item association as discussed in **1.3 "Business Packages with Special Customization Facilities."** In short, these are partitioning guidelines for partitioning by the correspondence of supplementary routine units related to business specifications to data items.

The demarcation of business and operation, the first partitioning guideline, is not unique to the SSS component set but rather is also employed by other tools such as 4GLs. You could therefore say the effect of this partitioning guideline is widely recognized. Based on this guideline, common main routines can be reused and productivity improved, and portions with high customization potential can be narrowed down to a certain range of programs.

Partitioning with data item association, the other partitioning guideline, cleverly perceives the characteristic of specification change requests being represented by data item names in the business field. In **1.3 "Business Packages with Special Customization Facilities,"** this book already stressed the fact that this partitioning guideline has opened the way for building a practical and effective component-based reuse system.

Now I would like to restate the most important points. As for data item components partitioned according to these partitioning guidelines, their correspondence with business specifications becomes clear, and that alone results in meaningful closed blocks, most of which are one hundred lines or less. In addition, they are stereotypically easy to decipher. Furthermore, using names that start with the respective data item names results in easy-to-retrieve blocks. All this has allowed the portioning of application programs into blocks known as components, or more precisely, data item components. This was more than just meager success.

Visual development support tools also seem to comply with the partitioning guidelines of partitioning with data item association at first glance, but these tools do not comply with the guideline perfectly. The scope of the study, we could not find tools other than the SSS component set which comply with the guidelines completely, perfectly and faithfully. Since this point is crucial, we will describe it in even more detail a little later in **3.2.3-s "Second Improvement of Partitioning Guidelines for Compartmentalization of Components."**

Incidentally, in relation to partitioning guidelines for the compartmentalization of components, a look at Smalltalk systems, recognized as making reuse easy, reveals that they have adopted the guideline of three-way partitioning known as MVC (model view controller) for deciding the configuration of objects that form a hierarchy. Furthermore, the importance of this partitioning guideline is often cited. This is exactly the sort of partitioning guideline that is substantial.

3.2.3-q Improvements for RRR Family

Full-scale development of the RRR family as an RSCA that was positioned as the successor of SSS began in 1995.

After some retrospection, since SSS never differentiated between component sets and component synthesis tools, we decided that the RRR family clearly needed to separate them. This book distinguishes between the two by calling the former the RRR component set and the latter RRR tools.

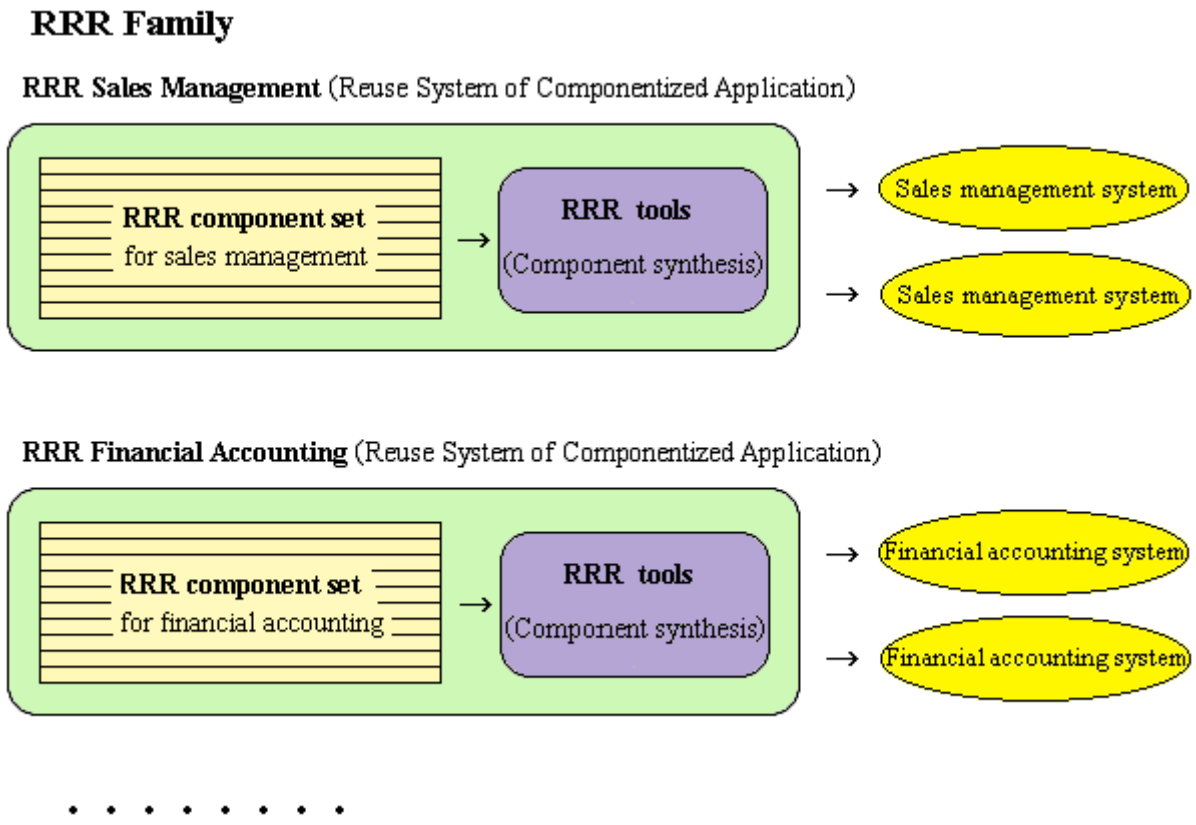


Figure 3-2: RRR Family Components and Component Synthesis

As shown in **Figure 3-2**, the products belonging to the RRR family, such as RRR Sales Management, are component-based reuse systems comprised of one RRR component set (in this case the RRR component set for sales management) and RRR tools. A RRR component set is a group of components required to construct a specific business program. RRR Sales Management employs the RRR component set for sales management and RRR Financial Accounting employs the RRR component set for financial accounting. In addition, the individual components in a RRR component set are a typical example of the 'Business Logic Components' discussed in this book.

When developing the RRR family, we compared the fill-in system SSS with event-driven systems like 4GLs and visual development support tools, and then adopted the best parts of each. The following paragraphs will describe the two main results, namely a call mechanism for components and partitioning guidelines for the compartmentalization of components, which were obtained from that comparison.

As will be described below, the call mechanism for components is better in event-driven system tools than in fill-in systems.

With a fill-in system, you generally have to modify part of the skeleton routine depending on the number of supplementary routine units that must be put into it. In short, you must delete a slot (or fill it in with a dummy unit) when there is no unit to put into it, and when there are many units that must be put into the skeleton routine, you must increase the number of slots. However, event-driven systems never require such work.

Consequently, it goes without saying that we adopted a more modern event-driven system instead of a traditional fill-in system. Note that the parameters during the calling of event procedures were fixed in visual development support tools, but we added improvements to this to raise the degree of freedom in RRR tools.

3.2.3-r First Improvement of Partitioning Guidelines for Compartmentalization of Components

If you take a look at the partitioning guidelines for compartmentalization of components, you will see fill-in systems and tools that have adopted an event-driven system to employ the following two partitioning guidelines as a general rule. The first is demarcation of business and operation, and the second is partitioning of data item correspondence (object correspondence). However, regarding how faithfully partitioning guidelines should be followed; fill-in systems and tools are far from the ideal of RSCAs.

No visual development support tools faithfully follow the partitioning guideline for demarcation of business and operation. In visual development support tools, the demarcation of business and operation becomes quite vague, and they prefer to prioritize enabling detailed control of hardware operation.

For example, typical events in a visual development support tool are a mouse click on a GUI control or a key press on a keyboard. Rather than saying these are the demarcation of business and operation, it is more appropriate to say that they are event architectures that are convenient for controlling hardware devices from application programs as intended. Specifically, to enable different processing when a keyboard key is pressed and when it is released, there is a detailed event architecture that can differentiate between them, and events are made to occur frequently each time a character is entered. In such low-level event architecture, if you tried to create an application program focused on the handling of text input using the keyboard, you would end up having to mix and write operability-related programs containing more lines than the business specification-related program written within an event procedure. This does not at all follow the partitioning guidelines of demarcation of business and operation.

However, with even such low-level event architecture, it is possible to separate programs to a certain extent depending on how they are viewed. Take for example the development of an application program for easy-to-use GUI operation. Almost all of the operability-related processing can be left to the GUI operation base, so all that has to be written is the program related to business, mainly event procedures. Visual development support tools generally provide functionality that targets hobby programmers who account for the majority of their users, and they make money by the number of units sold. Using such functionality aimed at hobby programmers makes it extremely easy to create a toy-like program that produces, for

example, a pigeon when a button is pushed. Consequently, we can see that the partitioning guideline of the demarcation of business and operation is being observed for this type of simple program.

However, for programs that perform input processing from a keyboard, as is often found in the business field, the partitioning guideline of the demarcation of business and operation as discussed above is almost totally ignored. For example, you might think that being able to move the cursor with the arrow keys is a very basic and simple operation, but doing so requires the writing of a program related to arrow key operation as an event procedure that must describe business processing.

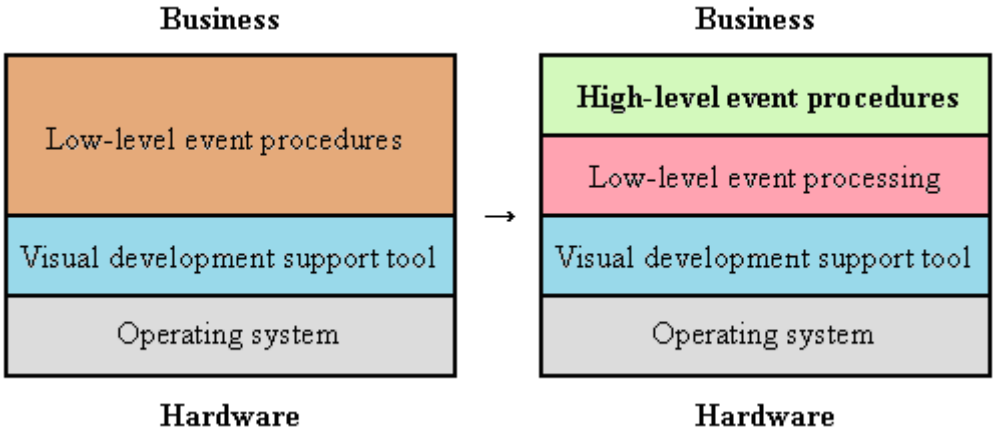


Figure 3-3: Low-Level Events and High-Level Events

Since we wanted to build the RRR family on top of a visual development support tool, we built an event architecture named high-level event on top of a low-level event architecture as shown in **Figure 3-3**. Furthermore, we made the high-level event architecture faithfully follow the partitioning guideline for demarcation of business and operation.

While visual development support tools are made to cause an event when even a single character is entered for an item on a form, with high-level event architecture we defined an event (high-level event) as the timing at which the entry of data for an item on a form should be checked. Doing so eliminates the need for a program that decides whether a check of data is necessary in any case. You might think that this is just a slight improvement, but if you total up all the programs that make such decisions, the number of lines in such operability-related programs (framework in the narrow sense) will surpass those in business processing-related programs. Consequently, there is a major difference in deciding on a high-level event architecture that is convenient for business processing rather than an event architecture that is convenient for the detailed operation of hardware devices. Using high-level event architecture eliminates the need for programming portions unrelated to business processing, reduces the number of program lines for high-level event procedures, and results in easy-to-understand programs.

The bottom line is low-level event architecture tends to intermix procedures related to operation characteristics with programs related to business. On the other hand, high-level event architecture has interfaces for programs related to business, and they eliminate the need to write procedures related to operation characteristics. That is why the number of program lines for portions related to business decrease.

The reason for such an effect can also be interpreted as follows. Since there is a major gap between the low-level event architecture close to hardware devices and the events that appear in the business scene, a large quantity of programs are required to fill the space between them. Accordingly, providing high-level event architecture near the events that appear in the business scene, instead of low-level event architecture,

reduces the number of program lines to those necessary for a small gap, and thereby results in easy-to-understand programs.

Although high-level event architecture can produce such an effect, on the other hand it makes the detailed control of hardware devices difficult. Most business programs in the business field need not worry about that, but there is a related problem. The problem is that hardwiring a program, which performs detailed hardware device control, ends up hardwiring operation characteristics. This is nothing other than the problem that was expressed as a "frustrating lack of freedom" in general 4GLs. Note that we will discuss how we confronted this problem in **"Topic 6: Tools for a Componentized Event-Driven System."**

One other troubling point about high-level event architecture seems its lack of ability to deal with events that appear in actual business scenes. Since low-level event architecture is comprised of detailed events, we know from previous experience that any business scene can be dealt with by assembling such events. On the other hand, high-level event architecture is not all that detailed, which leaves us wondering whether it can deal with business scenes. However, up to this point (about three years following the writing of the initial versions and then seven years after the following revisions), the combination of high-level event procedures has been able to deal with a variety of business scenes with no problems. We cannot say there is no chance of encountering situations that we cannot handle, but if we cannot, we also have the option of adding a new high-level event to the current event architecture. Consequently, there is no problem in terms of the ability to deal with the business scene. This is also backed up by the fact that this sort of problem related to handling ability does not occur in 4GLs that have adopted architecture close to high-level event architecture.

Our discussion of the partitioning guideline of demarcation of business and operation ended up being quite long, so we would like to wrap it up here. Visual development support tools do not faithfully obey these partitioning guidelines. Accordingly, for the RRR family we created high-level event architecture on top of the low-level event architecture of a visual development support tool and it was designed to faithfully follow the partitioning guidelines of demarcation of business and operation.

3.2.3-s Second Improvement of Partitioning Guidelines for Compartmentalization of Components

If you take a look at the tools that have adopted an event-driven system and partitioning guidelines for the compartmentalization of components in a fill-in system, you will find that they have adopted a guideline close to the demarcation of business and operation, as well as the partitioning guidelines of data item correspondence (object correspondence). However, regarding how faithfully partitioning guidelines were followed, there was a wide gap between the tools and the ideal of RSCAs.

At first glance, 4GLs and visual development support tools seem to follow the partitioning guideline with data item association. Specifically, when an event related to a certain data item occurs, the event procedure corresponding to the event classification of that data item will run. As a result, it would seem that one event procedure was being created for each data item. Strictly speaking, one event procedure is created for each data item and event classification combination, but if you think of event procedures related to the same data item as belonging to the same group, then partitioning is truly being carried out for each data item.

However, a vital piece was missing. An event that is crucial to the formation of data item components, i.e. an event related to update propagation, was missing. Because of this missing event, the compartmentalization per event procedure data item became blurred.

As a specific example of this, think of using a visual development support tool to develop a program that supports the entry of orders received data for a product. Specifically, we are talking about the creation of a business program that has a form similar to the one in **Figure 3-4**. Entering data into the data item named "product code" in this form will run an event procedure for "product code." Within this event procedure, it is common to not only check whether the entered "product code" is in the product master file, but also perform display processing for "product name" and "product unit price." The problem here is other data item processing, such as "product name" and "product unit price" also end up being carried out within the event procedure for "product code." This means that the demarcation between data items has become blurred.

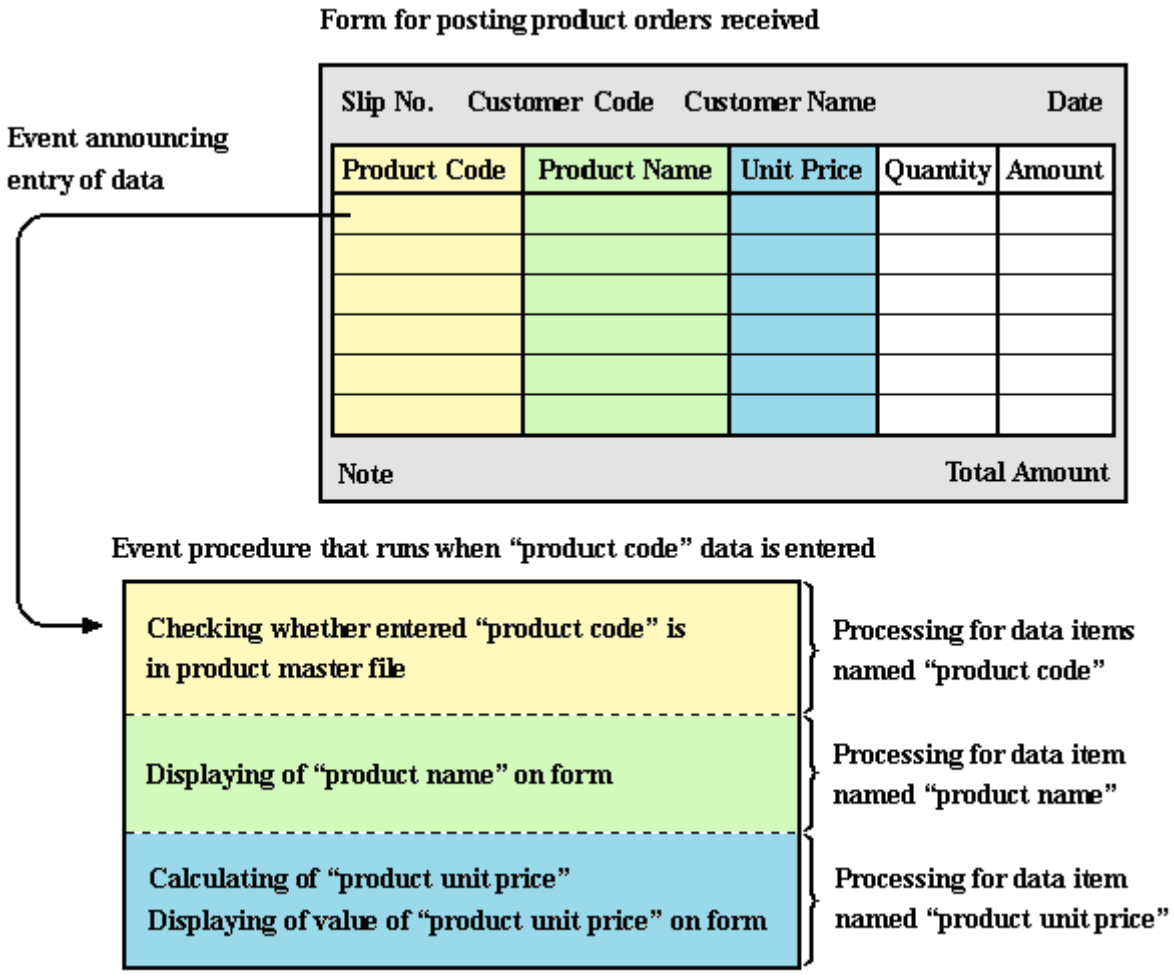


Figure 3-4: Business Program Using a Visual Development Support Tool

There are many inconveniences in such a program. For example, carrying out customization to discount "product unit price" by twenty percent during a product sales campaign for a certain period of time requires changing the event procedure for the "product code." Since this is customization for the "product unit price," we would like to change the event procedure for that unit price, but that is not how it works. Furthermore, since this event procedure can only be used for forms in which "product code," "product name," and "product unit price" appear, the situations in which it can be reused are limited. If possible, we would like to reuse it in forms in which "product code" and "product name" appear as well as in forms in which "product code," "product name," "product unit price," and "product expiration date" appear, but that is not how it works.

Such inconveniences are caused by the blurring of demarcation between data items. The solution is to clearly define the demarcation between data items. This means processing for data items, other than the checking of whether an entered "product code" is in the product master file, must not be carried out within the event procedure for the "product code." The display processing for "product name" and "product unit price" should be performed within separate event procedures for the "product name" and for the "product unit price."

This requires a mechanism for starting an event procedure as a trigger for changing the value of other data items. In short, we need a mechanism for update propagation. To implement this mechanism we decided to create event architecture named high-level event on top of the low-level event architecture of a visual development support tool as shown in **Figure 3-3**. We also added a special high-level event within the high-level event architecture to enable the use of an update propagation mechanism.

As a result, we were able to compartmentalize data item components with good demarcation between data items just like with SSS components. This not only resolved all of the previously mentioned inconveniences, it also lightened the load of development as follows. Whereas SSS required the handwriting of programs for carrying out update propagation processing, RRR enabled them to be automatically generated.

Note that in the automation of this update propagation processing, we employed directed graph theory for one field of mathematics and adopted a system that does not produce unnecessary calls (width first system).

Since event procedures in 4GLs and visual development support tools do not faithfully follow the partitioning guideline of partitioning with data item association, they unfortunately cannot be called components. However, RRR component sets and other methods that faithfully partition event procedures in accordance with data items increase cohesiveness and enable reuse in a variety of situations. They produce component value, and such components can be called 'Business Logic Components.' We will call this system that improves upon general event-driven systems and enables event procedures to have value as components a componentized event-driven system. Note that we applied for a patent of this componentized event-driven system.

An evaluation of the event architecture of 4GLs and visual development support tools will reveal that the creation of event procedures has not become any easier, most likely because their first priority is enabling the creation of any sort of program. In contrast, when designing the event architecture of the componentized event-driven system, we placed an emphasis on the easy creation of event procedures and ease of customization. This required a mechanism for update propagation.

Topic 6: Tools for a Componentized Event-Driven System

RRR tool development was divided between Woodland Corporation and AppliTech Inc. Specifically, AppliTech Inc. handled the adding on of functionality to a visual development support tool to transform it as if it were a componentized event-driven system, and Woodland Corporation handled everything else, which included database access-related work.

In the following paragraphs, we will introduce **MANDALA**; an add-on tool developed by AppliTech Inc. for the visual development support tool Visual Basic of Microsoft Corporation.

MANDALA, positioned at the core of RRR tools, adopted the previously mentioned componentized event-driven system. In addition, making it an add-on tool for Visual Basic, which is a sort of industry standard, lightened the development load of **MANDALA** itself. If we had developed all the functionality required by the tool ourselves, development speed would have decreased and we would have not been able to quickly respond to environmental changes. There have been many cases of the weight of developing tools for improving productivity bogging down development projects, resulting in the ladder being pulled from under users and causing tremendous trouble. To keep such things from happening, we decided to develop only the functionality lacking in a visual development support tool by creating an easy-to-develop add-on tool.

We paid attention to the following four points in the design of **MANDALA**:

- Incorporating all 4GL functionality not included in the visual development support tool.
- Creating a structure that could easily deal with customization requests for operation characteristics.
- Enabling not only attractive operation characteristics via a GUI, but also conventional keyboard operation.
- Employing standard language specifications, i.e. not creating local language specifications.

By paying attention to these points, we were aiming for the ultimate open tool by incorporating into **MANDALA** all functionality considered to be useful in 4GLs. For example, a form application we generated by this tool could be made multifunctional so that it not only could add data to a database, but also carry out processing for viewing, updating, and deleting. As for the problem of the "frustrating lack of freedom" for which 4GLs were despised, we responded by enabling parameter customization for **MANDALA**. However, there were some cases that even this could not handle. Accordingly, we made program customization for **MANDALA** itself easy and also offered timely support service for developer requests. A "Royal Toolmaker Service" is one AppliTech service for the program customization of **MANDALA** itself, and we have already offered it for a fee to over ten customers. Note that the average tool vendor does not aggressively carry out tool customization service because they prefer business that does not require constant monitoring just like business package development firms, as discussed in Chapter 1. However, AppliTech Inc. is carrying out this service with information-gathering tactics in mind for customization requests that will enable the adaptation of **MANDALA** to a wide variety of environments.

We also developed **MANDALA** as a program generation tool in consideration of such things as the performance of business programs developed by it. Generally, program generation tools can be classified as a pre-generator (software tool) or a post-generator (software tool). Since a pre-generator carries out generation processing ahead of time, developers must make changes to the generation results. A post-generator, however, analyzes developer-written programs and then carries out generation suited to them. Generation tools are frequently said to have many problems, but such problems lie in pre-generators.

MANDALA was made into a post-generator that does not have such problems because it can change and regenerate business programs with a high degree of freedom.

For reference purposes, we have listed the main high-level events used by business programs developed by means of **MANDALA**. The name of each high-level event represents what sort of processing should be carried out in each high-level event procedure. I would like the reader to focus on the high-level event "derived value" therein. This high-level event is related to update propagation.

- Input check (Check): Checks the data entered for the data item.
- Derived value (Derived): Sets the value of the data item upon being triggered by the entry of data in another data item.
- Initial value (InitVal): Sets the initial value of the data item at the completion of each data registration process.
- Guidance display (Prompt): Issues prompt messages for the data item.
- Selection list (SList): Displays a list of possible data entries (selection list) for the data item.

Also for reference purposes, we have listed the main events (low-level events) that can be used in Visual Basic. The name of each event represents what will trigger the calling of each event procedure.

I would like the reader to focus on the difference between low-level and high-level events. In contrast to low-level events that are hardware-dependant, high-level events are directly tied to the business logic of a business program.

- Change: The changing of the objects content is the trigger.
- Click: The clicking of the object is the trigger.
- GotFocus: The moving of the cursor to the object is the trigger.
- KeyPress: The pressing of a keyboard key when the cursor is on the object is the trigger.
- LostFocus: The removal of the cursor from the object is the trigger.
- MouseUp: The releasing of the mouse button when the cursor is on the object is the trigger.

Note that object here means a form or a field (data item) in a form.

CHAPTER 4 Software Development Productivity

Generally, people who have no experience developing software tend to think that the development of software is the same as the manufacture of goods, but this produces tremendous misunderstandings. It is also wrong to think of software development productivity as something that can be easily measured in numerical terms. Furthermore, discussing software development productivity without actually getting involved with software development cannot be expected to yield useful results. It would merely lead to a discussion that makes no sense and leave the numerical terms of productivity standing alone and out of context. It would be utterly fruitless.

This chapter will start with a rather thorough analysis of what software development is and then try to examine the proper method of measuring its productivity. After that, we will reflect back on the history of software development thus far to see whether productivity is improving, and finally, we will explore some methods for improving productivity.

In this chapter, the text seems to have become just a bit redundant, probably due to dealing with bloat within programs, i.e. redundant portions. Readers who feel that the content of this chapter is common knowledge should probably just skim through and read only those parts that interest them. In this chapter, we have written about matters that back up the content of Chapters 3 and 5, and therefore, if you agree with what has been written here, the purport of this book will be conveyed even if you skip this chapter. However, readers who are of the impression that the content of this chapter is not common knowledge probably should take the time to read it carefully.

4.1 What is Software Development Productivity?

Even if people in the position of managing software development think they would like to play some part in software development productivity, they do not necessarily have to know the details of development work. However, they do at least have to properly recognize what sort of work software development is. This section will start by looking at what sort of work software development is and then go on to deepen understanding about its productivity.

4.1-a Software Development is Design Work

It is said that software productivity is not improving in proportion to the dramatic improvements being made in hardware productivity. The meaning of this sentence is intended to rouse software engineers to action. This statement in itself should be accepted without protest, but we could interpret its meaning as noted hereafter.

We can consider productivity related to the manufacture of hardware (goods) as what is dramatically improving, this it stands to reason that productivity is not improving when human beings carry out design work of software development. Consequently, we can understand the above statement as saying, “The productivity of software design is not improving in proportion to the dramatic improvements being made in the productivity of hardware (goods) manufacturing.”

To begin with, the expressions hardware productivity and software productivity are vague. To clearly define their meaning, we must establish the separate categories of “design productivity” and “manufacturing productivity” for both hardware and software. In other words, we must make comparisons

between hardware design, hardware manufacturing, software design, and software manufacturing. By doing so, we will recognize, if anything, what they have in common based on essential differences between design and manufacturing and the slim difference between whether the target is hardware or software. For example, if you were to compare support tools, you would find they correspond to each other, such as CAD (computer aided design) for supporting hardware design being equivalent to various types of editors and compilers that support software design. Similarly a hardware simulator is equivalent to a software debugger. Once you realize this, you will see the real meaning of "the productivity of design is not improving in proportion to the dramatic improvements being made in the productivity of manufacturing," whether you are talking about software or hardware.

To deepen understanding in this area, let's start by clearly defining the meaning of the words "design" and "manufacture."

"**Manufacture**" assumes that design has already been completed and means "the production of a physical object" that was clearly defined through design work. The manufacture of a car, computer hardware, a structure, or a printer is the production of some sort of object clearly defined by a design drawing, while the manufacture of software or a publication is nothing but the production of a floppy disk, CD-ROM, or bundle of bound paper containing a copy of the original.

"**Design**" is something that clearly defines what is to be manufactured, i.e. it completely eliminates vagueness about what is to be made. The act of designing dispels any doubts about what will be produced at the manufacturing stage and leaves no room for creativity. Another way of saying this would be to say "design" is the preparation of everything so that not even one ounce of creativity will be necessary for what is to be manufactured (in this place manufacturing method details are a separate matter).

If you take a novel as an example, "design" is work wherein you develop an idea, collect material, write, polish, and then work out the binding. On the other hand, "manufacturing" is work wherein you set type to produce the original plate, run the printing press, and then bind the books.

Note that we distinguished between the work of clarifying requested specifications and the work of implementing them in Chapter 3, but the "manufacturing" process is what follows both of them. Furthermore, these two types of work can be perceived as "design" in the broad sense. If you want to distinguish between them, the clarification of requested specifications is what you do before "design" in the narrow sense and you can perceive the implementation (coming up with implemented figure) of requested specifications as being equivalent to "design" in the narrow sense.

Software development includes such work as planning/analysis, general design, programming, and testing whether it is based on a waterfall model or a spiral model. Since we refer to programming within this work as manufacturing, misunderstandings arise, but if you think about it a bit, programming is definitely not manufacturing. The manufacturing process is the copying/burning of data onto floppy disks or CD-ROMs (software products) after what is to be manufactured is clarified. Since programming is nothing but work that clarifies what is to be manufactured (software), it is included in the design process. Thinking of it as being equivalent to work wherein you draw up design drawings would be appropriate.

To begin with, almost all software development is design work. Consequently, if we want to improve its productivity, we must come up with a plan for improving design work productivity. If you were to force the comparison of software development to a manufacturing process, even if we wanted to capitalize on the fruits of productivity improvements in manufacturing work, it would be no more than a mere wish. As discussed later in **4.3 "Is Software Development Productivity Improving?"** it is difficult to apply a plan for improving manufacturing productivity to design work. You are free to compare software development

to a manufacturing process, but we cannot recommend doing so because it merely breeds misunderstandings and fosters illusions.

4.1-b How to Measure Software Development Productivity?

Productivity is the “quantity of products” divided by the “total work hours” spent in manufacturing or developing products and it means the quantity of products per unit work hours. The value of productivity increases when you raise the **quantity of products** or decrease **total work hours** compared with the former production method, and productivity is determined by the ratio of these two items (quantity of products and total work hours).

Productivity = Quantity of products/total work hours

First of all, we count only the hours directly spent by workers for the measurement method for **total work hours**. In the case of software development, the greater part of development cost is labor, so even if we only count that, there probably will not be a big error.

However, in the case of the process industry, which requires large investment in facilities, we must consider the cost of equipment. In short, to enable more precision, we should include the work hours that workers spent to develop and manufacture equipment. This is a reasonable way of thinking, but we are dealing with software development productivity here. Therefore, in order to simplify the discussion, we will calculate productivity using only the total work hours directly spent by work in development and manufacturing, based on the idea that the previously mentioned facilities are infrastructure for developing and manufacturing.

Total work hours can be easily dealt with in this manner, but how to measure the **quantity of products** is a difficult problem. How to measure the quantity of products in design work, like software development, is particularly difficult.

If you look into how to measure products in manufacturing work, this too is by no means easy. If you want to try comparing different types of products, for example, car manufacturing and bookmaking, how to adapt the criterion for the quantity of products becomes a problem. If you attempt a comparison, the only choice seems to be a monetary quantity converting each type of work into an added value. It is generally difficult to compare the quantities of products when they are different.

For manufactured goods of the same type, for example, automobiles, you can measure (count) the quantity of products by the number of vehicles. However, there are both luxury cars that require extensive manufacturing and economy cars that are more easily manufactured. If you measure the quantity of products simply by the number of vehicles, productivity will be higher for economy vehicles than for luxury vehicles. That does not necessarily mean economy car manufacturing has better manufacturing technology for improving productivity than does luxury car manufacturing (such tendencies are sometimes seen). A fair comparison of productivity in which the quantity of products is measured in product units will only be possible with the same type of manufactured goods that are of the same quality.

In the case of things for which only one, or an extremely small number, are produced, such as high-rise buildings that have a unique design, what units should we used to represent the quantity of products is a particularly difficult problem. This means even if we compute productivity based on rough values, such as Building 1, Building 2, the data cannot be said to be meaningful because there is no way to compare it to anything else. If we assume that the total work hours for the construction of Khufu's Pyramid amounted to one million man-years, then productivity in that case would be 0.000001 building/man-years. However, this does not have numerical-term meaning for productivity; it is nothing more than the reciprocal of mere total

work hours. Data with meaning for productivity will only result through a representation that is easy to compare with something else rather than units, such as Building 1, Building 2. We need a method that measures the quantity of products using a common criterion, such as floor space or volume, even though this might ignore some aspect of a building's value in a certain sense.

Unlike with small-scale production, the comparison of productivity between the same types of manufacturing goods in the case of industrial goods that are mass-produced is easy. If you measure (count) the quantity of products in units, such as Item 1, Item 2 and then compute productivity, you will end up with data with sufficient meaning, at least for those manufactured goods. For example, productivity data indicating that a certain automobile can be manufactured at a rate of 0.3 vehicles/man-day would serve as an indicator as to whether changing the manufacturing method of that automobile would be effective in improving productivity. This means such data would be a barometer for evaluating manufacturing technology for the automobile.

In this manner, productivity representing the quantity of products in a number of units (count) has sufficient meaning in the case of repetitive work. However, in the case of small-lot production or products of different types or qualities, productivity can only be compared if we find a common criterion suitable for representing the quantity of products.

We have thus far looked into how to measure products in manufacturing work, but similar things can be said about the quantity of products in design work such as software development.

Since software has normally been developed on an individual basis (please note that this book does not recommend this, but this is the way it is), the problems involving small-scale production resemble high-rise buildings that have a unique design. Also, there are some software products that take this much effort to be developed and some that don't. Consequently, there is no way to make comparisons with anything other than productivity computed by the rough measurement (counting) of the quantity of products, such as System 1, System 2. In this business system, even if you say productivity was 0.01 system/man-month, the numeric value only means production took a total of one hundred man-months and does not have any significance in terms of productivity. Enabling comparisons of productivity among different types of development demands that we find a way to represent the quantity of products. We require a method like representing the quantity of products for a pyramid by means of floor space or volume.

There are a variety of proposals and arguments concerning this common criterion. They include many that might be seen as last resorts such as:

- With the number of program lines excluding comment statements, there is the problem that it can undergo any quantity of padding, but we have no choice but to use it, because there is no other good criterion; or
- You should use a weighted score to measure the numbers of forms, report forms, records, and data items; or
- You should measure by the revenue generated by the products.

There are many variations even among ways of measuring that are similar to the number of program lines. There are also detailed debates as to whether counting the number of statements or the number of characters in a program is more exact. This book represents this common criterion using "the number of program lines excluding comment statements," without entering into such debates.

4.1-c Minimum Information Content of a Program

We would like some sort of theoretical backing that will enable us to fully accept the method of measuring the products of software development. The well-known Shannon's theorem, which forms the foundation of information theory, clearly defines the relationship between **information content** and channel capacity, and it is the theoretical backing for removing redundancy from information. Accordingly, if we measure the quantity of products using this "information content," we should be able to get objective data that we can accept. In short, if we knew the minimum information content required to write (or transmit) a program for solving a problem of a certain scope, we could create a single definition of objective productivity by adopting that value as the quantity of products. In simple terms, this minimum information content is the program size when the program is the most compacted. In terms a bit closer to Shannon's theorem, the minimum information content is the number of bytes after the program has been skillfully compressed so as to reduce the bytes transmitted as much as possible when it is sent over a communication line. Note that we use the term **true productivity** to describe the computation of productivity that adopts the minimum information content of a program as the quantity of products.

After all this preparation, you will probably be shocked to learn that no algorithm has been discovered for determining the minimum information content of a program. Furthermore, just as no general solution exists for the well-known halting problem of the Turing machine that appears in computer theory (see **Note 8**); we noticed that there seems to be no solution for determining the minimum information content of a program. Without a general solution, we can only determine the minimum information content in a practical manner by exercising our creativity for each individual program. In other words, we make the program as small as possible by utilizing the technique of eliminating unnecessary portions, grouping common portions into common subroutines and common main routines, and then compressing the program information (see **Note 9**). This work is extremely difficult, and it will likely take so much effort that it cannot even be compared with developing a program. Furthermore, since it is difficult to declare that a program cannot be made any smaller, we end up with a never-ending job. In short, determining the minimum information content is extremely difficult.

Accordingly, it is usual to adopt the number of developed program lines excluding comment statements as the quantity of products instead of determining the minimum information content. If this value is assumed to be proportional to the minimum information content, then we can regard it as productivity that has objective meaning. The number of developed program lines excluding comment statements and minimum information content certainly can be thought of as tending to be statistically proportional. However, if you take a look at individual development cases, there are both programs that have been tightly compacted and those that are filled with **bloat** (redundant portions). If we do not somehow estimate and compensate for the degree of bloat, we will end up misevaluating development as having increased productivity without knowing we have an absurd program with a false bottom and padding.

Note 8: A Turing machine is a virtual computer employed when developing computer theory. It is so named because A. M. Turing devised it. Additionally, the A. M. Turing Award that is given by the Association for Computing Machinery (ACM) to individuals or groups who contribute to the computer field is also named after him.

The halting problem of the Turing machine refers to the problem of deciding whether a program that has been started will stop at some point or will continue running indefinitely due to complex loops. If we consider a specific program, we can sometimes come up with the answer to the halting problem. However, Turing has proven that there is no general-purpose algorithm that can come up with this answer no matter

what program is considered (in other words, we cannot create a program that produces an answer to the halting problem). This theorem in the computer science world, which states, "You cannot produce an answer to the halting problem," corresponds to K. Gödel's incompleteness theorem, and it has become an example of the existence of problems that can never be solved.

Note 9: Shannon theoretically showed the limitations of data compression as follows. If we know the probability distribution of program usage for a given set of programs, the program that has a probability of usage P, could be compressed to the following bits.

$\mathrm{Log}_2\,(1/P)$

For example, a program that's probability of usage is 1/8 could be compressed to 3 bits, and a program that's probability of usage is 1/256 could be compressed to 8 bits, that is 1 byte. You might think this value is too small from common sense. However, you could agree with this value when you think of it as a program name or program identification code. By the way, to achieve this theoretical compression result, a tacit decompression system is required. It means that a so called tacit program library is necessary to extract the original program from its name.

Here, we treat programs separately depending whether they are in the given set or not in the set. For the former programs, we apply the theoretical compression result on the assumption that we all have the tacit program library. For the latter programs, we decompress the original programs from the compressed programs without using the tacit program library.

A typical example of the latter case is when we think it is better to develop a new compression program to achieve smaller size, and do so. In such a case, we must add the minimum information content of the decompression program as the computed information contents. This is because we could not get the decompressed original program at the destination, unless the decompression program has been sent to the destination with the compressed program via a communication line. The original program must be changed into the self decompression program.

Another example of the latter case is to utilize an existing compression program such as ZIP. In such a case, we must add the minimum information content of the decompression program name or program identification code, with a size of around 3 bytes if we use the file extension. This is because we do not need to send the decompression program with the compressed program via a communication line, but instead to send the decompression program name or program identification code to indicate which one of the decompression programs to use.

If we compare this to productivity for pyramid construction, the number of developed program lines excluding comment statements will be equivalent to the volume of the pyramid as measured from the outside. However, there are probably also pyramids that are densely packed with stone, as well as those filled with an enormous amount of space or bloat, so to speak. Assuming that, representing the quantity of products using the criterion of apparent volume will not necessarily be appropriate. Pyramids containing space inside should be reconstructed into a minimum pyramid that minimizes apparent volume by filling it with stone, and then that actual volume (minimum information content) should be adopted as the quantity of products. However, since reconstructing the pyramid would take an enormous amount of work, we have no choice but to use apparent volume to measure the quantity of products. It would be nice to have a

method for easily determining the actual volume of the minimum pyramid without the need for major reconstruction, but there is no such method for programs.

4.2 Various Ways of Measuring Software Development Productivity

There is no measurement method for software development productivity that everyone accepts. Accordingly, to create one that is more widely accepted than current methods, it would seem better to make comprehensive decisions through the use of multiple measurement methods rather than relying on just one. We will therefore introduce a number of measurement methods.

4.2-d How to Compensate Productivity that is based on Number of Program Lines

We will start by describing productivity as represented by the following formula that is widely used to indicate the results of the productivity of design work (software development).

Number of developed program lines excluding comment statements/Total work hours

This book refers to values calculated by this formula as **plain productivity** without any compensation. Plain productivity is nice because it can be easily determined, but there are negative consequences in relying on it. Namely, developing a compact, densely packed program will cause productivity to be evaluated as low, while developing one that is full of space will cause productivity to be evaluated as high. Adopting such an evaluation method will impede efforts to make programs compact and encourage the use of easy ways that inflate programs with bloat. Even if you assume that it is rare for programs to be padded to raise apparent productivity with bad intent, before you realize it, we will end up getting used to thinking that bloated programs are the norm. In fact, it appears we have already come to that point.

This sort of problem occurs because of the adoption of apparent volume (number of developed program lines) as the quantity of products. To avoid this, we should adjust the quantity of products by the degree to which a program is packed rather than using plain productivity as is. In addition, we should use “a productivity value that is completely eliminated of bloat, such as true productivity.” Note that this book refers to values adjusted from plain productivity as **compensated productivity**.

As for how to compensate productivity, we think simply subtracting bloat whenever it is found from the quantity of products would be easy to understand. For example, whenever you found a space inside a pyramid, you would subtract it.

A specific example would be whenever you found unnecessary program steps you would subtract that amount from the number of developed program lines. Furthermore, regardless of whether reuse had been enabled for common subroutines registered in the library, if you found duplication that was newly added, it is bloat, and therefore, that number of lines should be subtracted from the program that was developed.

There are a variety of software resources that can be reused. As we have made clear in this book, not only common subroutines, but also common main routines related to operation characteristics and data item components related to business specifications can be reused. Consequently, the principle of thoroughly reusing resources registered to component libraries should be applied to these as well. On the other hand, if you find out those common main routines, or data item components, routines equivalent to them, have been redundantly developed, you should make an adjustment.

Compensated productivity that can be determined in this manner can be represented as follows. Incidentally, we will point out the same representation for plain productivity too. Note that the “number of program lines actually developed” indicates a value greater than the “number of program lines actually required to be developed” by only the “number of bloat lines found.”

Compensated productivity
= (Number of program lines actually developed - Number of bloat lines found)/Total work hours
= Number of program lines actually required to be developed/Total work hours

Plain productivity = Number of program lines actually developed/Total work hours

The above mentioned compensation method is simple and easy to understand, but since bloat has to be detected manually, we run into the problem that some of it might be overlooked. In short, the value of compensated productivity will end up varying depending on the sensitivity of bloat detection. There will likely be times when we carelessly fail to find bloat or will not be able to detect parts we never suspected to be bloat in the first place. For example, even if we are not told about common main routines related to operation characteristics and end up redundantly developing that part, we might not notice it is bloat. Since we can only compensate when we notice bloat, compensated productivity will not end up being a productivity value from which bloat is fully subtracted, such as with true productivity. There is certainly no way to detect parts that no one has yet noticed, but the foundation of this method is to try to increase awareness about bloat, and to compensate for it whenever it is noticed. It would be nice if there were another method for strictly determining compensated productivity, but there is unfortunately no other ingenious way except for compensating as described above.

Apart from plain productivity, which is pretty much meaningless, I would like you to see that meaningful software development productivity cannot be easily measured in numerical terms.

There are a number of problems, but the fact is that more than plain productivity, compensated productivity indicates a value closer to true productivity (productivity value from which bloat has been completely subtracted). Furthermore, the negative consequences that come from using plain productivity can be greatly mitigated by adopting compensated productivity. Plain productivity currently dominates, but we should quickly switch to compensated productivity. Although the troublesome compensation work, which was not necessary until now, might be a barrier to switching, it can be conducted concurrently with program review.

Topic 7: PC-Based Development and Review

I once stated at a seminar, “Compensation work for determining compensated productivity should be performed concurrently with program review,” but this just elicited puzzled looks. After looking into this, we discovered that program reviews are not being performed anymore. Out of the attendees there, only about ten percent where in the habit of conducting reviews.

Since computers were once expensive, the time when you could use them was limited. Accordingly, a lot of effort was put into reviews so that debugging could be finished in a short time. It is great how PCs have become commonplace and available for use at any time, but it seems that reviews have been forgotten.

For the record, a **review** is work wherein a program that has already been written is revised. It is equivalent to polishing writing. There is self-review conducted by a single person, but most often, review is

conducted by several people affiliated with the development project. The purpose of a review is to find interface inconsistencies and bugs, but a review is also about information exchange among programmers. In short, it also serves as a setting for viewing beautiful programs and exchanging opinions on creating them.

Without information exchange, a program becomes a self-centered work and we miss chances to brush it up. Also, since common portions in each other's programs are left unidentified, reuse is hindered. These are tremendous problems because even if you heavily use computer power, development personnel will not be able to catch up no matter how many people you add during the development of bloated programs. Determining compensated productivity should substantiate this.

Development firms that directly link values estimated in the unit of "man-months" may temporarily profit, but they will end up paying it back in maintenance work. In short, they will need even more manpower for maintenance. This might be good for firms that profit by selling maintenance services, but for those on the ordering end, it is a harsh situation that adds insult to injury.

Therefore, even if you are using a PC, a review should be performed. Unlike before, we now have an environment in which we can conduct reviews between people that are geographically separated by using a network, such as a WAN or LAN, and suitable groupware. It is really ironic that although we have such a great environment for conducting reviews, they are actually being conducted far less due to heavy computer use.

Incidentally, the aim of a review can also be achieved by refactoring, which springs from a slightly different viewpoint than reviewing. Consequently, if reviewing is the older of the two, then maybe refactoring is better. Based on this, we probably should have been asking, "Are you refactoring?" rather than "Are you reviewing?" Refactoring is not only the reduction of redundant portions in a program; it is also the enabling of reuse by transforming a program in specifically targeted situations.

4.2-e Implementation Verification for Determining Improvement Rate of Productivity

We have been recommending the combined use of multiple measurement methods to find a productivity value that is more widely accepted than current methods, but now, instead of questioning the absolute value of software development productivity, let's focus on the relative value of how many times productivity will increase/decrease, i.e. the scaling factor, when certain seeds (materials, measures, mechanisms, and/or structures) are sown. We will use the term **improvement rate of productivity** to represent how much each seed improves productivity.

The improvement rate of productivity has the potential to be meaningful data that is clear and easy to understand. This value is useful information for making a rational decision about the order in which seeds should be sown. It should be enough to measure this improvement rate of productivity when evaluating what became of productivity as a result of sowing a certain seed. There should be no need to go out of the way to determine the absolute value of productivity.

There are two methods for determining the improvement rate of productivity. Let's start with the method for finding it by means of implementation verification.

At first glance, the improvement rate of productivity, due to certain seeds, may seem easy to measure by means of implementation verification. In other words, you may think that if the same development team develops the same development target, first by not sowing those seeds and next by sowing them, you would

be able to determine the improvement rate of productivity (reciprocal of the total work hours ratio). For example, if you assumed that it would take two hundred man-months if those seeds were not sown and one hundred man-months if they were, then you could say that productivity only doubled due to those seeds. This is nothing more than determining the improvement rate of productivity by means of implementation verification.

However, such implementation verification is not so easy. If the same team were to develop the same software twice, productivity would obviously be higher the second time because the developers would have remembered important development information from the first time. Reuse would be carried out naturally. In this situation, different teams would have to develop the same development target to get a valid measurement. The problem here is that there would be no guarantee that both teams would have equal development ability. Accordingly, the need arises for having multiple teams develop the same thing and then taking the average value to get a value that is statistically meaningful. For example, we would have to conduct large-scale implementation verification in which we made ten teams develop without sowing those seeds, made another ten teams develop by sowing those seeds, and then compared the results. It would be like a test for verifying the effects of a new drug.

This measurement method may seem like a major undertaking, but it can produce highly precise measurement results. This is because, if we assume that the quantity of products is the same without worrying about how to measure the products or how to detect bloat and adjust plain productivity, then we need only to compare total work hours. The problem with conducting this implementation verification is that we get products of varying qualities, such as those with good/bad performance and those that are buggy/not buggy. Consequently, we will not get a fair measurement result unless we deal with such qualities as well. It is actually difficult to match the qualities of products, but it seems that we can determine a fairly pertinent **improvement rate of productivity** by setting up a concrete standard for software quality and then making comparisons after establishing a standard of attaining or exceeding a certain level.

This sort of implementation verification may seem extremely meaningful, but we have never heard reports of it being conducted. People probably avoid it because they consider it to be troublesome and a major undertaking. Alternately, they consider it to be unproductive, and therefore, do not apply to become members of such a development team. Above all, mounting cost is a drawback. However, like motor sports, which consume vast sums of money, but also improves automobile safety, the development race really does contribute to increased productivity. Implementation verification could likely be carried out smoothly if it were possible to appeal to playfulness and procure funding by increasing the number of people that approve taking pleasure in the development race.

First of all, software development depends almost completely on the parameter of human mental activity. Since this parameter, influencing productivity, varies widely in a variety of senses, we are always under its shadow, even if we try to measure the effects of each of the seeds. However, if we want to seriously measure the improvement rate of productivity, it seems appropriate that we conduct this sort of implementation verification. However, as long as we do not conduct implementation verification with this level of detailed attention, it looks like we will not be able to accurately determine the improvement rate of productivity.

Based on this, I would like you to once again see that software development productivity and the improvement rate of productivity cannot be easily measured in numerical terms.

4.2-f Build-Up Method: Another Way to Determine Improvement Rate of Productivity

In addition to implementation verification, the build-up method is another way to determine the improvement rate of productivity due to the sowing of certain seeds.

The build-up method starts with a step that estimates effects in individual work, and then by means of a step that builds this up, it computes the total rate of work saving to determine the improvement rate of productivity. The latter build-up step will produce the same result no matter who does the computation, but the former effect estimation step cannot avoid relying on subjectivity and sensitivity. To determine the improvement rate of productivity under such adverse conditions, it is important to make an effort to be as objective as possible and shed light on what makes the value relevant. Accordingly, we recommend representing effect estimation as the product of the **percentage of supported work** and the **percentage of work saving**. Making this the product of two numeric values, rather than one, demands even deeper effect analysis, and if a dramatic difference arises compared to another person's estimated value, analyzing that difference would also be helpful.

To determine the improvement rate of productivity, we should estimate the following two percentages, convert them into a ratio (where 100 % is 1), and then calculate the reciprocal of their product.

• Percentage of supported work:
(Indicates, as a total, what percentage of work those seeds support out of all development work for the software. In other words, totaled percentage of supported work, wherein software development work, extracting maintenance activities, will be 1, i.e. 100 %.)

• Percentage of work saving:
(Indicates what percentage of the work will be automated within the supported work. In other words, the percentage of work saving within the target work. For example, 100 % if complete work saving is possible, and 0 % if there is no support for it whatsoever.)

For example, if 5 % of the work accounting for 20 % of all development were supported, then by determining the product of each percentage, we would see that only 1 % work saving is possible. 1 % work saving means a productivity improvement of about 1.01 times (1/0.99 times).

The percentage of supported work and percentage of work saving are actually quite difficult to measure in numerical terms, so there probably is no other way to boldly estimate it than by relying on subjectivity and sensitivity. However, as long as the degree of roughness is at least 5, 10, 20, or 40 percent, we can likely come up with an estimate without much error. Even if we can accept that degree of precision, we still would like to measure by numerical terms.

However, among seeds that improve productivity, there are those that not only support specific work, but also affect the post processes. For example, seeds that support the clear definition of requested specifications not only improve the productivity of the work itself, but also have the effect of decreasing the need to redo post-process work. The rate of work saving for post-processes in such cases is similarly represented in the form of the product of the percentage of supported work and percentage of work saving, and then this is built up (accumulated) to calculate the total rate of work saving.

Incidentally, this computation method also seems usable for quantitatively theorizing the effect of productivity improvements by increased reliability. Since improving reliability makes it possible to decrease needless work spent in post processes, productivity can be improved.

Now when we actually try to calculate the degree of productivity improvement, such as tool improvements, on a per-seed basis, we are almost always disappointed by the extremely low values that do not reach one percent for most of the time. Telling yourself that every little bit helps and sowing many seeds that have a small effect is probably another way. However, since effective seeds have already been exhausted, it is really unfortunate that there are no more left with a substantial effect.

Please refer to **Appendix 3 "Example Using Build-Up Method to Determine Improvement Rate of Productivity,"** which gives an example of calculating the degree of productivity improvement by the build-up method.

4.3 Is Software Development Productivity Improving?

This section tackles the problem of whether software development productivity is improving. However, it is incredibly difficult to answer this question by presenting hard evidence. Accordingly, we will make comparisons with manufacturing work productivity, and so on, while reflecting back on software development history to present circumstantial evidence as to whether software development productivity is improving.

4.3-g Why Has It Been Possible to Improve Productivity of Manufacturing Work?

For reference purposes, let's begin by trying to look into the reason why manufacturing work productivity is improving.

Even in manufacturing work, the productivity of repetitive work where large numbers of copies are made is reaching considerably high levels through the mass production of industrial goods. In the manufacturing of many industrial goods, machines rather than human beings already carry out a major portion of manufacturing work. Accordingly, human beings perform nothing more than ancillary work. Through today's advanced technology, although costs may be high, even the complete automation of manufacturing work is already possible in many areas. For example, if you visit a plant that manufactures robots, you can observe robots that are automatically manufacturing certain types of machinery. You can also see with your own eyes how such robots themselves are being automatically manufactured day and night by other robots.

Since total work hours in which human beings are directly involved will approach zero, if adequate facilities investment is made, productivity (based on certain infrastructure) can be become nearly limitless.

However, since the monetary amount of investment diverted for infrastructure improvement can be suppressed according to how much goods of a certain price level can actually be manufactured, it has become a battle to see how far productivity can be raised within a limited range of resources. Another way of saying this is, although productivity can be raised, doing so inexpensively is difficult.

Why do you suppose it has been possible to raise productivity to a high level when manufacturing a large number of copies? A major factor therein, of course, is the build-up of ideas and efforts by a large number of people, but you could also say it was due to the following two factors of copy manufacturing.

The first is productivity that can be expressed in objective numeric terms. When a new proposal is made for improving productivity, we are able to accurately grasp its degree of effect and degree of side effect, and thus there is no room for the emergence of mistaken beliefs. New proposals can be accurately and clearly evaluated.

The second is that manufacturing, no matter how you look at it, is an activity in which mechanical processing is possible requiring no creative intellectual work. Actually, whenever you try to mechanize the manufacturing process for some sort of goods, there will likely be a variety of problems in carrying it out, and solving them will likely require the intellectual work of manufacturing engineers. However, as already indicated by many past mechanization achievements, there is no doubt that almost all such problems can be solved technologically.

4.3-h Why is It Difficult to Improve Productivity of Software Development?

In contrast to manufacturing productivity, design work, such as software development, does not have the above-mentioned two factors that are helpful in improving productivity.

First of all, it is difficult to express productivity in objective numerical terms. To obtain a common view about productivity, there is no escaping, to a certain extent, the reliance on subjectivity and sensitivity when determining such numeric values, even if you exert your best efforts to adopt the most scientific technique possible and make a decision through the combined use of multiple methods. Large-scale implementation verification is required so you will not have to rely on subjectivity and sensitivity. This was already discussed in **4.2 "Various Ways of Measuring Software Development Productivity."** Consequently, when a new proposal for improving productivity is made, what degree of effect it really has is not entirely clear. Under such circumstances, there is a tendency for people to think that their technique is the best.

Second of all, design includes work that is difficult to process mechanically. The main portion of design is thinking up what you will try to make and then fleshing it out so that anyone can manufacture it. Since the majority of such intellectual work cannot be mechanically processed, there is no other way but to have human beings take care of it.

What computers can do is either to perform calculations according to formulas set up by human beings or to support human design work in an auxiliary or indirect manner. Under usual circumstances, we would like to leave the main portion of design work to computers rather than to human beings. However, the majority of design work carried out by human beings is not of a type that can be represented by numeric expressions. Furthermore, even figuring out which approach should be taken to make a computer take care of such work is like groping blindly in the dark at present. In contrast to this, we can confidently state that, for the most part, mechanical processing is possible in manufacturing.

4.3-i Improvement of Software Development Productivity in the Good Old Days

Under such circumstances wherein it is definitely not easy to improve productivity, you might wonder whether software development productivity is improving. The general consensus is that "software design productivity is not improving in proportion to the dramatic improvements being made in hardware manufacturing productivity." Even the vast majority of people involved in software development, if they were to reveal their true feelings, probably think productivity has hardly improved at all.

To underscore this problem, we will try to definitively show that software development productivity has hardly improved at all. Then we would like to decide whether productivity is improving based on whether we can present a counterargument to what we have shown.

Since there must be some sort of seeds (measures, mechanisms, and/or structures) for improving productivity, we'd like to start by trying to prove whether picking up such seeds has really improved productivity by briefly reflecting back on software development history.

Based on this, we get the following argument stating that it was possible to improve software development productivity in the good old days.

> Everyone will admit that the evolution of programming languages from **machine language** (first generation) to **assembler language** (second generation) and then to **compiler language** (third generation) has produced easily discernable productivity improvements. The use of interactive computers and the development of editors have also doubtlessly had an impact in their own right. For example, if you estimate that switching from assembler language to compiler language reduces by 5/6 of the work that occupies a total of sixty percent of development, and then the resulting work saving for all development would be fifty percent, thereby improving productivity by a factor of two.

As you can see, because there were a number of seeds (measures, mechanisms, and/or structures) that made computer processing easier and had a major effect in the early days of computer development, incorporating them into tools made it possible to greatly improve productivity.

The above-mentioned argument therefore makes sense, and we can thusly accept the fact that software development productivity improved in the good old days. We can even accept the fact that the utilization of compiler languages raised productivity to a certain level, and that that level became the industry norm.

However, if you look at what happened after, it appears that seeds that had a major effect in the good old days were soon exhausted, and productivity from that point on barely improved at all. Perhaps this is what has been causing the stagnation of tool productivity over the past ten years or so. In other words, it seems we have not been able to raise productivity to as high a level as the previous industry norm.

Do you think a counterargument can be presented for this? Doing so will require us to discuss seeds that have a major effect and clearly indicate their effect. Since **4.2 "Various Ways of Measuring Software Development Productivity"** and **Appendix 3** "**Example Using Build-Up Method to Determine Improvement Rate of Productivity"** provide an in-depth description of the evaluation method for seed effects, we should use this information to present a counterargument.

Do you, the reader, know what seeds we could use for our counterargument?

4.3-j Productivity Improvement Plan Based Only on Tools

Having been a minor development support tool vendor since 1994, if anyone could, I should be able to present a counterargument to this. Unfortunately, it is not possible to present a counterargument. Although we would like to present a counterargument, the seeds that make computer processing easier and have a major effect have already been exhausted. Consequently, in the wake of seeds that enable mechanical processing in design work to replace what humans do, there are no good seeds left.

After studying a group focused on the average value of the number of program lines that a single person could write (complete) in a day, it seems during the past ten years or so, that value has been confined to between twenty to thirty lines and has barely increased. You would think that by now we would be able to write even a few more programs, but the industry is stuck at this low norm. Since this value depends on the

speed at which human beings can carry out intellectual work, we cannot expect any major growth as long as human beings themselves are conducting the main portion of design work. Of course, there will be variation in the speed at which each person can produce a program, and we can also expect speed improvements through training. It would seem that such factors of variation would be hard pressed to raise the average value by even a factor of two or three overall. Another matter all together would be a mutation that dramatically improved the development ability for software which then spreads to the entire human population.

Based on this, we would like to pin our hopes on seeds that have computers carry out the intellectual work that forms the heart of design. In short, it would be nice to have computers rather than human beings carry out the main portion of design work. However, this is far out of the realm of possibility due to the current state of technology. Once we accept this fact, there are two things we can do, as follows:

- Incorporate the seeds sown so far into tools, or in other words, extract what can be processed mechanically from design work and have computers carry it out; or
- Aim for a cooperative effect that accompanies support by tools.

In the former case, unfortunately, the only seeds left have almost no effect whatsoever. However, since we say that every little bit helps, even supporting seeds that improve productivity little by little, is probably not a waste. Then again, we can no longer expect a major effect.

The cooperative effect of the latter is created by computers carrying out mechanical processing. For example, word processor capabilities not only offer support that makes editing work, such as the addition, modification, and deletion of text easy, but also support the creation of difficult documentation. No one wants to read illegible handwriting to which lots of corrections have been made, but when you see nice word processor output, polishing the writing becomes enjoyable and you want to unify the format whenever you find parts that are not aligned. Doing so certainly increases productivity, although it is difficult to measure the effect in numerical terms. Computers are only as effective as their users.

4.3-k Providing a Pleasant Software Development Environment

If there have only been slight productivity improvements in terms of the average value of the number of lines that a single person writes (completes) in a day, you might wonder what on earth tools have done, or if that is all tools for increasing productivity are capable of. Let's respond to this here.

Almost all tools make human work much easier by taking over work that is grueling and troublesome for them. The diagram editor of CASE tools reduces the load of tiresome diagram revision work, word processors and program editors take the trouble out of editing work, such as additions, modifications, and deletions, and documenters extract a great deal of documentation from program information (as a result, human beings no longer have to check for inconsistencies between program information and documentation). Aren't these really helpful tools? Combined with the cooperative effect that comes with tool support, they really make development work much easier.

From this perspective, you can see how the seeds that have been incorporated into tools over the past ten years or so, have been pursuing the goal of making software development work more pleasant, more than contributing to productivity increases in the true sense of the term. ~~For example, automobile development aimed at offering pleasant spaces after basic car functionality matured.~~ Another example of this can be seen in the shift from basic car functionality to offering pleasant spaces in automobile design. Similarly, providing a pleasant software development environment for software development support tools is also a

crucial topic. There is no doubt that this is something that increases productivity in the broad sense. However, it is not a productivity increase in the strict sense as we imagined being measured, as discussed in **4.2 "Various Ways of Measuring Software Development Productivity,"** but instead is mere fanfare.

Topic 8: How Much Do Tools Improve Productivity?

Being a minor development support tool vendor, I am sometimes asked an annoying question. Some people will point out a specific tool and ask how much it improves productivity. The annoyance comes from the fact that even if they ask about the improvement rate of productivity for a specific tool, they themselves are the ones who hold the answer to that question.

We have already introduced two methods for determining the improvement rate of productivity. It does not matter whether you use implementation verification or the build-up method, but in either case, there are many things that must be made clear.

Now I would like to introduce how to find the effect of the seeds that support the drawing of diagrams using tools, as illustrated in **Appendix 3 "Example Using Build-Up Method to Determine Improvement Rate of Productivity."** I will explain that in the case of development projects with a high rate of error in comprehending the requested specifications, productivity can be increased to a certain degree, but in other cases it cannot. As a result, we can only determine the improvement rate of productivity if we know the error rate in comprehending the requested specifications in this example. Furthermore, the degree of benefits of tools will differ depending on whether the development project is using the technique of writing things down in diagrams. As you can see, the improvement rate of productivity by tools will vary widely depending on the nature of the development project and the development techniques employed therein.

Regardless of what will come in a hundred years from now, there currently are no tools that automate all development work. Consequently, we must assume that human beings must first and foremost pursue jobs using tools. Once we do this, what sort of actions human beings have taken comes into question, and the improvement rate of productivity will vary depending on that.

Accordingly, I usually answer in the following manner when asked about the improvement rate of productivity for a specific tool.

> Generally, tools increase productivity when they are used in a manner that suits their purpose. Therefore, when selecting a tool, it is important not only to listen to and agree with explanations about where it excels, but also to actually give it a try so that you can get a feel for whether or not it is suited to your development project. Generally speaking, there are no tools or other aids that can declare they increase productivity x times in every case. Only by trying a tool out to see how it will work for your business program development will you come to see to what degree productivity can be raised.

To paraphrase President J.F. Kennedy, "Ask not how much a tool can raise productivity for you; ask how much you can raise productivity using that tool."

4.4 Improving Software Development Productivity

We have thus far discussed the negative opinion that software development productivity has hardly improved at all over the past ten years or so. If we tentatively accept this, does that mean that the way to improving productivity is completely blocked? Well, if it is difficult to raise productivity, we should probably ask whether there is a way for harnessing an effect similar to productivity improvement. In other words, we should rephrase the problem by saying what can be done if we say that the development speed of a densely packed program is limited.

4.4-l Improvement Rate of Productivity by Reuse

Fortunately for people who would like to improve productivity, most programs are inflated with bloat making bloat a good place to focus. Measures for reducing bloat and eliminating needless work, i.e. promoting component-based reuse, raise productivity. This is a way for improving productivity not with tools alone, but rather by the combination of tools and components.

Interestingly enough, when you promote component-based reuse, in extreme cases you will encounter situations in which you can produce (develop) a software product with certain features even without writing (developing) a program. Consequently, we need a renewed awareness of software development productivity. Here, we would like to recall "**Topic 1: Dreaming of the "Golden Egg" Business Package**" that was discussed in **1.1 "Differences between Custom Business Programs and Business Packages."**

To begin with, the word "productivity" does not click with the approach of thorough reuse without software development. People who are misled by the sound of the word "productivity" tend to fall into **plain productivity for its own sake**. They simply think they should increase the quantity of products. Since changing your mindset to promote component-based reuse, i.e. increasing the rate of reuse (rate of non-production), rather than thinking only about increasing the quantity of products will decrease total work hours spent on development, it is another way to actually improve productivity.

Accordingly, we will introduce the **improvement rate of productivity by reuse** as a gauge representing how much productivity has actually improved. Since you can eliminate needless work if you remove bloat from a certain piece of software, i.e. you thoroughly reuse what can be reused; this gauge indicates how much productivity will actually improve. Accurately determining this value requires the previously discussed implementation verification for both the non-reuse case and the reuse case. However, if a rough value is sufficient, you can use the following formula:

Improvement rate of productivity by reuse = Plain productivity/Compensated productivity

Since the value of this formula is the same as the number of program lines actually developed divided by the number of program lines that really need to be developed (value of the formula shown below), it indicates the improvement rate of productivity by reuse if we assume that the work time per number of program lines (time taken to develop the program) is fixed.

Number of program lines actually developed/Number of program lines that really need to be developed

While we are on the subject, let's try using the formula to determine the improvement rate of productivity by reuse for two special cases. The first one is a case of all bloat. In this case, compensated productivity is zero and the improvement rate of productivity by reuse is infinite. The second one is a case of no bloat whatsoever. In this case, plain productivity and compensated productivity coincide. Consequently, the improvement rate of productivity by reuse will be 1. A value of one means that since there is no bloat, productivity cannot be increased through reuse. From another point of view, if there is bloat, there will be room for improving productivity through reuse. That is why we have focused on the fact that most of the programs developed thus far are inflated with bloat.

4.4-m What Does Improving Productivity by Reuse Mean?

This book has so far discussed on many occasions the improvement of productivity through reuse, which means an **improvement rate of productivity by reuse** of above 1. You could reword this by saying that since reuse will enable you to get rid of needless development work that produces bloat; it will end up improving productivity.

Note that the value of **compensated productivity** will not end up increasing, even by aggressively pursuing component-based reuse, because compensated productivity is a criterion that does not consider bloat to be a product, i.e. a strict criterion that regards waste as waste. On the other hand, **plain productivity** even includes bloat among development products, and can thus be called a rough criterion so to speak.

Just as the question "How much can tools raise productivity?" is generally meaningless, the question "How much can reuse raise productivity?" is also generally meaningless. However, when you aim to develop a specific business program for a specific development project, both of these questions will come to have meaning. There are even cases where just a little improvement in a certain development project will exert an eye-opening effect on productivity improvement.

For example, to get the benefits of reuse, turning redundantly developed portions into common subroutines will exert an eye-opening effect on development projects that are not accustomed to the use of common subroutines. In addition, to raise productivity, switching to a development method that distributes templates of main routines related to operation characteristics and then copies and appropriately revises them is also effective for projects whose developers develop multiple individual procedures related to operation characteristics.

As you can see, the degree of productivity improvement by reuse will vary depending on the degree of component-based reuse carried out. Essentially, the lower the degree of reuse in a development project, the greater the effect. Furthermore, some sort of an effect can be expected as long as the development project is not carrying out one hundred percent component-based reuse. Note that one hundred percent reuse means preparing a business program simply by combining components.

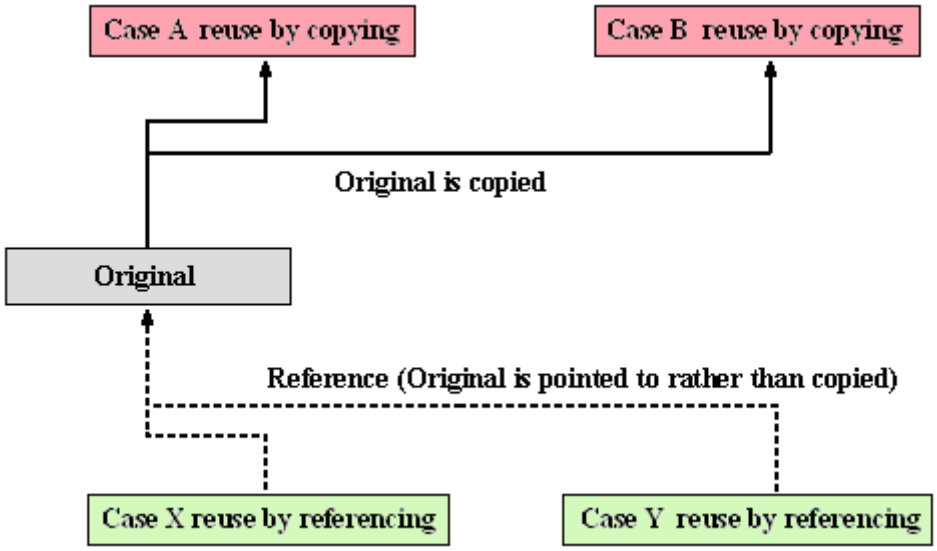
When you look at it this way, there are almost no development projects that carry out one hundred percent component-based reuse, and thus, we can see a great deal of room for improving productivity. In short, we could improve productivity by promoting reuse b the combination of components and tools (or by tools, such as a 4GL, that include components). In contrast to this, we have previously discussed that we do not know how to continue improving productivity because productivity improvement plans based only on tools have already used up the seeds for doing so.

4.4-n Two Methods for Improving Productivity by Reuse

As **Figure 4-1** shows, there are two methods for component-based reuse. Adopting either one will produce better results than a development method that lets developers do it their own way. Although developers may get joy from the creative endeavor of making new discoveries when doing it their own way, the end result tend to be haphazard. Therefore, we can be sure that copying with a known lineage is still better than using haphazard methods, and it is much better to systematically employ reuse by referencing than by copying.

Reuse by copying

Can increase development speed for programs.
However, it results in bloated programs.
Seen as high productivity because a large number of programs can be developed.
However, it increases resources that must be maintained, thereby making maintenance difficult.



Reuse by referencing

Number of lines to be developed for a program tends to be cut in half or a third.
(Example: Enhancing common subroutine libraries and then thoroughly reusing them.)
Results in compact programs.
Apparent productivity will not improve, but productivity in the true sense will improve.
Resources that must be maintained decrease, making maintenance easier.

Figure 4-1: Two Methods of Reuse

One method of component-based reuse is **reuse by copying**. It increases program development speed regardless of whether the program is bloated.

There are a variety of conceivable methods for increasing speed if you are developing a program that is not densely packed. For example, you could teach program expressions that are often used in order to cultivate veteran developers who can repeat them off the top of their head. If repeating them off the top of the head is difficult, another conceivable method is learning by rote each program fragment that will serve as an example and then recommending that the appropriate things be copied. Among such methods, you should probably recommend most strongly such things as copying the main routines related to operation characteristics.

If you were to include the copied portion in the count of newly developed lines in this manner, apparent productivity (plain productivity) would dramatically increase. Doing so would not increase problems beyond conducting development one's own way as has been the tradition. The problem of increased software resources filled with bloat, causing heavier maintenance loads, still remains, but copies (bloated programs) for which the lineage is known are easier to maintain than haphazard bloated programs that are reinvented by developers who do things their own way.

Another method is **reuse by referencing**. It aims to develop programs that carry out equivalent processing with fewer newly developed lines. Bloat is extensively removed using this method.

This enables the development of a program that carries out processing equivalent to what has been possible thus far by using fewer newly developed lines (half to a third less than what was required by one program thus far). For example, you would enhance general subroutine libraries and then teach developers to extensively reuse them. However, under the present circumstances in which the main general subroutines have already been brought to light, and the grouping and reuse of common subroutines has become commonplace, it would probably be difficult to use even fewer newly developed lines than before. This is certainly true, since the common main routines and data item components discussed in this book have not been used much thus far, using them will likely have a considerable effect. In short, if we enhance common main routines and data item components and then aggressively reuse them, there will be grounds for creating programs that carry out equivalent processing with fewer lines.

For the record, reuse by referencing is an approach that pursues no copy reuse rather than bloating, and since it does not inflate programs, it does not improve apparent productivity (plain productivity). Nevertheless, it still has the same effect as improving productivity.

4.4-o Evaluating the Two Reuse Methods

Now let's try to evaluate these two methods.

- **Reuse by copying**
- **Reuse by referencing**

Reuse by copying is an approach that gets developers accustomed to reuse and recommends systematic copying. Therefore, bloat resources will accumulate. Since copied resources will be changed over time, their lineage will be more and more difficult to understand, and eventually the copied portions will have to be considered new software resources independent of the original copy source. In other words, this means resources that have to be maintained will increase, leading to a heavier maintenance load.

Reuse by referencing is similar to this in the fact that it gets developers accustomed to reuse, but it does not recommend reuse. It is an approach that changes software into an easy-to-reuse structure, resulting in the elimination of bloat resources. For example, even though this approach reuses 'Business Logic Components,' it does not increase software resources that have to be maintained.

From a maintenance standpoint, reuse by referencing is definitely more preferable. Furthermore, reuse by referencing has future possibilities. Just as mechanization in manufacturing is a way to approach infinite productivity, thoroughly pursuing reuse by referencing in software development can do the same thing. By

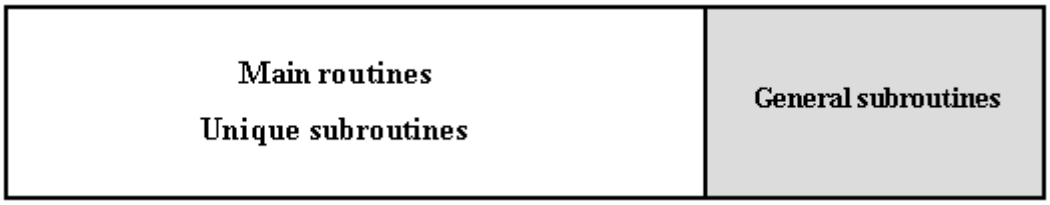
enhancing ‘Business Logic Components’ fully, the necessity of new development will decrease to nearly zero. This is more than possible using current technology.

Of course, the way to infinite effect will probably not be easy. Just as infrastructure for mechanizing manufacturing processes on a per-manufactured-good basis is required in industrial goods manufacturing, the infrastructure of a set of ‘Business Logic Components’ on a per-application-field basis is also required in reuse by referencing. There are probably also issues such as continued efforts for enhancing ‘Business Logic Components.’ However, there is no doubt that reuse by referencing substantially rationalizes software development.

4.4-p Reuse Stages and the Two Methods

As shown in **Figure 4-2**, there are three stages of development in component-based reuse for a typical business program in the business field.

Traditional Business Program (First Stage)



Business Program Developed Using Fourth-Generation Languages (Second Stage)

Componentized Application (Third Stage)

The shaded portions are those that are actually reused or become easier to reuse when covered by ‘Software Components’ or ‘Business Logic Components.’

Figure 4-2: Expansion History of Component-Based Reuse

Most development projects reuse general subroutines by grouping common portions. The reuse of such subroutines has already become the norm. A business program developed by such projects at least reaches

the **first stage**, and from twenty to thirty percent of the program can be covered by the reuse of general subroutines. Obviously, even if more subroutine reuse were recommended for such projects, it would not be possible to improve productivity any further. Even if it were possible, only a very small productivity improvement would occur compared to what we failed to reuse.

Incidentally, there seems to be a fair number of projects that do not reuse common main routines related to operation characteristics. Such projects can increase productivity by starting to reuse common main routines. Business apps brought to the **second stage** in this manner are likely covered twenty to thirty percent by the reuse of general subroutines and thirty to forty percent by the reuse of common main routines. Furthermore, productivity can be improved only by the amount that common main routines were newly reused, i.e. the amount of increase from the first stage to the second stage.

There is a way to start the reuse of data item components in projects that attempt to carry out reuse even more thoroughly. Business apps brought to the **third stage** in this manner are likely covered twenty to thirty percent by the reuse of general subroutines, thirty to forty percent by the reuse of common main routines, and the rest by data item components. In short, they are turned into componentized applications. However, going from the second stage to the third stage does not increase productivity that much when developing the first version of a business program. Productivity increases in the phase where the first version of a business program (i.e. a componentized application) is reused. Examples would be customizing the first version of a business program for another customer, or performing maintenance on the first version of a business program.

Now let's take an in-depth look at the process from the first stage to the second stage. There are two methods for reusing main routines. One is "reuse by copying," which distributes templates of main routines related to operation characteristics and then copies and appropriately revises them. The other is "reuse by referencing," which reuses common main routines by means of a 4GL (tool that includes components) and a fill-in system (combination of components and tools). Note that you can say development by means of a pre-generator (software tool) that we discussed in **"Topic 6: Tools for a Componentized Event-Driven System"** in **3.2.3 "From SSS to RRR Family"** is reuse by copying. Development by means of a post-generator (software tool) is reuse by referencing.

You can improve productivity in a similar manner no matter which method you adopt. However, since reuse by copying increases maintenance load, reuse by referencing should always be used. You will better understand this difference in maintenance load if you consider an example of when changes must be made to the original. With reuse by referencing, changes only have to be made to the original, but with reuse by copying, all programs developed by copying the original must be changed.

We have thus far discussed the method for turning a typical business program in the business field into a third stage componentized application. We think you will be able to see how doing this will dramatically improve productivity since almost no development products have yet to attain one hundred percent component-based reuse.

Topic 9: Improvement Rate of Development and Maintenance Productivity

This book has examined productivity targeting only development work, i.e. productivity that does not take maintenance work into consideration. However, considering the fact that the term **maintenance hell** frequently pops up, it is also crucial to take measures to prevent being inundated by maintenance work (including adding/changing features and fixing bugs) which is two to three times more time consuming than development work. The actual percentage of maintenance work varies depending on a variety of circumstances in each enterprise's information department, but from the perspective of maintenance cost, it normally accounts for a large percentage as a fixed cost and cannot be ignored.

Note that the following discussion of the **improvement rate of development and maintenance productivity** will cover **improvement rate of development and customization productivity** by replacing the word maintenance with the word customization. Consequently, we ask that you read the following paragraphs from both the viewpoint of maintenance and customization. For example, the following discussion is very important for businesses that customize business packages because productivity targeting customization work is a key factor influencing profit.

Now we will examine not only the improvement rate of productivity targeting development work by itself as we have done so far, but also the improvement rate of productivity targeting maintenance work. Note that we will refer to the former as the **improvement rate of development productivity** (IR of DP) and the latter as the **improvement rate of maintenance productivity** (IR of MP). In addition, we will refer to the improvement rate of productivity through the entire lifetime as the **improvement rate of lifetime productivity** (IR of LP).

The following relationship exists between these three improvement rates of productivity:

$$\frac{1+\alpha}{\text{IR of LP}} = \frac{1}{\text{IR of DP}} + \frac{\alpha}{\text{IR of MP}}$$

(1 + **α**) / improvement rate of lifetime productivity
= 1/improvement rate of development productivity
+ **α** /improvement rate of maintenance productivity

In the above formula, **α** represents the maintenance ratio. This is the scaling factor that tells us how many times the total work hours that were required for development will be spent on maintenance. For example, if we let three times the total work hours of development be the time spent on maintenance, then the value of **α** will be 3.

For reference sake, let's try to evaluate the improvement rate of lifetime productivity by applying the above-mentioned formula to a specific case. Since the above-mentioned formula is hard to understand intuitively, presenting an example in numerical terms will make it easier to understand.

Let's say the total work hours of development is 10,000 hours and total work hours of maintenance is 30,000 hours. This means the value of **α** will be 3. Let's try to calculate whether improving development productivity five times or maintenance productivity two times is more effective in this case. According to the above-mentioned formula, the former improvement rate of lifetime productivity will be 1.25 and the

latter improvement rate of lifetime productivity will be 1.6, and thus we see that improving maintenance productivity two times is more effective.

We can prove this as follows. In the former case, work that previously took 40,000 hours ends up taking 32,000 hours (10,000 x (1/5) + 30,000), but in the later case, it ends up taking only 25,000 hours (10,000 + 30,000 x (1/2)). Consequently, you can see how improving maintenance productivity two times is more effective.

There is a tendency to question only the improvement rate of development productivity when trying to improve software productivity, but doing so is apt to lead to errors in selecting the improvement method. So, we should naturally question the improvement rate of lifetime productivity. Since whether you should make an effort to improve development or maintenance depends on whether the value of **α** is greater than or less than 1, it is important not to make errors in how you distribute your efforts. When **α** is greater than 1, i.e. a so-called maintenance hell is assumed, striving to raise the improvement rate of maintenance productivity more than development productivity will be more effective in improving the improvement rate of lifetime productivity. I think this is a very common-sense decision, but perhaps we often forget or ignore such rational decisions these days.

Incidentally, among seeds that improve productivity, there are those that have an effect on development work and those that have an effect on maintenance work. Therefore, in order to determine the effects of each seed, we need both the value of the improvement rate of development productivity and the improvement rate of maintenance productivity. For example, if we represent seeds that result in beautiful program structure using the two values of one percent for the improvement rate of development productivity and ten percent for the improvement rate of maintenance productivity, then we can clearly comprehend their effect. Furthermore, we can determine which seeds we should select to raise productivity based on whether the value of **α** is greater than or less than 1. When the value of **α** is greater than 1, we should emphasize seeds that improve maintenance productivity, and when it is not, we should prioritize seeds that improve development productivity.

The general tendency, thus far, has been to be totally preoccupied with development productivity and to neglect maintenance productivity. Consequently, this neglected maintenance productivity seems to be stuck at a low level and the seeds for producing a major effect have still not been fully gathered. Since this is the case, sowing effective seeds for maintenance should raise productivity over a lifetime.

However, when sowing seeds that will improve maintenance productivity, we often need to consider matters from the outset of development because cheap tricks alone cannot result in improvements. (For example, component-based reuse by ‘Business Logic Components.’) This makes it necessary to switch to a resource architecture that employs “business logic components.” Such a switch will reorganize resources that have to be maintained, thereby reducing bloat and dramatically improving maintenance productivity.

CHAPTER 5 Theory of 'Business Logic Components'

So far, we have discussed what a practical and effective component-based reuse system is, by zeroing in on business programs in business fields, such as production management and sales management. In this chapter, without limiting the fields of application, i.e. generally, we will attempt to find out what a practical and effective component-based reuse system is.

Some people with software-development experience have a strong distrust of **software components** that are nothing but generalized business logic components. This is likely due to the fact that high expectations for software components, which many people secretly harbor, are often shot down or fail to pan out when component-based trials do not go well. Such experiences have fermented distrust.

As we pursue a generalized construction technique for component-based reuse systems, we are realizing that making such a system effective is as hard as discovering a goldmine. Therefore, our inability to realize such a system or our regarding it as impossible for the time being seems reasonable. Moreover, it makes the breaching of the goldmine entrance during SSS (Triple S) R&D feel incredibly fortunate.

In order to understand this matter and to include a sense of conclusion, this chapter will develop our slightly abstract theory concerning a generalized construction technique for component-based reuse systems. We would also like to study the RRR family, a specific example of 'Business Logic Components' that we discussed in Chapter 3, in light of this theory.

5.1 Requirements for Practical and Effective Component-Based Reuse Systems

Up to now, component-based reuse in software has been carried out by general subroutines (small unit) or business packages (large unit). Furthermore, component-based reuse on a personal basis for most software assets has been flourishing. Based on the understanding that component-based reuse is carried out in that manner, and looking into what a component-based reuse system should be like to have a greater effect, the following three requirements come to mind:

- **Software products must be built up solely by combining components.**
  (Reuse such as with general subroutines when possible is half-hearted.)
- **The system must be able to meet all customization requests.**
  (Responding only to customization requests envisioned ahead of time, such as for business packages, is inadequate.)
- **Large numbers of developers must systematically benefit from component-based reuse.**
  (Component-based reuse on a personal basis will lead nowhere.)

In the following sections, we will individually describe the significance and difficulty of fulfilling each of the three requirements. After reading this information, I would like the reader to judge whether a component-based reuse system would be able to meet the majority's expectations if all of these requirements were fulfilled.

5.1-a First Requirement for Practical and Effective Component-Based Reuse Systems

According to stories from experienced developers devoted to preparing libraries for business programs in business fields, you reach the saturation point once you collect about 400 general subroutines, and any more you add after that will be used only on very rare occasions. They also say that their goal is to cover

forty percent of program code with general subroutines, but that is very hard to achieve. It seems there is a limit to the use of subroutines (the extraction of meaningful general functionality).

Many people have probably experienced not getting the business app they expected when they merely combined general subroutines, and there are likely those who felt the limitations of component use therein.

To transcend the sort of component-based reuse system that uses general subroutines and their built-in call mechanisms, increasing the percentage of code that can be covered with components is required. Accordingly, we should consider the **first requirement** of a desirable component-based reuse system to be "**Software products must be built up solely by combining components**" in order to transcend the half-hearted level of using suitable components if they can be found. In other words we are talking about trying to construct one hundred percent of a business program's program code using only components.

There is probably no need for a long-winded account of the difficultly of fulfilling this requirement, even if we look at the fact that almost no cases of attempting to do so have been reported. There are also those who think that it is impossible to fulfill this requirement because it is too severe.

However, the mixing in of bad currency, that cannot be called components, will have a negative impact even on portions where component use would be achievable, thereby making reuse difficult. Hopefully, it would be achieved by removing bad currency and re-minting it.

5.1-b Second Requirement for Practical and Effective Component-Based Reuse Systems

If all we had to do were meet customization requests envisioned ahead of time, it would be best to develop a business package and then employ parameter customization. You could say that a business package was a system that carries out component-based reuse by thinking of software as a product unit, or in other words, "component = product."

Transcending the component-based reuse system of a business package requires the ability to meet all requirements as with a custom business program. Accordingly, we should consider the **second requirement** of a desirable component-based reuse system to be "**being able to meet all customization requests**," not just the customization requests envisioned ahead of time. In short, we should aim for a component-based reuse system that is able to develop custom business programs.

The difficulty of fulfilling this requirement can be easily explained using the concept of **NCA** (Need for Creative Adaptation). Refer to **Note 10**. We will put aside a description of this term for now. Suffice to say there is a need for solutions through the creation of new programs, i.e. program customization, because NCA is high in business fields, and that is why it is difficult to fulfill this requirement.

Note 10: Creative adaptation here means creating a new program when trying to meet a variety of requests in a certain application field or area and uncreative means, such as setting pre-existing parameters, do not go well. Parameter customization can meet requests in areas in which NCA is low (low NCA areas), but that is not so in areas in which NCA is high (high NCA areas). In the latter case, adaptation by creating a new program, i.e. program customization, is required. Note that a parameter here means declarative information with clear-cut meaning when viewed externally just as with parameters in parameter customization.

A look at business packages will reveal that a gap exists between those with a wide variety of customization requests and those without, depending on the type of business and industry for which they are intended. For example, due to the constraints of legal provisions that must be obeyed in financial

accounting, the types of customization requests will also be limited, but in production control, there are a wide variety of customization requests, most likely because possibilities are pursued in a free-for-all environment.

In relation to this, I would like the reader to recall the content of "**Topic 1: Dreaming of the "Golden Egg" Business Package**" discussed in **1.1** "**Differences between Custom Business Programs and Business Packages**."

Perceiving the gap in ease of adaptation in simplistic terms may lead to the false idea that parameter customization can adequately meet all customization requests simply by increasing the number of parameters, even in an application field with a wide variety of customization requests. However, NCA for that application field is what determines whether it is acceptable to think in such simple terms. If NCA is high, there is no end to the parameters that could be added. This may be an overstatement, but new customization requests occur each time a new customer (enterprise) is encountered. Furthermore, there is no knowing what those new requests will be ahead of time.

Consequently, there is no simple answer as to whether meeting a wide variety of customization requests requires a large number of parameters or just a few. If anything, we should question whether the NCA for the application field is high or low by directing our attention to the root qualities causing such differences. Incidentally, many business programs in business fields are developed and were developed as custom business programs rather than business packages because NCA is high in these fields as a general rule.

5.1-c Third Requirement for Practical and Effective Component-Based Reuse Systems

Component-based reuse on a personal basis occurs naturally. There are some painters who draw sketches of flowers and birds ahead of time, and then when the time comes to paint the actual picture, they refer to their sketches on paper or recall them from memory. Component-based reuse on a personal basis for all manner of software assets, such as specifications and program code, is exceedingly common.

Based on this current state of component-based reuse, obtaining more benefits than possible from component-based reuse on a personal basis, or a single pipe, so to speak, requires increasing the number of pipes in order to be effective. Accordingly, the **third requirement** of a desirable component-based reuse system should be "**large numbers of developers systematically benefiting from component-based reuse**," rather than a single person benefiting from reuse on a personal basis.

This requirement assures that it is okay for the people who develop business logic components to be different from those who reuse them (for example, people who develop business packages with special customization facilities and those who carry out component customization). This sort of specialization is possible as long as this requirement is fulfilled. In short, this is a crucial requirement for allowing large numbers of developers to customize business packages with special customization facilities.

The difficulty in fulfilling this requirement is discussed hereafter.

When large numbers of developers start systematic component-based reuse, troublesome decipherment work on source program code becomes a major problem. First of all, when attempting to find modules that must be deciphered, developers may find that the correspondence between programs and specifications is complex or they may encounter modules that the original developers arbitrarily partitioned, making it difficult to identify what should be deciphered. Then, when decipherment starts, many difficulties may arise, such as the need to decrypt programs written in spaghetti-state code and the gradual increase of the range of code that must be deciphered as work progresses. Reusing source program code developed by someone else is generally never easy. Developers are thus faced with resolving these problems.

5.1-d Summing Up Requirements

We have so far provided an in-depth description of the three requirements for a practical and effective component-based reuse system. Refer to **Figure 5-1** for a graphic representation of the requirements.

- Software products must be built up solely by combining components;
- The system must be able to meet all customization requests; and
- Large numbers of developers must systematically benefit from component-based reuse.

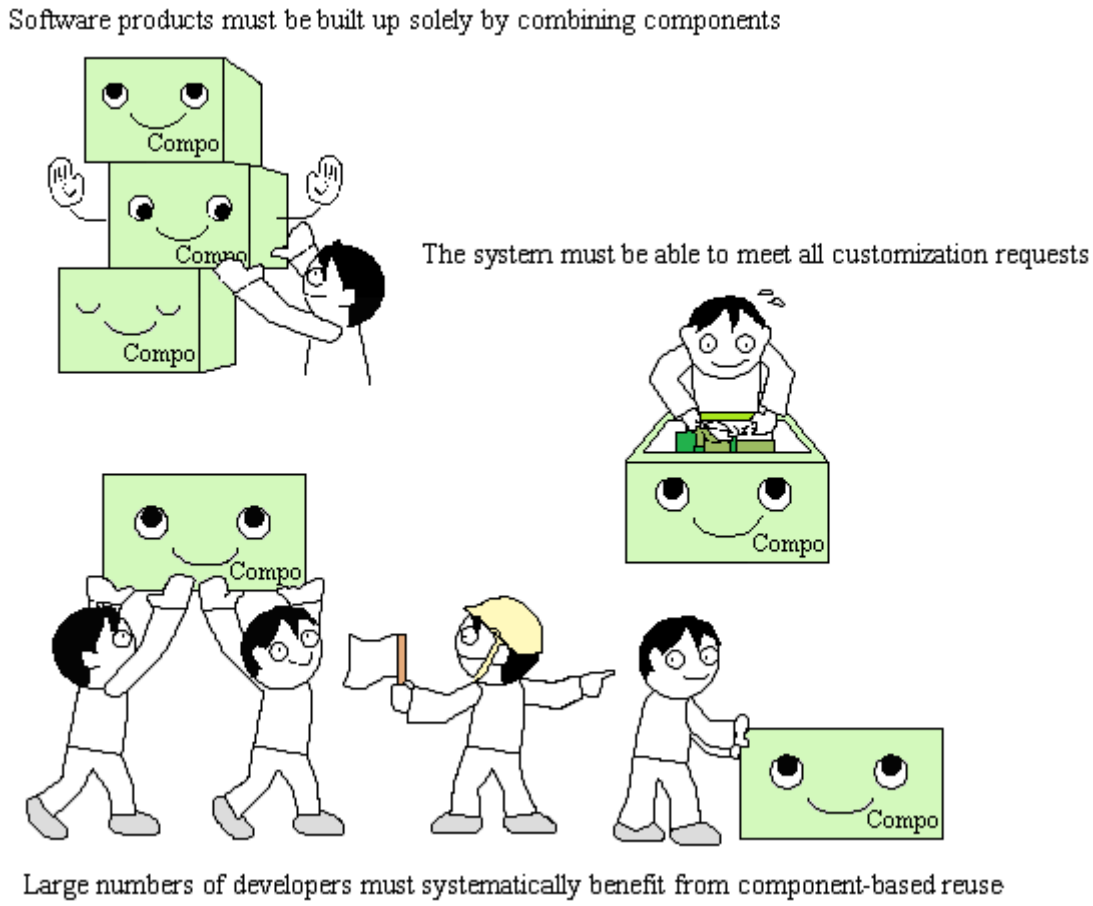


Figure 5-1: The Three Requirements for a Practical and Effective Component-Based Reuse System

You can probably see how a practical and effective component-based reuse system would result if the three requirements were all fulfilled. Such a system would appear to be able to rationalize the custom business program development industry and business package development industry from the bottom up in business fields. However, I would like the reader to understand the difficult problems that must be resolved before this can be realized.

Fulfilling the first of the three requirements, i.e. “Software products must be built up solely by combining components,” requires reconsidering what business logic components are. The reason for this is that general subroutines only cover low NCA areas, and you cannot build business programs in business fields based on that alone.

Accordingly, we must think about building software products using a combination of a number of "component" types. To cover high NCA areas, we need new types of 'Business Logic Components.' It is as if the need for an all-star cast of 'Software Components' was fulfilled by preparing the components that have not taken form on hard disk and then dragging them on stage.

As for the second and third of the three requirements, i.e. "the system must be able to meet all customization requests" and "large numbers of developers must systematically benefit from component-based reuse," it would be relatively easy to find a method that fulfills only one of them. On the other hand, fulfilling both would be extremely difficult because working out such problems results in mutually exclusive situations as described hereafter.

When a large number of developers aim for systematic component-based reuse, they often fall into the mindset that they must enable reuse even without deciphering programs. However, by doing so they can only do about as much as could be done by parameter customization of a business package. Consequently, it would only be possible to meet the customization requests envisioned ahead of time.

If you aim for a system that must be able to meet all customization requests, you will always be faced by the need to decipher programs. Furthermore, this would seem to be an obstacle to systematic component-based reuse by a large number of developers.

5.1-e Area Covered by General Subroutines

This section discusses an important matter related to the second of the three requirements for a practical and effective component-based reuse system.

The same argument for NCA in the area covered by business packages can also be developed for NCA in the area covered by general subroutines. In other words, the ease of meeting customization requests depends on NCA across the entire application field of the business package, and with general subroutines as well, the ease of doing so depends on NCA in the problem area that they cover. Both cases are exactly the same in substance, regardless of the size of the area itself. There is really no need to repeat the same description here.

I would like to change the perspective here a bit and discuss important matters concerning the area covered by general subroutines. Specifically, it is said that merely combining general subroutines will not yield the business app you expected, so now let's look into the reason why this is true.

If you think about common subroutines rather than general subroutines, it would be possible to build the business app you expect by developing subroutines and then combining them. This means there is some sort of difference between general subroutines and common subroutines. You can probably deepen your understanding by thinking about what is different while reading the following paragraphs.

When a developer provides a certain subroutine and has a user employ it, various requests for the addition of functionality may result. Sometimes those requests are fulfilled by developing new subroutines and turning them into a subroutine set. For example, adding a subroutine for calculating tax inclusion to where there was only a subroutine for calculating tax exclusion results in coverage beyond what there was before. If the developer knew from the beginning that there would be both tax inclusion and tax exclusion calculations, it would be best to enable both by establishing parameters in a subroutine for calculating consumption tax.

Anyway, no matter whether the developer meets the requests using a subroutine set or parameters, there are areas that subroutines clearly cover, for example, the calculation of consumption tax.

When a developer develops and provides a subroutine, whether it truly covers the relevant area comes into question. If it does not, the user of the subroutine will claim that the degree of coverage is inadequate. Furthermore, they may make requests, such as wanting to enable the calculation of consumption tax when they select simplified taxation. Since NCA is lower in the area of consumption tax calculation, requests will also be limited. However, in the case of subroutines that cover high NCA areas, there is the potential for endless requests. Such pesky problems have a tendency to pop up. When they do, the only way to handle the situation is to either modify the subroutine each time or turn over the source program to the requesting user and tell him/her to fix it themselves. In short, there is no escaping program customization.

Once a developer encounters, such as situation, they will be criticized for making poorly designed subroutines. Consequently, the subroutine designer will tend to deliberately choose low NCA areas and cover only such areas. In short, they end up aiming to cover every inch of the area by means of parameter customization that combines a subroutine set with parameters. In reality, it is possible to skillfully extract low NCA areas, and if a developer has adequate knowledge and experience in that field, they will be able to cover every nook and cranny of the area. Refer to **Note 11**. This eliminates the humiliation of having to switch to program customization later on. In short, there will be no annoyances from endless requests. If, on the other hand, the developer decides to cover high NCA areas, it will be highly likely that user requests that cannot be covered by setting up parameters will occur, no matter how many parameters the developer prepares ahead of time and how skillful the design is. That is why the coverage of subroutines, particularly general subroutines, is limited to low NCA areas, and high NCA areas are left out of general subroutine coverage and remain untouched

In other words, this means there are areas that cannot be covered by general subroutines alone. This backs up the realization many people have had that merely combining general subroutines will not yield the business app they expected.

If you change the viewpoint, all customization requests should be able to be met simply by combining general subroutines, as long as you are talking about low NCA areas. However, there is no such luck in business fields because they are high NCA areas.

Note 11: Even for low NCA areas, developing subroutines without any forethought will cause a variety of problems. This is because subroutines will be poorly developed, which is another problem besides what is mentioned in this chapter.

5.2 Technique for Constructing Component-Based Reuse Systems and an Actual Example

This section will examine what should be done to construct a component-based reuse system that fulfills all three of the previously mentioned requirements.

Although there can be many types of component-based reuse systems, this book focuses on RSCAs. Specifically, when constructing a component-based reuse system capable of component synthesis for application programs in a certain field, this book adopts the idea that the source of the components must exist within the application program that actually runs in the field. Furthermore, skillful surgery on the application program that actually runs in order to partition it into little pieces (i.e. refactoring the application program to make it component-based) can transform it into a component-based reuse system, which we call a reuse system of componentized applications (RSCAs).

Based on this idea, this section will place the spotlight on RSCAs that can satisfy all three of the previously mentioned requirements. We will start by presenting a generalized construction technique for an RSCA. Next, we will compare the technique with the construction technique for the RRR family, a specific example of a component-based reuse system. Finally, we will evaluate the generalized construction technique for an RSCA.

5.2-f Generalized Construction Technique for a Reuse System of Componentized Applications

The "generalized construction technique for an RSCA" refers to a construction technique that can be applied to non-business fields as well. The details of this technique are described below.

Generalized Construction Technique for a Reuse System for Componentized Applications:

- Partition the area that software products cover into low NCA areas and high NCA areas.
- Cover all low NCA areas by using 'Business Logic Components' or 'Software Components' that have the following quality (i.e. quality required of a 'Black-box Component'):
  - Ability to meet all envisioned requests in areas that must be covered by selecting components from component sets and specifying parameters (generality).
- Cover all high NCA areas by using 'Business Logic Components' or 'Software Components' that have each of the following four qualities (i.e. qualities required of a 'White-box Component'):
  - Easy retrieval of desired components (retrievability).
  - Number of components that must be revised is limited (locality).
  - Size of each component is suitable (suitable size; suitable granularity).
  - Each component is easy to decipher (readability).

This generalized construction technique was derived by organizing and breaking down the previously mentioned three requirements. For information on the process by which it was derived, refer to **Appendix 5** "**Generalized Construction Technique for a Reuse System of Componentized Applications**."

Note that if a certain software product can be constructed as a componentized application according to this technique, or in other words, if there is an RSCA that complies with this generalized construction technique, there is evidence for the theorem that the system truly can fulfill the three requirements in Appendix 5.

Since we are using the terms black-box component and white-box component in this generalized construction technique, let's take a look at what these two terms mean.

A **black-box component** refers to a component that can be used without deciphering the program in which it is contained. To clearly define the terminology, this book uses quotation marks specifically for 'Black-box Component' when it has the above-mentioned quality. As will be made clear later, a 'Black-box Component' is a type of "software component." Note that a black box is generally considered to be something that can be freely used without knowing its internal structure.

A **white-box component** refers to a component that sometimes requires the decipherment of the inner program, before it can be used. As with a 'Black-box Component,' this book uses quotation marks for 'White-box Component' when it has all of the four qualities - retrievability, locality, suitable size (suitable granularity), and readability - mentioned above. Refer to **Figure 5-2**. As we will make clear later, a 'White-box Component' is also a type of "software component." Note that just because something is a

‘White-box Component’ does not mean decipherment is mandatory, but rather you can make heavy use of portions that are usable without decipherment.

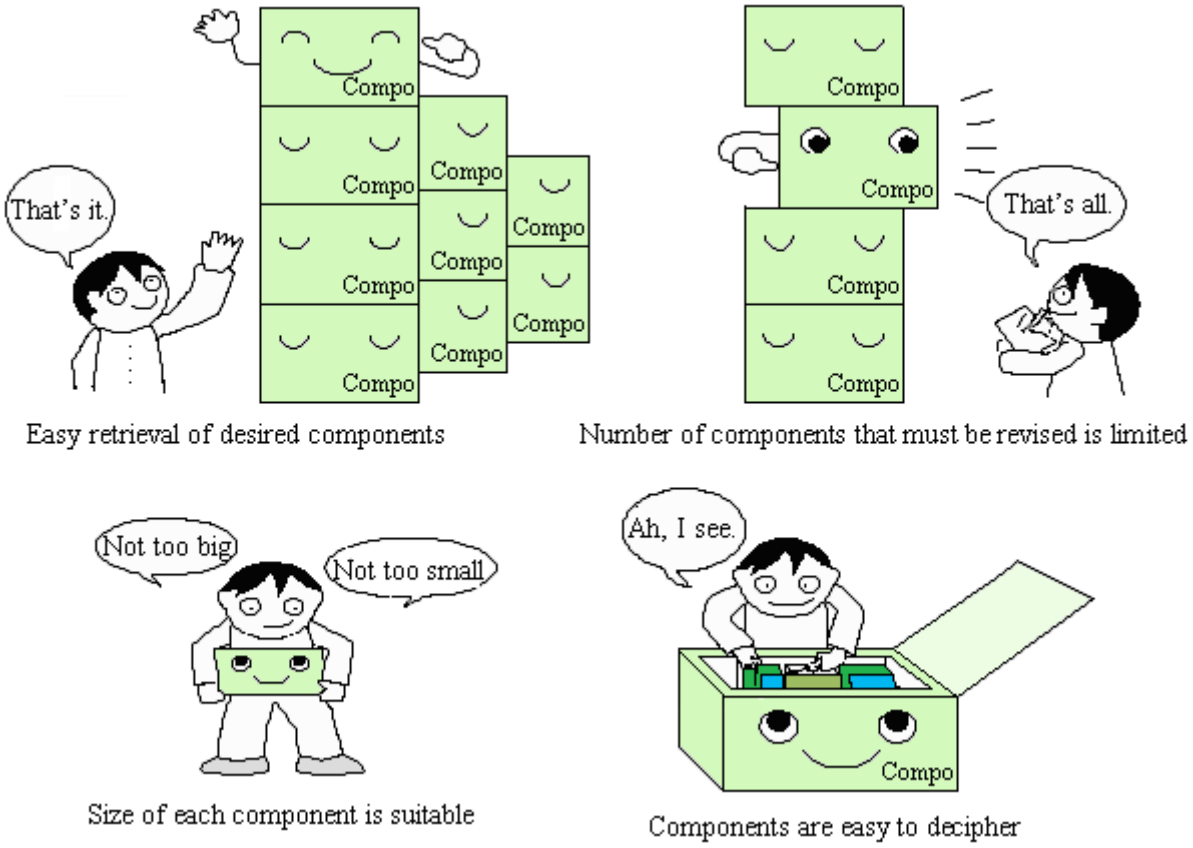


Figure 5-2: Qualities Required of a “White Box Component”

What I would like the reader to focus on here is the demand for readability as one of the qualities required of a “white box component,” which is not demanded of a ‘Black-box Component,’ because it does not require decipherment.

Note that “readability” includes not only “visibility” by means of structured programming, but also the quality of many things, such as clear-cut meaning and being easy to understand.

5.2-g Comparing RRR Family Construction Technique to a Generalized Construction Technique

Now that we have finished introducing the generalized construction technique for an RSCA, let’s take a look at this generalized technique in light of the construction technique for the RRR family. We will check whether the construction technique for the RRR family is in line with this generalized technique. By doing so, it will become clear that the generalized construction technique is substantive rather than a mere play on abstract terms. Furthermore, the use of an actual example of the RRR family makes it possible to provide a concrete image.

First, let’s look into the concrete meaning of the following statement within the generalized construction technique.

• Partition the area that software products cover into low NCA areas and high NCA areas.

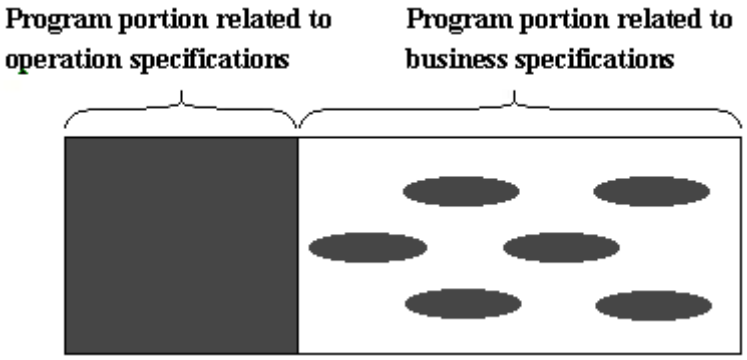
For the RRR family, we extracted portions where the potential for program customization was high according to the partitioning guideline of demarcation of business and operation. Specifically, we narrowed down the portions where the potential for program customization was high to only programs related to business specifications. Consequently, this can be seen as a rough partitioning of low NCA areas and high NCA areas to start with, according to the partitioning guideline.

It should be fine for program portions related to operation specifications partitioned in this manner, as a general rule, to be made the same for Company A and B, and the need for program customization can be considered quite rare. Therefore, such portions can be said to be low NCA areas.

At the same time, program portions related to business specifications may have common portions as well as company-specific portions, and these are portions where the potential for customization is high. Such portions can be thought of as high NCA areas or mixed areas.

If we assume that program portions related to business specifications are mixed areas, we should be able to focus on low NCA areas within this mixed area and extract them as general subroutines. This is nothing but the extraction of general subroutines related to business specifications as has been commonly done in the past.

As you can see from the above observation, low NCA areas in the RRR family are program portions related to operation specifications and general subroutines related to business specifications, and high NCA areas are program portions related to business specifications excluding general subroutines. Refer to **Figure 5-3**.



White portions represent high NCA areas, and they are covered by white-box components.

Black portions are low NCA areas, and they are covered by black-box components. For example, the black ovals in the program portion related to business specifications (NCA generally is high) are low NCA areas that can be covered by general subroutines (black-box components).

Figure 5-3: Areas in Which NCA (Need for Creative Adaptation) is High/Low

The concrete meaning of the following second statement of the generalized construction technique becomes clear once you understand the above-mentioned information.

• ‘Business Logic Components’ or ‘Software Components’ that have the following quality demanded by ‘Black-box Components’ cover all low NCA areas:

- Ability to meet all envisioned requests in areas that must be covered by selecting components from component sets and specifying parameters (generality).

The RRR family covers program portions related to operation specifications by using general main routines (see **Note 12**) and partially covers low NCA areas within business specification by using general subroutines.

Note 12: Just as generalizing a common subroutine results in a general subroutine, generalizing a common main routine results in a general main routine (a framework in the narrow sense). Note that, generalization means enabling the component set to meet any envisioned request by selecting components from component sets and specifying parameters. In other words, generalization is nothing but providing the qualities required of a 'Black-box Component.'

The above-mentioned second statement will be satisfied as long as general main routines and general subroutines have the qualities required of a 'Black-box Component.' In short, they are satisfying one of the required conditions that the RRR family comply with the generalized construction technique. Accordingly, it would seem best to investigate whether such components can really be called 'Black-box Components' that have generality.

It is great to have general main routines and general subroutines, but if we could completely build in what was necessary for providing the qualities required of a 'Black-box Component,' we should be able to meet any customization requests for these portions by means of parameter customization. Thus, we should investigate whether customization requests are being met in this manner. Frankly, there are few places where it is difficult to say we can meet customization requests completely, but we can say that the RRR family is more or less meeting customization requests by means of parameter customization. It is rare to have to customize the portions that we assumed to be low NCA areas. Consequently, we can say that general main routines and general subroutines are essentially 'Black-box Components.'

Furthermore, we also can come to understand the concrete meaning of the following third statement which is one of the generalized construction techniques.

• 'Business Logic Components' or 'Software Components' that have all of the following four qualities demanded of a 'White-box Component' cover all high NCA areas.
- Easy retrieval of desired components (retrievability).
- Number of components that must be revised is limited (locality).
- Size of each component is suitable (suitable size; suitable granularity).
- Each component is easy to decipher (readability).

The RRR family covers nearly all high NCA areas (i.e. portions where NCA is high within areas related to business specifications) by using data item components. Therefore, the above-mentioned third statement, which is a required condition for the RRR family to comply with the generalized construction technique, is true as long as data item components have the qualities required of a 'White-box Component.'

A look into this to provide a concrete image will reveal that data item components have the qualities required of a “white-box component” as discussed in the following paragraphs.

The quality of retrievability (easy retrieval of desired components) is one of the major features of data item components (this has already been discussed). Specifically, data item components are easy to retrieve because they are assigned names that begin with the name of their data item. In other words, they take advantage of the feature of specification change requests being represented by data item names in business fields. What I would like the reader to focus on here is the fact that the retrievability obtained in this manner is a naturally endowed quality of components and differs from forced retrieval using a component retrieval system.

The quality of locality (the number of components that must be revised is limited) refers to whether there are few data item components that require decipherment and revision when meeting customization requests. To have fewer of such data item components is obviously advantageous. Based on our experience with customization in the RRR family thus far, most of the time the extent of impact is limited to one or two data item components. Having to revise one hundred or two hundred components is very rare. One reason for this quality is related to the feature of specification change requests being represented by data item names in business fields. Another reason can be attributed to the extraction of components according to an orthogonal coordinate system in n dimensional space consisting of n data items. This would not be so if something that differed from the coordinate system of the specification change request were adopted as the coordinate system for component extraction. Decipherment and revision of many locations would likely be required.

To satisfy the quality of suitable size (the size of each component being suitable), components would have to be one hundred lines or less. Any larger and it becomes difficult to say whether decipherment would be easy or not. Based on the track record of RRR component sets, the average size of a single component is 55 lines. Since one data item component is comprised of two or three high-level event procedures (hook methods), it can be partitioned into blocks of twenty lines give or take a few.

The quality of readability (each component being easy to decipher) is determined by whether a component’s meaning is clear, whether its content is simple, whether it is stereotypical, and whether it can be understood independent of other components. It also includes “visibility” through structured programming. Since data item components fit into types, are stereotypical, and have clear meaning, once you have deciphered ten or so, decipherment for the rest will be easier. In addition, as long as you decipher that many, you will be able to understand them without knowing anything about other data item components. Consequently, data item components can be said to be easy to decipher.

Systematic component-based reuse is being implemented by the majority of developers because data item components have all of these four wonderful qualities (easy retrieval, limited number, suitable size, and easy decipherment). Over three hundred developers who do customization fulltime are actually using the RRR family and SSS in their customization work. These people are not related to the development of the RRR family.

Adopting partitioning with data item association, as one of the partitioning guidelines for the compartmentalization of components is what has made this possible. In other words, you could say that partitioning with data item association was by chance able to fully provide the qualities required of a “white box component”

Based on the three points mentioned above, the RRR family can be said to comply with the generalized construction technique for an RSCA.

Topic 10: Size of 'Business Logic Components'

The size of 'Business Logic Components' (or 'Software Components') must be suitable and easy to handle.

An extreme point of view is that it would be possible that each and every statement of a program is a primitive software component, but this holds no benefits whatsoever. Although it is possible for a software component, which is compartmentalized according to a certain ideal, to just happen to be a single statement, it is generally clear that software components will only help improve productivity if they are built out of multiple statements.

Nevertheless, they will be unsuitable if they are too big. Something as big as a single business program, for example a business package, should be called a finished product rather than a component. Only if the finished product is built from multiple software components will there be profit in combining semi-finished products.

Programs concerned with a single form or a report form should be called semi-finished products, but there is also the point of view that they are also business logic components. This point of view may seem believable, but there is a problem because no method has been considered for supporting the customization of what is associated with forms and report forms. Here it is required that forms and report forms be built from a number of small business logic components and be easy to customize. In short, there must be a hierarchical structure in which small business logic components build forms and report forms, and these in turn build business programs. If that is the case, forms and report forms can be called semi-finished components, otherwise they are nothing more than mere semi-finished products.

Software components can be classified as black-box components whose contents are not visible and white-box components (type where the source program is disclosed) whose contents are visible. It does not matter how big black-box components are because their content is not subject to decipherment. However, white-box components should be kept at around one hundred lines at most to ensure easy decipherment.

There are also opinions to the effect that size is not the only problem; content must also be closely examined. When you are in a situation where you are using white-box components and you extract one component, a weak, insubstantial component that is useless will be just as troubling as a complex, difficult-to-understand component that is useful. A good balance is important between "suitable size" and "suitable ease of use" for the content of software components.

5.2-h Comparing RRR Family to the Three Requirements

The fact that the RRR family is in line with the generalized construction technique for an RSCA is shown in the previous discussion. Based on that, we can say that the RRR family is fulfilling the three requirements for an effective system because we can deduce it through syllogism if we use the theorem provided in Appendix 5. Specifically, we can deduce this by using the theorem that says if a certain software product is constructed as a componentized application according to this generalized construction technique, it will certainly fulfill the three requirements.

Reaffirming this will simply be repeating our discussion, so in the following paragraphs we will reaffirm that the RRR family fulfills the three requirements one by one, while considering whether there are problematic points. We will try to be critical as possible.

- Software products must be built up solely by combining components.

General subroutines alone cannot satisfy this requirement. Accordingly, the RRR family fulfills this requirement by combining them with general main routines, which fulfill the roles of operability-related processes among other things, and data item components that cover high NCA areas.

The main part of the RRR family is actually comprised of the three components known as general subroutines, general main routines, and data item components. Since each is really a "software component," we can say that software products can be built from 'Software Components' alone.

Strictly speaking, however, there are also cases of accumulated data item components not being able to cover what is needed, necessitating the development of a number of new ones. In addition to data item components corresponding to a single data item object, there are, to a lesser extent, relation check components corresponding to multiple data items and form components corresponding to a form object. Whether these can be called 'Software Components' requires some debate.

If we were to make a comprehensive judgment of these, we should probably say that they more or less satisfy this requirement.

• The system must be able to meet all customization requests.

Fulfilling this requirement is no problem as long as it is for data item components for which you are resigned to performing program customization. However, general subroutines and general main routines that do not permit program customization require selective checking. An in-depth look into these will reveal that they basically have the qualities required of a 'Black-box Component' to some extent. Consequently, we can say that the majority of all customization requests can be met.

There are, however, unusual cases where the insides of 'Black-box Components' must be revised. For example, there are cases where components developed as 'Black-box Components' lack adequate generality, as well as cases where the need for creative adaptation rarely exists even if you are talking about low NCA areas.

Although there are cases that necessitate the revision of 'Black-box Component' content as mentioned above, the frequency of customization for such areas is low, and they can be handled without problems from a practical standpoint. Specifically, the best response would be to secure a developer who specializes in such customization. Furthermore, since the need for such a response occurs at a low frequency, there needs to be only a relatively few number of specialties. However, if no steps are taken, caution must be observed because of problems that may occur such as a "frustrating lack of freedom" that we introduced as a problem with 4GLs.

• Large numbers of developers must systematically benefit from component-based reuse.

There is no problem with fulfilling this requirement as far as general subroutines and general main routines that do not permit program customization are concerned. However, data item components that require program decipherment should be selectively checked. Accordingly, you will find that they have all of the qualities required of a "white box component." That is exactly the reason why systematic component-based reuse by a large number of developers is possible.

Besides data item components, relation check components and form components are few and far between, and it is somewhat debatable whether they can be called 'Business Logic Components' as we have already discussed.

5.2-i Historical Development of Component-Based Reuse Systems

If you look into whether 4GLs and visual development support tools are fulfilling the three requirements for a practical and effective component-base reuse system, you can verify the superiority of the RRR family.

Not even 4GLs or visual development support tools can fulfill all three of the requirements. They only fulfill about half of the requirement that "Software products must be built up solely by combining components." Although they almost totally fulfill the second requirement that "the system must be able to meet all customization requests," doing so will end up preventing the fulfillment of the third requirement that "large numbers of developers must systematically benefit from component-based reuse." In short, it is difficult for the second and third requirements to co-exist.

Based on our studies thus far, we have yet to find a component-based reuse system that can fulfill all three of the requirements, besides systems developed based on SSS tools. Therefore, we consider the RRR family to be on the cutting edge of component-based reuse systems.

Looking back at the history of component-based reuse systems, we can say that the range covered by 'Software Components' has expanded as shown in **Figure 4-2**.

It started with the use of subroutines. In the first stage, only 'Business Logic Components' in the form of general subroutines were used, and there was nothing else that could be called 'Software Components.' Consequently, the target for component-based reuse was limited to an extremely narrow range.

Next, fill-in systems, 4GLs, and visual development support tools opened the way for the reuse of main routines, thereby widening the range in which component-based reuse could be easily performed. We call this the second stage. The event procedures of 4GLs and visual development support tools, however, could not be called 'Software Components.'

After that, we enter the third stage in which data item components play a major role. For example, by adopting a componentized event-driven system, the RRR family has enabled the use of 'Software Components,' which are data item components (i.e. componentized event procedure) that extend the range general main routines cover and are cleanly partitioned per data item. This makes it easy to carry out component-based reuse for all software products.

5.3 Meaning and Significance of 'Business Logic Components'

This section will consider the meaning and significance of 'Business Logic Components,' not only from a technical viewpoint, but also based on the impact they have on custom business program development firms. We will also clearly show that 'Business Logic Component Technology' is effective not only for customization work, but also for general maintenance work.

5.3-j What are 'Business Logic Components'?

The first 'Business Logic Components' (in the narrow sense) were born when we were able to prepare them by partitioning them into modules during the development process for the SSS component set. The reason we gave the name 'Business Logic Components' was that we wanted it to be analogous to mechanical components where a problem spot is apparent at a glance and limited to a specific component, which means the problem itself will be readily understandable if you extract that component.

At first, the term ‘Business Logic Components’ indicated something extremely specific. Therefore, we strived to widen the meaning by pointing out the thing itself and emphasizing the fact that “such and such is a “business logic component,” but the others are not.”

Generally, when some sort of equipment is invented, such as a hardware device, it seems best to point out the thing itself. In that respect, ‘Business Logic Components’ too is a term pointing out the thing itself. However, the wording “this is what a business logic component is” was regarded as being against better judgment and self-righteous. The thing itself, i.e. source program, which we took the trouble of pointing out, was not understood without protest. People who manage development work cannot evaluate programs because they are not in the habit of reading them, and since developers are so proud of the fact that they can develop programs themselves, they are not interested in an approach that renders programming unnecessary.

Accordingly, I conducted a theoretical study and then explained it in this book so that people could understand even without looking at programs. Furthermore, this book clarified the conditions to construct component-based reuse systems as mentioned below when defining ‘Business Logic Components.’ In this manner too we opened a way for discovering ‘Business Logic Components,’ although there is no guarantee that similar things to SSS-style ‘Business Logic Components’ will be found. Consequently, a definition of a “business logic component” in a form that clearly specifies conditions will avoid criticism that it is self-righteous, and this might enable us to receive some sort of evaluation. I welcome any evaluations from my readers.

The definition of a “business logic component” in this book is as described hereafter. It is a slightly revised definition based on the “generalized construction technique for an RSCA” with which the reader is already familiar.

‘Business Logic Components’ are fragments (modules) that partition and prepare software products covering a certain area so as to fulfill the following three conditions:

Condition 1: Partition the area into a number of modules that cover low NCA areas (these are called ‘Black-box Components’) and a number of modules that cover high NCA areas (these are called ‘White-box Components’).

Condition 2: Prepare and enhance functionality so as to have the following quality for ‘Black-box Components’:

- Ability to meet all envisioned requests in areas that must be covered by selecting components from component sets and specifying parameters (generality).

Condition 3: Adjust and prepare partitions (i.e. refactoring) so as to have the following four qualities for ‘White-box Components’:

- Easy retrieval of desired components (retrievability).
- Number of components that must be revised is limited (locality).
- Size of each component is suitable (suitable size; suitable granularity).
- Each component is easy to decipher (readability).

In the final analysis, whether something is a “business logic component” depends upon the definition of a “business logic component.” Whether the definition is thought of as appropriate or not is ultimately a subjective and sense-based judgment made by each individual.

As much as possible, I would like people in general to accept that what we have defined in this book are ‘Business Logic Components’ without having to add the disclaimer “according to this book.”

Now we will carefully examine whether the above-mentioned definition is appropriate.

Since the definition is essentially the same as the generalized construction technique for an RSCA, we can deduce from it that the three requirements discussed earlier in this chapter are being fulfilled. Once you understand this, there is no doubt that you will consider this definition to be appropriate.

Conceivably, some may think that the conditions for this definition are too strict. Specifically, they may criticize the conditions by saying that there are SSS-style ‘Business Logic Components’ in business fields but maybe not in other fields, or there are only SSS-style ‘Business Logic Components’ in business fields. They may have doubts because they feel it is just like the old trick of a large organization wanting to buy a specific company’s product and a condition for eliminating other products. It is troubling when people say that, but to alleviate such suspicions even a little bit, it is probably better to make the definition in the form of requirements rather than conditions when building ‘Business Logic Components.’ Also, if the requirements are too strict, we should study to what extent they can be eased later on.

Based on such thinking, we tried to define a “business logic component” by indicating requirements as shown in the following paragraphs.

We call the components of software products synthesized by a component-based reuse system that fulfills the following three requirements, ‘Business Logic Components.’ The following three requirements are the “three requirements” you are already familiar with, and we use them to try to define ‘Business Logic Components.’

- Software products must be built up solely by combining components;
- The system must be able to meet all customization requests; and
- Large numbers of developers must systematically benefit from component-based reuse.

Even if we define ‘Business Logic Components’ in this manner, it may seem almost the same as the previous definition, but it may appear that the scope of ‘Business Logic Components’ widens only by the portion that is vague.

The ‘Business Logic Components’ that descended from SSS, which we have discussed in this book, fit in with both of the above-mentioned definitions. They also obviously fit in with the definition that further eases these requirements.

If so desired, the requirements can even be appropriately eased here.

For example, let’s try to limit the target to which the three requirements are applied to a certain portion of a software product, rather than the entire software product. In other words, let’s make the following changes. Such changes appear to be meaningful.

- Certain portions of software products must be built up solely by combining components;
- The system must be able to meet all customization requests for those portions of software products; and
- Large numbers of developers must systematically benefit from component-based reuse for those portions of software products.

Based on the above definitions, we will call the portions that can be covered by ‘Business Logic Components,’ among those software products, “component-enabled portions” By doing so; we will have component-enabled portions in varying degrees in all fields of application. In the business field, we can say that component-enabled portions are nearly one hundred percent. Furthermore, promoting component-based reuse can be said to be maximizing component-enabled portions in each field.

As an example of easing requirements, let’s say that the scope of the three requirements is the portions that can be covered by general subroutines among certain software products. Once we say this, it is obvious that the general subroutines considered to be business logic components in the past really are ‘Business Logic Components.’ When the definition is eased that much, it ends up being just like the old one, and thus there is nothing to be thankful for. Nevertheless, thinking this way enables us to confirm that the definition demanding the original strict requirements includes general subroutines within ‘Business Logic Components,’ and thus, it is in an extended form in order to be more effective. Still, easing the requirements in this manner limits the situations in which they are helpful, making them impractical. Consequently, I would like for them not to be eased too much.

I think the reader’s understanding will grow in light of that. Wouldn’t you give your approval if you could truly accept as the general definition for ‘Business Logic Components,’ a definition in the form of the above-mentioned conditions, a definition in the form of requirements, or a definition that demands maximizing component-enabled portions?

Topic 11: Are ‘Business Logic Components’ Modules?

One person who read a rough draft of this book asked whether ‘Business Logic Components’ were program modules. Since both are fragments into which programs are partitioned, I answered with a conditional yes. Originally, modules were intended to be combined in order to develop software as if making a mosaic. Actually, software is comprised of a number of modules. As a result, there is no doubt that ‘Business Logic Components’ in this book are a type of module.

The reverse, i.e. modules are ‘Business Logic Components,’ is not necessarily true. Generally, module partitioning has a high degree of freedom. What to make a module is basically up to the developer, and thus a module is born as a result of arbitrary development. Conversely, a “business logic component” is something that is skillfully partitioned and prepared so as to fulfill strict conditions that are suitable to it. You could say that no matter who partitions modules, they will become blocks just the same, as long as those conditions are fulfilled. Module partitioning conditions are that strict and clear. Therefore, it is hard to think of freely partitioned modules as being the same as ‘Business Logic Components.’

In retrospect, arbitrarily leaving module partitioning to each developer is not a good idea, and that is why partitioning guidelines are created at the beginning of a development project. Thus, we can say that ‘Business Logic Components’ will result if we carry out module partitioning after adopting the qualities of ‘Business Logic Components’ in this book as guidelines. Generally speaking, however, module-partitioning work can hardly be expected to achieve its aims because it has been carried out haphazardly thus far. To better understand this, try imagining a situation where a work instruction for module partitioning is given. I would like the reader to stand in the shoes of a development manager or developer and try imagining what the outcome would be according to the following order. Here, let's clearly define the guidelines.

The special feature of ‘Business Logic Components’ in this book is the covering of high NCA areas with ‘White-box Components.’ ‘Black-box Components,’ the other form of modules, are no particular change

from the conventional partitioning method used when extracting general subroutines. Therefore, we will focus on 'White-box Components' here. Since the following four statements are qualities required of a "white box component," we will use them as our guidelines.

- Easy retrieval of desired components (retrievability).
- Number of components that must be revised is limited (locality).
- Size of each component is suitable (suitable size).
- Each component is easy to decipher (readability).

Accordingly, let's try to imagine a scene where a person in the position of a development manager who points out these four qualities to developers and then instructs them by saying, "Make the fulfilling of all the qualities their partitioning guidelines for module partitions so as to make customization work easier." There is one important matter that should be considered when doing so. We should note that rather than simple partitioning, an ingenuity, such as a special technique, preparation, or refactoring is crucial. This is often neglected despite its significance.

The most likely response to this instruction would be for a developer to say, "I don't have the confidence to skillfully partition even if I am told about such an ideal, but I'll try to make an effort." It is actually extremely difficult to fulfill all four of these qualities. Generally speaking, there is no guarantee that partitioning that fulfills them is even possible. Therefore, it appears to be considered as a mere ideal that is impractical. It is also highly likely that the developer will quit after only a halfhearted effort and say, "I tried, but it's just impossible, so give me a break!" But guidelines are after all just guidelines. Since it is not customary for development to strictly follow guidelines, you can imagine the result when developers consider them to be impossible.

To begin with, conforming to the above-mentioned guidelines requires experience with and knowledge of customization requests. As a result, most developers who do not have this experience just give up. It is probably seen as a form of bullying, like being forced to deal with impossible problems.

However, after receiving the above-mentioned instruction, a developer just might say, "I did it" after succeeding in partitioning and preparing modules that fulfill all four of the qualities, although this is a most unlikely response. This book's standpoint is if the result were to fulfill the above-mentioned guidelines, then it would be nothing other than a 'White-box Component.'

Actually, only one partitioning technique that fulfills these guidelines has been discovered, and that was the result of repeated trial and error over more than a year while trying to provide component-like qualities during SSS component set development. This was like trying to strike gold that was unlikely to be found. This was a major success, although the applicable scope was limited to the business field.

It is generally very difficult to carry out ideal module partitioning that fulfills the guidelines as mentioned above. There is little hope of discovering a good method without extensive consideration over at least a year. Consequently, the custom of carrying out module partitioning each time a business system is developed is mistaken. Once an ideal solution to module partitioning is found, that solution must be reused. 'White-box Components' in this book indicate components for which module partitioning and preparation has already been completed. Therefore, to enhance a lineup of 'White-box Components,' we simply must create them in line with the partitioning method. This is not particularly difficult to do. Devising the ideal module partitioning method is difficult, but reusing that method is easy.

Based on this, 'White-box Components' should be perceived as distinct from modules that do not have any special benefits. The adoption of event procedures would be a better comparison.

For event procedures, module partitioning is determined by the combination of object and event types. This aspect is similar to 'White-box Components.' In addition, enhancing an event procedure lineup is not very difficult and only requires creating event procedures in line with the designated partitioning method. This aspect too is similar to "white-box components." However, ordinary event procedures do not fully have the qualities required of a 'White-box Component's. That is where the difference from 'White-box Components' lie, and the reason why ordinary event procedures could not improve the productivity.

Note that a 'White-box Component' layer can be established on an event procedure by revising the event architecture for event procedures and building in a mechanism for update propagation. In short, a componentized event-driven system can be created. **MANDALA**, which serves as the core of RRR tools, can be called a component synthesis tool that integrates and synthesizes 'White-box Components' as previously mentioned into programs that actually run.

5.3-k Near-Future Image of 'Business Logic Components'

Due to differences in industry and business, there are several hundred business programs in business fields, but if it were possible to develop one business package with special customization facilities per application field, custom business program development firms and business package development firms could rationalize from the ground up. Furthermore, the development of custom business programs for each and every enterprise due to subtle differences in business procedures between customers would no longer be permitted. This is because a component customization on a suitable business package with special customization facilities would suffice, thereby making it possible to supply a perfectly suited business program at a reasonable price.

Actually, it is likely that business programs that are even better suited than before would be demanded. To realize the effect of really utilizing a computer, you must carry out functional tuning while breaking in the business system as discussed in "**Topic 4: End-User Development and the Spiral Model**" in Chapter 3. Until now, developers could not handle such tuning because of all the redundant, wasteful development, but from now on; they will have the luxury of doing so. In light of this, we can predict that customization firms will undergo a massive transformation into tuning firms, thereby forming a new market.

A lineup of business packages with special customization facilities is required before this can happen. The creation of this lineup is work in which business programs are reconfigured using 'Business Logic Components.' This work may seem like a major undertaking because it must target several hundred-business programs, but it appears that the lineup can be completed more quickly than previously thought because of the following three points.

First, the size of the programs that must be developed will decrease (by about one-half or less) because of 'Business Logic Component Technology.' In the past, it was usual for COBOL programs to mix together portions related to operation and portions related to business. The use of general main routines will thereby eliminate the need for programs related to operation, which means all a developer has to do is prepare portions related to business as a program (i.e. collect and incorporate portions that have already been developed or develop new ones). Doing so raises development speed just like using a commercially available 4GL.

Second, some 'Business Logic Components' can be used across multiple fields of application. Just as some subroutine libraries could be used in financial accounting, production management, and sales management in the past, some data item components related to business can be commonly used in a number of application fields, and this can accelerate advances into various fields.

Third, a mad dash in many development projects is expected, and this is likely to have the greatest effect. Since Japan has far more software development firms than the kinds of business programs that are necessary, the lineup will be completed in one to two years if each firm develops one program.

A lineup of business packages with special customization facilities will be completed surprisingly fast because of the above-mentioned three points and this will conceivably rationalize custom business program development firms and business package development firms from the ground up. In reality, not one but several business packages with special customization facilities will likely be developed per application field. Also, once competition in business packages with special customization facilities begins, firms will not be able to even participate with such competition unless they employ 'Business Logic Component Technology.'

From a primarily technical viewpoint, this is expected to happen quickly. However, it is also true that there are negative feelings toward the rationalization of software development out there. These feelings are not out in the open like the protest movement against mechanization in the early stage of the Industrial Revolution, but rather they are a touchy matter simply because they are deeply woven into people's psyche. Consequently, a slow change is most likely. In particular, the awareness of executives and managers at business program development firms is a crucial key in deciding the pace of change. The pace of change will pick up speed only after they realize that merely gathering people and accepting work is not enough.

Predicting the pace of change in this manner is difficult, but rationalization will surely proceed at any rate. Furthermore, incorporating 'Business Logic Component Technology' as soon as possible will surely lead to high profit and high growth.

5.3-l Customization and Maintenance

Thus far, this book has clearly stated that the reaping of the benefits of 'Business Logic Component Technology' has been due to customization work, in the broad sense, at custom business program development firms and business package development firms in the business field. However, 'Business Logic Component Technology' is similarly beneficial even for general maintenance work.

The software assets retained by large enterprises consist of millions of lines, and the burden of maintenance work for them is becoming a routine problem. Maintenance work is often carried out in-house because it is difficult to contract externally. One theory holds that seventy percent of information department employees are engaged in maintenance work. Alternatively, the cost of tying down a related software development company accounts for a major percentage of fixed costs. A maintenance hell is a word that represents such a dead-end state at an enterprises' information department. Note that maintenance work can include things like debugging, revisions accompanying business content changes, and miscellaneous improvements, but here we are mainly talking about non-debugging maintenance work.

Customizing a business program for a certain industry/business field into a version for Company A and another for Company B is very similar to maintaining a business program for a certain enterprise to create a Month A version and a Month B version. The former customization work is a spatial expansion while the latter maintenance work is a temporal expansion, and therefore, they are essentially the same; it is only the difference between space and time.

By replacing the word customization with maintenance, this book can be changed into a book about improving maintenance work. The problems and solutions when large numbers of developers get systematically involved in trying to meet various customization requests are similar to the problems and

solutions when people besides the original creators (original developers) get involved in trying to meet various maintenance requests.

Still, reaping the benefits of 'Business Logic Component Technology' requires a switch to the refined asset architecture of 'Business Logic Components.' Unfortunately, this technology is not a magic cure for current maintenance hell. Instead, it is a specific cure for internal improvements at present and the prevention of maintenance hell in the future. Consequently, the sequential application of 'Business Logic Component Technology' from development in the works is practical. For example, we recommend it be applied along with a migration to an open system.

Finally, let's touch on the subject of NCA (Need for Creative Adaptation) in relation to the exchange of time and space. At information departments, there are anguished voices saying such things as "why is so much maintenance necessary?" Actually, this too is related to NCA. The explanation that new customization requests occur each time a new customer (enterprise) is encountered when NCA is high is a spatial perception. A temporal perception results in new maintenance requests occurring as time passes when NCA is high. This is a phenomenon seen in highly competitive industries and businesses subject to rationalization. The special feature of 'Business Logic Component Technology' in this book lies in delving into high NCA areas and then using 'White-box Components.' Accordingly, we recommend the prescription of this technology based on a long-term vision as a specific cure for maintenance anguish.

CHAPTER 6 Evolution of Life and Component-Based Reuse

Complex systems have become a popular topic of late. In the world of physics in particular, reductionism had been the prevailing concept. It held that we could come to know everything by breaking down matter into its components. However, many people realized that this was not true. They began focusing on the fact that when simple components come together to form an organized system, they will end up performing previously unimagined functions. A complex system is a system in which such phenomena occur.

Within research on complex systems, there is the field of artificial life, which seeks to unravel the mystery of life. Within this field, research on the mechanisms of evolution and other subjects is being conducted by equating **DNA information** (genetic information) of life forms to information in computer memory.

Based on the idea that the relationship between **a program and its functional specification** is analogous to the relationship between **DNA information and an individual organism**, we will refer to the mechanisms of evolution and related subjects as we ponder component-based reuse.

Since this chapter basically consists of a single topic that reinforces this book's assertions, it should make for some light-hearted reading.

6-a What is Darwin's Theory of Evolution?

DNA had not yet been discovered in the days of **C. Darwin**, but in this section, we will start by using recent knowledge on DNA to review Darwin's theory of evolution.

Generally speaking, an individual organism is created according to DNA information, which is equivalent to a blueprint. For the sake of making our description clearer, we will postulate the existence of a biogenesis machine at the risk of somewhat neglecting factual accuracy. Biogenesis using this machine would produce a monkey with the input of a monkey's DNA and a human with the input of a human's DNA. As you can see, organisms with different characteristics would be produced depending on the DNA information that is input. In other words, DNA information would serve as a blueprint for the resulting organism. Based on this, an individual organism could be called a genetic **phenotype**. Now if we say that a program phenotype is its functional specifications, then we can equate the relationship between programs and functional specifications with the relationship between DNA information and individual organisms.

A phenomenon known as **mutation** plays a crucial role in Darwin's theory of evolution. A mutation refers to an extremely rare mistake that can occur when DNA information is copied from parent to child. Since a mistake in copying results in different DNA information, the characteristics of the individual organism (genetic phenotype) will differ from that of the parent.

However, most organisms are conceived via sexual reproduction in which they inherit half of their DNA information from their mother and the other half from their father. Subsequently, the DNA information that the child inherits does not fully match that of the mother or father, and neither does the child's characteristics match those of the mother or father. Nevertheless, half of the DNA information is a copy of the mother's and the other half is of the father's, so the child's characteristics will still resemble those of the mother and father.

The mechanism of sexual reproduction plays a crucial role in accelerating evolution. However, we would like to set aside sexual reproduction for a while so as not complicate our discussion. Focusing on simple asexual reproduction instead of sexual reproduction will help us highlight the essence of Darwin's theory of

evolution. This is because one hundred percent of the parent's DNA information is copied to the child in asexual reproduction, which means excluding exceptional cases, the DNA information inherited by the child will exactly match the parent's. In other words, the child will be a clone of the parent. Consequently, mutations will play a central role in DNA information change, and this will allow us to get closer to the essence of Darwin's theory of evolution.

Let's consider the occurrence of copy mistakes (mutations) that cause partial changes in DNA information. As you can easily imagine, there are cases where DNA information is meaningless. Therefore, inputting such DNA information into our biogenesis machine may result in an error, causing the generation of the individual organism to fail. On the other hand, there will also be cases where generation may succeed with no errors, even when copy mistakes occur.

In cases where no errors occur, the inputting of the DNA information will generate an individual organism from our biogenesis machine. This individual organism may or may not be adapted to the environment in which it lives relative to other life. There will be an overwhelming number of cases of not being adapted. However, there will also be extremely rare cases of the individual organism being adapted. In such cases, the offspring of the individual organism that received modified DNA information due to copy mistakes should flourish.

Darwin's theory of evolution says that all life resulted from such a mechanism. In other words, when copy mistakes in DNA information occur and happen to result in an individual organism with a phenotype that is well adapted to its environment, the organism's offspring will flourish.

There is criticism voiced against Darwin's theory of evolution that it states extremely obvious facts or that it is tautology and really says nothing of consequence. The offspring of organisms with high environmental adaptability will obviously flourish, and conversely, you could also say the characteristics of flourishing organisms are generally highly adapted to the environment. Consequently, Darwin's theory of evolution is criticized as not really being a theory at all. However, the essence of Darwin's theory of evolution lies somewhere else. Specifically, it elucidated the mechanism by which all life results by the accumulation of sudden copy mistakes known as mutations.

6-b Evolution by Copy Mistakes?

Darwin's theory of evolution can be called a general theory that can also be applied to inanimate self-replicating systems. If a mutation occurs in a self-replicating system, something different from what was there before (a mutant) will result. Furthermore, if we apply some sort of filter, such as environmental adaptability to the newly resulting mutant and the previously existing type, only offspring of one or a few types will survive.

Based on this logic, you can understand how evolution might work in this manner, but Darwin's theory of evolution is actually something that is not easy to believe.

In the case of machines and software, there is always a designer. However, there is no designer for living creatures, such as human beings, which are vastly more complex than machines or software. It is not easy to believe that such complex human beings resulted from accumulated copy mistakes. Mutation is nothing but a completely oblivious designer lacking rhyme or reason. It is like changing designs based on the roll of the dice. You would think that nothing good could result from such randomness, no matter how many mutations accumulated over time.

I have been involved in software development for some forty years but have never directly experienced nor heard of program copy mistakes resulting in a program better than the original.

Since leaving this matter unresolved felt unsatisfying, I went hunting for books about the evolution of life and developed an appreciation for biology. There is probably one other reason for my appreciation for biology. At that time, I felt that program development was terribly hard work, and thus if programs could come into being spontaneously according to the theory of evolution, it would make development work much easier, and that fueled my audacious desire to imitate such a mechanism.

6-c Natural Selection is Believable

If you look through actual examples of evolution, you will find reports of a phenomenon known as industrial melanism wherein the number of black moths became more numerous than white ones within a certain species during the mid-19th century industrial revolution in Liverpool, England. This phenomenon was the result of the destruction by atmospheric pollution of the white lichens that lived on tree bark. The coloration of black moths helped them hide and survive on the black tree surfaces that had been exposed. That is natural selection without a doubt. There is also a detailed report on the statistically significant change in the beak width of Galapagos Finches from an average of 8.86 mm to 8.74 mm due to the 1982 to 1983 El Niño effect. The size distribution of the seeds they eat changed due to environmental changes, and natural selection acted on beak size to make it easier for them to eat the seeds.

These are examples of the mechanism of natural selection, not mutation, within evolution. Moreover, they are examples of natural selection for life that reproduces sexually. In the former example, there originally were genes for white wings and genes for black wings in the moth species, but the individual organisms with genes for black wings increased. In the latter example, there was conceivably some sort of value for getting a gene that decides beak size in Galapagos Finches, but the individual organisms that had genes with values suitable to the environment at that time increased. Such change is in fact evolution, and the accumulation of such subtle differences results in major changes in life.

C. Darwin thought of natural selection as being analogous to **artificial selection** performed by livestock breeders. Artificial selection performed to improve livestock quality is the singling out and subsequent increasing (breeding) of individual organisms that have gene combinations that are advantageous to humankind. Although **natural selection** has nothing to do with what is advantageous to humankind, it is nothing more than the singling out and subsequent increase of individuals with gene combinations advantageous for survival. Consequently, we can truly say that natural selection is taking place in the two examples of evolution mentioned above. Strictly speaking, gene frequency change occurring within a species that shares a **gene pool** is an example of **microevolution**.

Natural selection and artificial selection are similar to parameter customization. This is because parameter customization is the selection and subsequent establishment of parameters advantageous to the customer.

The following process would result if we were to carry out parameter customization in a manner similar to natural selection and artificial selection. We would start by creating a particular combination of parameters consisting of a parameter set established for Company A and another established for Company B. It would be like creating child parameters with half the genes for Company A and the other half for Company B and then offering them to the customer. If the customer was satisfied with the parameters, the process would be complete, but if they were not, we would have to create another combination of parameters to offer them. If the combination was still unsatisfactory, we would create another combination of parameters by mixing in a parameter set established for Company C. This could go on and on until the

customer was satisfied. We would create an enormous number of child parameters and then have the customer evaluate them. If the customer had enough patience to stick with the evaluation process, this method would be able to identify the combination of parameters that was the most advantageous to them.

The normal method for establishing parameters, that is to say asking customers about each and every parameter, may seem better, but the ability to establish them by the above-mentioned method, modeled after natural selection, could also be believed. Nevertheless, as soon as I came to believe this, my unreasonable dream was shattered. This was because there were probably no customers who would be willing to continue patiently evaluating parameters in such a manner, and I thus realized that it would not be possible to spontaneously create a program by imitating the mechanism of the theory of evolution.

Even if it were possible to make clients more patient, deciding on a certain adaptability function and then having them evaluate it on a computer would enable us to determine the optimal values by imitating natural selection and artificial selection. This is known as a **genetic algorithm**, and it is actually applied to optimization problems among other things. For example, the traveling salesman problem — a problem in which you try to find the shortest route for a traveling salesman to visit all designated cities and return to his starting point — presents too many possible combinations for one-by-one comparison, and even computers cannot deal with it. Genetic algorithms can sometimes be effective for such problems. Furthermore, genetic algorithms would likely be effective for the problem of deciding the shape of a bird wing by adjusting various parameters until the most efficient design is achieved. This is because it is thought that birds evolved efficient wings through natural selection, the force behind genetic algorithms.

By thinking in this manner, I was able to believe that evolution by natural selection is entirely possible.

6-d Evolution by Copy Mistakes After All

One day when considering customization of business packages, I starting thinking, “There is also evolution by copy mistakes,” and at that moment I became a believer of C. Darwin. This might be the product of my appreciation for biology or due to an altered way of thinking about programs.

We face not only the problem of selecting optimal combinations from among many parameter combinations, but also the problem of having to create new things to find a solution. It could be compared to the inexorable existence of problems that would be difficult to use as true/false test questions. In this book, we have discussed how **low NCA** (Need for Creative Adaptation) **areas** can be solved using parameter customization, but **high NCA areas** require program customization. Program customization is nothing but finding a solution by creating a new program.

Measures, such as natural selection and artificial selection that are equivalent to parameter customization, are at a loss when dealing with problems that require such new creation. Therefore, some sort of creative activity is required. I also realized it was inconceivable that anything other than mutation is what carries out creative activity, and thus became a believer of C. Darwin. Generally, acting creatively requires wild ideas.

However, the creation of things normally is a very rare occurrence, and such things must therefore be carefully nurtured. Of course when a newly created thing is worthless, it will be abandoned through the mechanism of natural selection. But things that even have a little worth should be retained, and outstanding things should be reused. C. Darwin was saying that outstanding things passed through the filter of natural selection and were widely reused. This is believable.

Now what about the fate of mutation in the case of creation that is neither very good nor bad?

Dr. Motoo Kimura's **neutral theory of molecular evolution**, which has already gained acceptance worldwide, maintains that when mutations neither favorable nor unfavorable to an individual organism occur, **genetic drift** will take place, causing random changes wherein such mutations will either spread to the entire species or fade away. Therefore, mutations that are neither favorable nor unfavorable will be retained only for a certain period of time within the species' gene pool, or in other words, the component library. They also serve to expand the choices for parameters. Note that the two-chromosome redundancy (i.e. two pairs of DNA information) in each cell can be said to support the formation of a gene pool.

During times when the environment remains constant, mutations that are neither favorable nor unfavorable to an individual organism will settle into the gene pool, thereby enriching its diversity. Then when the environment changes drastically, such mutations will move into the spotlight though natural selection, be culled out, or be retained. The meaning of this will become clear if you consider the following. Mutations invent (create) ways to deal with a variety of problems ahead of time, and if they can be retained, they will help in swiftly dealing with any environmental changes that occur. In short, this can accelerate evolution because mutations that might be advantageous at a later time have already occurred and have been retained. This is far better than the random occurrence of mutations at the exact time they would be advantageous. If such advantageous mutations are retained, the mechanism of natural selection will do a fine job of spreading them throughout the gene pool through copying in a short period of time if the need arises. Consequently, this advance preparation enables adaptation to new and severe environments far more quickly rather than waiting for mutations that might happen at any time.

Because C. Darwin so overbearingly pointed out the role of creativity in evolution, I had a hard time completely believing it. However, I became a believer of C. Darwin by thinking in the manner discussed above. In the final analysis, you can say that the copy mistakes known as mutations are what play a crucial role in "creation" for evolution.

6-e Speed of Evolution and Component-Based Reuse

The self-replicating system of life that uses DNA information is said to have existed in the Earth's oceans 3.8 billion years ago. However, any attempts by humankind to make a machine that self-replicates would be far more complex than building a jumbo jet. There are various theories as to how such a complex replicating system spontaneously came into existence, and they form one genre of biological interest. If we switch the focus from machines to software, you can see how making something that self replicates would be comparatively easy. Specifically, computer viruses are self-replicating software in the true sense, and 60,000 kinds have already been created. Moreover, several new ones are created each day. You can understand just how indescribably advantageous the molecules that comprise organisms are as the building material for a self-replicating system after comparison to machines and software that self-replicate. However, opinion is still divided as to the origin of such life. Therefore, this book will not delve any further into the matter. We will instead just assume there was such a self-replicating system, just as C. Darwin did. Let's take an entertaining look at the evolution of life.

Note that the speed of evolution we are talking about here is how fast organisms adapt to an environment. Evolution is nearly synonymous with change in the strict sense and is not necessarily related to progress. Therefore, the transformation of lower animals into higher animals, such as humans, is not evolution. That is why we will consider the question of what needs to be done to rapidly adapt to an environment. Specifically, we will focus on the speed of customization.

The explosive diversification of life is said to have begun 600 million years ago in the Cambrian period. This is distinguished from the Pre-Cambrian period in which evolution proceeded gradually for the previous 3.2 billion years. There are various theories as to why diverse life appeared in the beginning of the Cambrian period, but I think that some mechanism for accelerating evolution was acquired (evolved).

One method for speeding up evolution is DNA information exchange. Evolution, by the combination of simple asexual reproduction and mutation, only proceeds sequentially, but with the exchange of DNA information between individual organisms, evolution proceeds in parallel and thereby accelerates. For example, to create something new in an environment wherein no information exchange with other people is possible, you must think up all the necessary ideas yourself, but if it became possible to exchange information with other people, you could reuse their ideas and thereby speedup your work. It is therefore clear that not only favorable mutations that occurred in a particular individual organism, but also the blending together of favorable mutations that occurred in other individual organisms will accelerate evolution if mutually favorable qualities can be acquired.

The exchange of genetic information is performed by blending DNA information. One of the mechanisms for blending together DNA information in unicellular organisms is a type of sexual reproduction known as **conjunction**. Another such mechanism is **transduction** in which DNA information is horizontally transmitted from one cell to the cell of another individual organism, sometimes even crossing the species barrier. Transduction occurs when a bacteriophage, known as a virus, which can infect bacteria in addition to other organisms, takes in DNA information from part of a certain cell, and then incorporates itself into another cell. There are a variety of such mechanisms that blend together DNA information, but I really have no idea which ones functioned efficiently in any particular period. The only definite thing that can be said here is that these mechanisms served to accelerate evolution.

Unicellular organisms that acquired such a mechanism diversified explosively as multicellular organisms at the beginning of the Cambrian period. In short, evolution took place. The fact that multicellular organisms evolved from unicellular organisms and took on various forms is just as we explained using the biogenesis machine. That is to say, if we input a variety of appropriate DNA information into the biogenesis machine and then control the machine accurately during generation, diverse forms of organisms will be produced. As you are well aware, multicellular organisms employ the mechanism of sexual reproduction to blend together DNA information. Furthermore, a phenomenon known as chiasma wherein the DNA information received fifty-fifty from mother and father is truly blended together takes place.

Gene duplication, which was put forward by Dr. Susumu Ohno as another mechanism that accelerates evolution in addition to the blending of DNA information, is thought to actually have an effect. Gene duplication says that if a copy of an effectively functioning gene is made ahead of time and a mutation later occurs on that copy, it will be easier to get usable functionality. This resembles one method of component customization discussed in this book, and I can identify with it. For example, you could say that component customization in which we copy the “general customer code component” and then change part of it to create the “important customer code component” is an application of gene duplication. Dr. Susumu Ohno’s well-known idea, which says something to the effect that similar to the evolution of life, the advancement of human culture has been dependant on the plagiarism of a small number of creative works, resonates deeply with the promotion of component-based reuse.

Note that for gene duplication, a mechanism for duplicating and copying part of DNA information is required rather than making an exact copy of what is transmitted to offspring. We can also predict its

existence based on the need for a DNA information editor. In fact, a mechanism equivalent to an editor that cuts and pastes DNA information is known to exist.

If we were to reduce the history of the Earth thus far into a single year, life would emerge around March, the appearance of multicellular organisms along with the explosive diversifications of life in the Cambrian period would occur in mid-November, the extinction of dinosaurs and the rise of the previously inconspicuous mammals would be in the afternoon on December 26, and humankind would first appear in the afternoon of December 31. Incidentally, computers do not even have a 150-year history equivalent to one second on this timeline. Life at some point acquired a mechanism that accelerated evolution, or in other words, it evolved some sort of tools for increasing efficiency and also reused the products that were created, and I cannot help feeling that this later resulted in an acceleration of the speed of evolution.

Rather than the establishment of DNA as a medium of creation, it was the evolution of DNA information that enabled the formation of a brain that could store and rapidly process information using neurons that guaranteed the acceleration of evolution. In short, the evolution of intelligence in animals epitomized by human beings produced new and different evolutionary trends. Specifically, in addition to genes formed from the medium of DNA, **meme** that formed from the media of neurons appeared and started evolving almost totally independent of and unrelated to genes.

According to R. Dawkins, who honed one aspect of the essence of evolution with his expression “selfish gene,” a meme is information that is copied through sensory organs and uses neurons as its medium. Examples of **meme phenotypes** are the opening of milk bottles by the Great Tit and potato washing by the Japanese Monkey. There are many products of culture that could be listed as the meme phenotypes of human beings including swimming styles, agricultural methods, language, religion, flight using hang gliders, the secrets of yacht racing, and programming methods that emphasize plain productivity.

I hope when I use the term meme and speak my mind, the memes based on the content of this book will get on the vehicles known as neurons of many people and flourish, thereby replacing the meme emphasizing plain productivity.

6-f High-Correspondence Portions and Low-Correspondence Portions

We have so far discussed not only how the occurrence of mutations is useful and crucial to accelerate evolution, but also the mechanism for editing and exchanging DNA information, and the function of gene pools (libraries) that retain DNA information. Now I would like to state my hypothesis that the correspondence between DNA information and its phenotype is also crucial to accelerating evolution. I would like to direct attention to the structure of programs themselves rather than simply focusing on tool types so to speak. To simplify the discussion here, let’s consider the case of asexual reproduction.

Focusing on the relationship of a program or DNA information and its phenotype, and then classifying them into high-correspondence portions and low-correspondence portions, will bring into view something very interesting.

It is thought that in DNA information there are **high-correspondence portions** that are easy to map to the individual organisms’ organs, which are a phenotype of that DNA information, as well as **low-correspondence portions**. For example, a genetic map for Drosophila, which is often used in heredity experiments, has been created; enabling scientists to know which part of a gene corresponds to Drosophila eyes, mouth, wings, or other body parts. Such parts can be called high-correspondence portions. Similarly, even in organisms for which no genetic map has yet been created, we can say that certain parts of their

DNA information will be high-correspondence portions that clearly correspond to eyes, mouth, and other body parts. However, it seems no one yet knows what part of DNA information is related to the mechanism that controls the generation of Drosophila from egg to adult. Since the correspondence between DNA information and the generation process is complex, I think research in that area will make little progress.

Programs could also be divided into high-correspondence portions and low-correspondence portions. Refer to **Figure 6-1**. **High-correspondence portions** can be easily associated with output and behavior (i.e. functional specifications) that can be observed externally, and they are simple to maintain because they make it easy to know what part of a program to fix when trying to change functional specifications. On the other hand, **low-correspondence portions** are hard to maintain because they make it difficult to establish an association with functional specifications.

A portion containing code for messages sent by the program can be called a high-correspondence portion as a general rule. However, when messages are assembled in a complex manner, low correspondence is the norm for programs that carry out such processing, and therefore, the message portions too are low-correspondence portions for the most part.

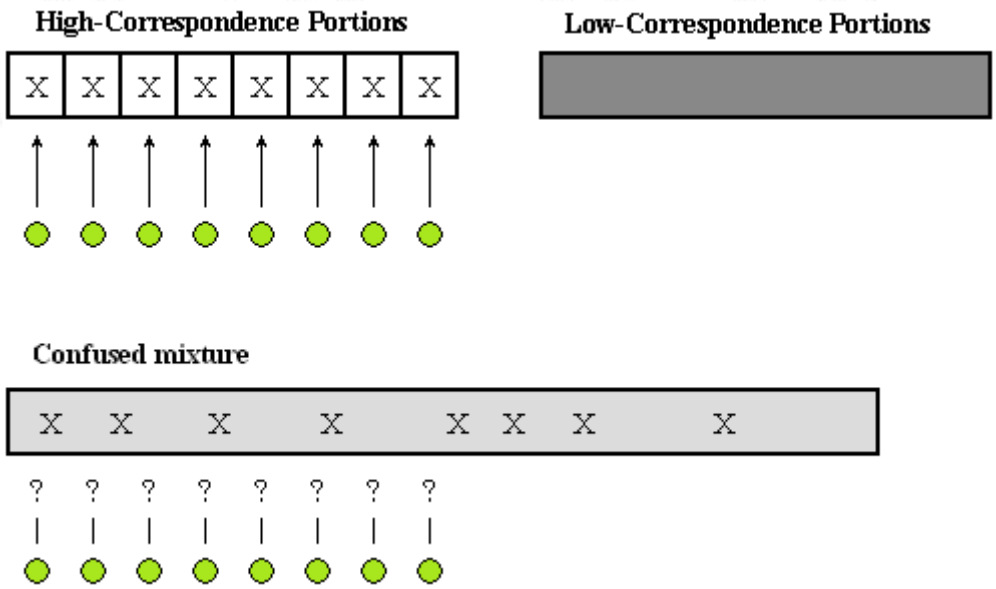
On the other hand, maintenance will become easier if you place all the messages sent by the program in one location and arrange them in an easy-to-understand order, such as alphabetically. This is an example of making high-correspondence portions clearer.

However, many programs can be created to fulfill the same function. That is why programs vary endlessly and become something different due to developers' ideas and programming methods.

Incidentally, this book explained that it is generally difficult to decipher programs created by someone else, but the basis for this lies in the fact that programs are rich in variety. If you assume programs fulfilling the same function would be the same no matter who creates them, there would be no difference between a program you created and one created by someone else, and thus you could decipher the other person's program as easily as you could your own. But the truth is no two programs are the same.

Since even programs fulfilling the same function vary endlessly and have all manner of forms, you will end up in deep trouble unless you make a conscious effort to separate high-correspondence portions and low-correspondence portions. Without such an effort, high-correspondence portions and low-correspondence portions will end up getting mixed together. Even in information theory, there is a law of increase in entropy. When no attempt is made to make distinctions, a completely confused state will generally result. Consequently, even if there were high-correspondence portions, they would end up being mixed together with low-correspondence portions, resulting in a disorganized state considered as low correspondence.

Let's suppose that high-correspondence portions and low-correspondence portions are confusedly mixed within the DNA information of a particular individual organism. For example, it would make things easier to understand if there was only one gene corresponding to making body color black, but since DNA information is a confused mixture, let's assume that the relationship is gene A, gene B, gene C, and so on. Let's also say that body color can only be black if these genes take specific values. Based on this, evolution that turns the body black would be nearly impossible because there is an extremely low probability of a mutation occurring that would fulfill the conditions of gene A taking value black A, gene B taking value black B, gene C taking value black C, and so on. As a result, such evolution should be nearly impossible.



Programs could be divided into high-correspondence portions and low-correspondence portions. High-correspondence portions (X) can be easily associated with outputs and behaviors (i.e. functional specifications) that can be observed externally.

With most programs, there has not been any effort to make high-correspondence portions clear and then extract them, and thus high-correspondence portions and low-correspondence portions are mixed together confusedly for the most part. Thus, there is no longer a way to clearly associate high-correspondence portions (X) with the functional specifications they correspond to.

Figure 6-1: High-Correspondence Portions and Low-Correspondence Portions

Based on this, it is conceivable that becoming able to acquire (highly correspond) new traits through a mutation on only a single gene of an organism would be the norm. If there were organisms for which this did not apply, they would either fail to adapt and become extinct, be destined for extinction, or portions that were not that way would have little relation to adaptability. Consequently, we can assume that high correspondence is the norm. Note that the specialization of organism organs appears to be due to the fact that individual organisms, that are a phenotype, are revealed by high-correspondence portions in DNA information.

An even bolder hypothesis would be to say we can predict that the evolution of such portions would be extremely slow because the mechanism that controls generation seems to be of low correspondence. In short, almost all mutations related to generation control are thought to be unable to generate individual organisms. If we equate this with program copy mistakes, it would correspond to the experience of inoperative programs most of the time. However, if one or more mutations occurring on such root portions are acceptable, they might very likely lead to **macroevolution** which changes the structure of the individual organism.

Now, instead of individual organisms, let's look at programs and functional specifications, which are their phenotype. Since most programs do not make an effort to clearly reveal and extract

high-correspondence portions, we can say that high-correspondence portions and low-correspondence portions are confusedly mixed together.

For example, when adding new functionality to a program, it is usual to have to work on multiple parts of the program at the same time. It is extremely rare to achieve the desired functionality simply by working on one part. This can be seen as evidence of the mixing of high-correspondence portions and low-correspondence portions.

If we could eliminate the confusedly mixed state of high-correspondence portions and low-correspondence portions for programs, achieving the desired functionality simply by working on one part would become the norm, at least for high-correspondence portions. There is a misconception that all programs are difficult to maintain. High-correspondence portions are by nature extremely easy to maintain. Consequently, it would be best to clearly reveal and extract high-correspondence portions. Based on such thinking, the meaning of refactoring becomes believable.

In the living world, confused mixing signifies a state in which evolution is impossible. However, with programs, the human intelligence of the developer lends a hand to evolution, and thus, in a sense, favorable mutations can be provoked simultaneously in many genes. Thus, even in a confusedly mixed state, programs can be evolved so as to adapt to a new environment. Thus far, no efforts have been made to clearly reveal and extract high-correspondence portions.

Since using ‘Business Logic Component Technology’ results in the clear revelation and extraction of high-correspondence portions, anyone would be able to easily carry out maintenance and customization, at least for high-correspondence portions. This is because normally areas that can be covered by ‘White-box Components’ are high-correspondence portions. You can understand this by recalling “Qualities Required of a 'White Box Component'.” A concrete example of high-correspondence portions is a program related to business specifications that were covered by ‘White-box Components.’ Actually, such portions can be called high-correspondence portions that can be associated with data items.

However, even portions that were thought to be generally high correspondence often can be found to be in a confusedly mixed state when studied closely. For example, in the event procedures of visual development support tools, the correspondence between each control is clear, and thus they are generally considered to be of high correspondence, but they are difficult to reuse easily. Based on this fact, there may be high correspondence from the viewpoint of controls, but from the viewpoint of data items, the partitions are blurred. If there was high correspondence from the viewpoint of data items, reuse of data items (this form of reuse is simple and desired) should be easy. Actually, adopting a componentized event-driven system that provides a mechanism for update propagation helps clearly delimit between data items, enabling data item components that are easy to maintain and customize. This fact was already discussed in “**Second Improvement of Partitioning Guidelines for Compartmentalization of Components**” within **3.2.3 “From SSS to RRR Family.**”

The essence of object-orientation lies in the attempt to emphasize the correspondence with real-world objects when designing software structure. Consequently, the clear extraction of high-correspondence portions and the clear definition of their correspondence with things can be said to be the very essence of object-orientation.

Postscript

Although a number of short to mid-range (in the two – twenty year range) allopathies have been thus far applied to business program development work, it appears that nothing like “the silver bullet” has been found. By all rights, we should strive for “the true automation of business program development,” which is likely to be made possible in the future (albeit in the long-run, possibly around a hundred years from now). However, when reflecting on this fragile reality without solid feasibility, after writing this book, I feel that the “component-based reuse” outlined in this book seems to be “the silver bullet” more than ever.

Tools Already at a Mature Stage:

Since the support tools for business apps are at their mature stage, no innovative improvement can be expected to come out in the foreseeable future. The situation could best be described using the analogy of climbing a hill that has a steep cliff. The gradual routes to the hill have already been completely developed. Up to the hill, a powerful engine called a computer works effectively. However, although minor routes to the hill will continue to be developed, that does not help very much with facilitating development work for business programs.

True Artificial Intelligence Technology Has Not Been Seen:

To develop business programs, it is necessary to climb up the steep cliff that rises on the hill. With flexible human intelligence it is possible to climb up the hill, but it is too steep for the computer to challenge by itself. The technology that can utilize the powerful engine to its fullest has not been discovered. Although many challenges have been made so far to enable the computer to climb up the cliff in various ways, none of them were successful.

It seems that the steep cliff on the hill continues endlessly. If we strove for “making the computer automatically generate application programs in the true sense,” we would be bound to face this steep hill, regardless of which route we choose. Intellectual, high-level work that humans do cannot be handled with either “the upper process intellectual support tools for business program development,” or “the reverse engineering of business programs,” because of the sheer hill. It is known that even the Y2K problem, in which the problem was limited, could not be solved automatically.

The dream tool that could perfectly generate business programs automatically is compared to alchemy in the Middle Ages. If ever developed successfully, the tool would have a tremendous impact on the computer industry. This leads to seemingly sensational people showing up every once in a while claiming, “I generated gold!”

As basic research proceeds, the steep cliff that blocks the routes will be identified, and it may be realized that it is impossible to generate gold because it is an element. However, if the research proceeds further, the technology based on nuclear physics would appear, which in fact enables alchemy. That is when the future computer would be able to climb up the steep cliff on its own, instead of humans. Although it may not happen for hundreds of years or even longer, I think it is a matter that will be solved in that time span. It is not something that can be solved simply by makeshift ideas.

Component-based Reuse is a Viable Solution:

If considering what we can do based on concepts like these, I presume the component-based reuse formed by ‘Business Logic Components’ is most suitable. Even though only humans can climb up the steep cliff on the gradual hill, once we have climbed up there, it will be possible to throw down a wire-rope,

thereby taking advantage of the computer engine. This approach is different from the computer itself climbing up the cliff, so it is sufficiently feasible with software products of the current technological level.

A Look at the Current Business Environment:

Aside from the future outlook, now let's take a look at the current business environment.

This book was written right in the middle of a recession. Looking at the internal causes, I noticed that chronic bloat, just like the one mentioned in the main text, was lurking underneath. Hence, might it be a chance for big business to promote streamlining and resolve the bubble condition?

It is foreseeable that we will lose any competitive advantage if we keep on mass-producing a large amount of software products without considering customization and maintenance. This is because as more and more **business packages with special customization facilities** are supplied, that perfectly fit a range of business programs at reasonable prices based on 'Business Logic Component Technology,' we are getting into an era of extreme natural selection. Meanwhile, this trend is also being boosted by the international competition in the ERP package business that raises an awareness of business packages.

In the course of time, the processes regarding business program development will be totally changed, and the division of labor will be promoted between **development firms of business packages with special customization facilities** and **customization firms** that tailor the products of the former firms to customers' specification requests. In other words, it is the division of labor between firms that componentize and firms that reuse.

Anticipating this outlook for the industry, I would like to recommend that those who are concerned with business program development in the business field consider the following:

- If you have expertise of business programs in a particular business or field, it would be beneficial to start a **development firm of business packages with special customization facilities** by tailoring the programs into business packages with special customization facilities and supplying them to customization firms.

- If you have a large number of customers, it would be suggested that you start a **customization firm** that provides them with business programs with a competitive advantage. This way, the firm can be expected to expand in the field of tuning service.

- If your firm's business system gets in a predicament as exemplified in a maintenance hell, it would be recommended to switch to a sophisticated resource architecture that employs 'Business Logic Components.' This will enable the firm not only to avoid the maintenance hell, but also to sell **business packages with special customization facilities**.

In any one of these cases, 'Business Logic Component Technology' is essential. If you seek a business opportunity during the resolution of a bloat, I strongly recommend that you employ this technology.

You can access the web-address below to obtain program samples of 'Business Logic Components' of this book. Also, I welcome your opinions and questions regarding this book, which can be communicated using the email address found at the following URL:

http://www.applitech.co.jp/

Lastly, I sincerely thank Ryusuke Shibata, a former president of Woodland Incorporation, who developed **SSS**, and Tsukasa Oonishi, a chief developer, for providing precious help to us in showing their development theory of **SSS**. In addition to my gratitude, I would like to express my great respect for their creativity. Also, their contributions to us in giving invaluable opinions in developing **RRR tools** have greatly eased our writing task. Moreover, I would like to thank Ryuji Asada, the chairman of the company, and Jyoji Ozaki, an executive, who eagerly recommended us to write this book, without which the book would have never been written. I offer them our genuine gratitude.

Drafts of this book have appeared on the previously mentioned web site for about two years. I appreciate the encouragement and comments from those who read it. Especially, I thank Professor Masashi Yoshida for offering suggestions for improving over two hundred points.

November 2003

Yasuhito Tsushima

APPENDIX 1 What Does Running a Program Mean?

In this appendix, I explain what it means to run a program as well as related issues.

Running a program means carrying out "something" in a lapse of time just like reading a book or playing a musical instrument. What is meant here by "something," is when a human's eyes follow letters on each page to understand meanings as in the case of reading a book, when his or her eyes read music notes while making sounds as in the case of playing an instrument, or when a computer's **CPU** computes tasks as it follows programmed instruction as in the case of running a program. Simply put, there is a device called a **CPU** in a computer that follows sentences just as a human's eyes trail letters or music notes. However, the computer does it at an incredibly high speed that can never be achieved by humans; it computes tasks as it processes some millions or even tens of millions of sentences in a second. This is what makes computer power so great.

Now, we can assume that a music note tells us what sound to make. Since the musical performance is to make sounds by interpreting notes, each note can be assumed to be telling what sound to make. Correspondingly, it can be presupposed that sentences in a program instruct what kind of computation to perform such as doing additions or comparing numbers. Accordingly, those sentences are presumed to be executable.

While in music there are symbols that indicate to repeat playing or those that indicate to jump to certain places (different from ones described earlier that indicate to make sounds), in the program there are those imperative sentences that direct to repeat computing or those that direct to jump to certain places (different from ones described earlier that command to compute tasks). These symbols or sentences with different functions from the previously mentioned ones are there to control the flow of music or the flow of the program. Also, the **control structure** in this book is the instructions that control the flow of programs.

It is conventional to use some programming language in program development. The program language is different from natural language and has no exceptions in grammar and little vocabulary. It is, so to speak, a simple language that only gives instructions to a computer. Although having these differences, they have some things in common, and hence, have common expressions. For example, just like writing sentences in natural language, developing a program is called "writing," "describing," "making," "creating," or "structuring." Yet, given only a few mistakes in programming, the **CPU** will not process what we intended it to do. The **CPU** does nothing but what is exactly commanded, nor does it give consideration to our intention. Therefore, debugging is required, which checks whether the **CPU** functions in a different way from what we intended it to do and correct the program if needed.

Due to the small vocabulary of the programming language, there are times when the computer language cannot communicate complicated expressions. Under this kind of circumstance, there is a mechanism that defines **subroutines** that correspond to new words and simplifies the complicated expressions. This can be regarded as the structure of component-based reuse. Just as we can contract a sentence by using words with a complex meaning, utilizing subroutines can shorten a program. Also, utilizing subroutines is termed "calling."

Since the speed at which the **CPU** processes sentences goes beyond some millions to tens of millions per second, it seems that a small program is executed instantly. However, because we can make a program loop many times by controlling its flow, some technological calculations, albeit small, take a fast **CPU** over an hour to execute. On the contrary, business apps in the business field do not generally perform complex

calculations like those technological ones. Instead, the **CPU** is expected to deal with a large amount of data in the business field, and hence, it mostly performs similar processes repeatedly.

To deepen an understanding of programs, it would be a shortcut to read programs written by other people or to write programs yourself. For example, if you have a widespread PC with Windows OS, it is easy to write a program and execute it, using a visual development support tool, such as, Microsoft Visual Basic. Visual Basic has functions not only for professional programmers but for hobby programmers as well, so it is a good choice for novice programmers to begin with. In addition, since it is advantageous to be familiar with event-driven systems in order to understand this book, it is recommended to use a type of Visual Basic that adopts an event-driven system.

APPENDIX 2 General Features of Business Applications in the Business Field

In this appendix, I list the three most conspicuous points that describe what kind of general features business programs have.

- **There are a wide variety of business programs in the business field.**

  What is included in a business process depends on a mixture of factors, such as the history of each company, corporate strategies, the power-balance of departments, enterprise characteristics, types of products or services, industry customs, industry characteristics, the creative activities for differentiation and so on. So, it varies from company to company. Therefore, a business program developed for Company A cannot be applied to Company B without modification.

  Also, in response to a **high NCA (Need for Creative Adaptation)** in the business field, often, business packages are developed that can be customized to either Company A or B by specifying parameters. Nonetheless, there are frequent situations where Company C orders specification change requests that cannot be adequately managed just by specifying parameters.

- **Business programs in the business field process various kinds of data.**

  Business programs in the business field normally deal with thousands of kinds of data, and it is conventional to call each of them a **data item** and identify them with data item names. For instance, they are clearly identified with the data items, such as “product code” or “sales date.”

- **Business programs in the business field process a large amount of repetitive data.**

  Although business programs in the business field deal not only with a high volume of data, but also many types, data still tends to be simple and repetitive. These kinds of data are, using an old trick, stored in containers called records. Records are the containers that are stored with correlated data items. Thus, if you build many records, you can store lots of repetitive data.

  Moreover, by applying the program procedure for a certain record to many others, it becomes possible to process a large amount of records one after another in order to process that record as requested. Also, while a record is being processed, it is in a critical condition because of the danger of simultaneous alteration, so **exclusion control** is needed to prevent access from the outside. In addition, in order to write several records to non-volatile memory, such as disks, each time a form is processed, **transaction control** is required.

Contrary to business programs in the business field, in the technological field, very profound calculations are done based on a few data; it can be likened to a deep narrow hole. Applying the same comparison to business programs in the business field, we can imagine street stalls at a festival, which have a wide entrance and a shallow depth. This is because with the programs that comprise a business program, calculations for each data item are not awfully complicated, but there are a vast number of data items.

If you consider how to divide business apps with these kinds of characteristics, and how to correspond them to data items, you could be convinced of the contents of this book.

APPENDIX 3 Example Using Build-Up Method to Determine Improvement Rate of Productivity

When you estimate the improvement rate of productivity by the build-up method, there are some parts for which we have no choice but to depend on our subjective views or feelings. How influential those subjective factors are can be realized if we actually do the estimation. In this appendix, I report the processes of estimating the benefits of one of the seeds (materials, policies, facilities, and mechanisms) for reference when such attempts are made.

General explanations of how to estimate the improvement rate of productivity in terms of the build-up method can be found in **4.2 "Various Ways of Measuring Software Development Productivity."**

Here, I selected some seeds that require consideration during the estimation or, in other words, seeds that are difficult to estimate. One of the popular seeds slotted in CASE tools is the support for writing various diagrams with upper process support tools. This seed is something that helps clarify requested specifications in the broad sense and, concretely, something that acts as a word processor to write diagrams. Also, since the seed has both direct and indirect work saving effects, we estimate each of these as the product of its percentage of supported work and its percentage of work saving.

Let's estimate the direct benefits first. The work of clarifying requested specifications sometimes accounts for about 20 percent of all development work, and it includes a wide range of tasks such as interviews with customers and end users. Considering that the diagram-making work is only part of that entire process, the percentage of supported work should be approximated, at the most, at 5 percent of all development work.

Opinions are divided on how to estimate the percentage of work saving in this situation. It is possible to view it as a word processor for drawing diagrams that merely supports revision work and nothing more. What is drawn as a diagram comes up in humans' minds, and no support is provided for what kind of diagram comes up in their minds. From this viewpoint, it seems that the benefits derived from this support are limited. Yet, this seed makes one think that rewriting is not troublesome, which results in the psychological effect of reducing one's reluctance to find overlooked ideas. There may be some other accompanying effects. Considering this, let's suppose the percentage of work saving by this seed is 40 percent. Then, the percentage of direct work saving by this seed can be estimated by multiplying the percentage of supported work, of about 5 percent, and the percentage of work saving of 40 percent, together, which is equal to 2 percent.

Since this value of 2 percent is an estimate for direct supports, we need to include indirect supports in the calculation as well. Specifically, we also need to calculate the benefits of the implementation work of requested specifications after diagrams are made. If wasteful work that is incurred in post processes that are caused from the inadequate clarification of requested specifications, there should be an indirect support in that it reduces such wastes. Simply put, productivity is improved through this process.

The indirect benefits are calculated by multiplying the percentage of supported work by the percentage of work saving after they are each obtained, as the same way as for the direct benefits. First, we suppose that the percentage of supported work is about 80 percent of all development work. Then, as for the percentage of work saving, its value varies depending on whether or not it is a development project in which many mistakes are often made in interpreting requested specifications, and also depending on when these mistakes are found. With a development project with many mistakes, the value of indirect benefits

should be larger since there is more room to improve its productivity. With that said, let's assume that the probability rate of making mistakes in interpreting requested specifications is X, and that we found those mistakes right in the middle of a project. On these assumptions, we must have conducted wasteful work of 0 to 0.8 X, and we should be able to save 0.4X waste work.

Having grasped the above explanations, you are ready to compare the indirect benefits before and after this seed is adopted.

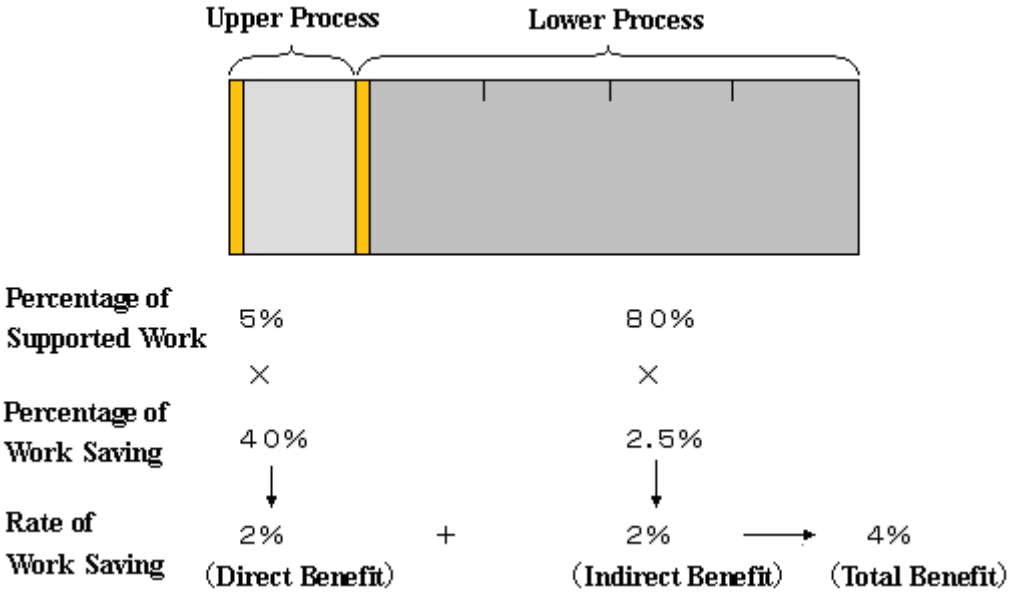
Before the seed is adopted, namely when you were creating diagrams with a pencil and an eraser, if, for instance, we made 20 percent mistakes (simply put, assume X = 20 %), you must have done 8 percent wasteful work in post processes on average (80 % × 20 % × 1/2).

After this seed is taken up, let's assume that the probability rate of making mistakes in interpreting requested specifications decreases from 20 percent to 15 percent through the adoption of support tools for creating diagrams. In this case, the wasteful work can be reduced from 8 percent to 6 percent on average (80 % × 15 % × 1/2).

Therefore, by utilizing this seed, we can save 2 percent of wasteful work on average.

Given the percentage of supported work is approximately 80 percent and the percentage of work saving is 2.5 percent (mistakes are reduced by 5 percent, and they are found in the middle of a project), the rate of indirect work saving is 2 percent.

Combining these benefits (or accumulating them), we can obtain the rate of work saving of 4 percent by adding up the direct rate of 2 percent and the indirect rate of 2 percent. Refer to **Figure A3-1.**



An example of the calculation for the rate of work saving when handwriting is changed to CASE tools for diagram writing work in a development project in which approximately 20 percent of mistakes are expected on average in interpreting requested specifications:

Figure A3-1: Value Obtained for Rate of Work Saving

Although we assumed that the probability rate of making mistakes is 20 percent, development projects with a higher rate have more room for productivity improvement, and vice-versa. That is to say, it is normal that improving productivity of a development project on which improvements have already been attempted is not easy.

Also, in this appendix we estimated the benefits, assuming a development project that has already adopted the technique of utilizing diagrams. But, in a development project with no such technique adopted, we can expect more benefits by putting together that technique and those tools.

Please note that I am not claiming that an exact value for productivity improvement for diagram writing support was calculated. The above calculations involve various assumptions, so looking at only those values can lead to a misunderstanding. Those calculations are intended simply to show the calculation method. I simply wish to draw the reader's attention to the process of estimation, and I would just like you to refer to them when you estimate the improvement rate of productivity for some given seed. Now, it's your turn to pick a seed and try the estimation for it.

APPENDIX 4 Demarcation of Figure and Ground When Recognizing Something

When a human perceives an object, an internal mechanism functions to associate and identify it with some preconceived concepts or images. In this case, the demarcation is made between the **figure** and the **ground** (background) so that we can perceive it.

The picture in **Figure A4-1** is "Vase/Faces Drawing (Optical Illusion)," which is famous for allowing two types of perceptions. When regarding the white portion as the figure (then, the black portion becomes the ground), you can perceive the picture as a vase; however when taking the black portion as the figure (then, the white portion becomes the ground), you can perceive it as two faces.

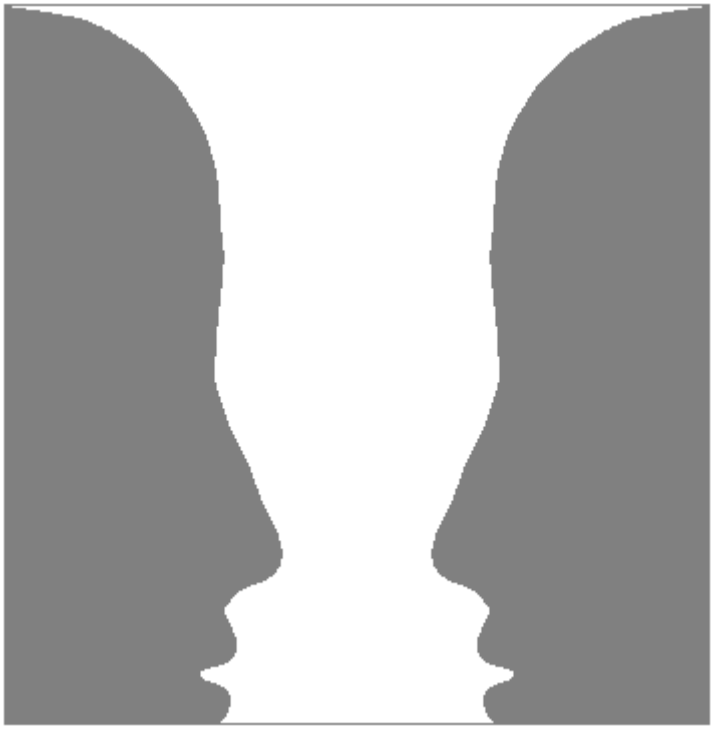


Figure A4-1: Vase/Faces Drawing (Optical Illusion)

In conventional software development, subroutines are considered a **figure** in general, and **common subroutines** have been aggressively created in most of the development projects. However, there has been no such viewpoint from which main routines are considered the **figure**, so I suspect that no effort has been made to find **common main routines**. With the latter perspective, we may find common portions that have not been found by perceiving things from a different angle.

This is related to the perception ascribable to the inversion of conceptions, such as the inversion of control in terms of object-orientation.

In the meantime, the relationship between the main routine and the subroutine is a relative one rather than an absolute one. Thus, when you pick one routine, it may be a main routine that calls other routines, while at the same time it could be a subroutine that is called by other routines. With this dual characteristic, we sometimes see this misunderstanding, that is, the idea that we should think of making only subroutines

as has been done without considering main routines. The point here is the perspective in finding common portions, not the implementation method for subroutines.

As for the implementation method, if you create a principle main routine consisting of a one-line program, it is possible to make all the other routines subroutines. This is because when there are things that are considered main routines, it is very easy to change them into subroutines (those that are called by the principle main routine); accordingly, it is obvious that we can transform all of them into subroutines. In this regard, they can function sufficiently as subroutines. However, doing this does not necessarily help find common portions that have not been found.

What is important is the perspective that facilitates finding common portions. When making the demarcation between **those that call** and those **that are called** to find common portions, we paid attention only to the commonality of those that are called. For this reason, we could not see the commonality of those that call. We could not, of course, because we did not try. Paying enough attention to the commonality of those that call, we can gradually see common portions; for example, it becomes obvious that skeleton routines of the fill-in system, operation bases that implement certain operation specifications, 4GL operation bases, GUI operation bases, and the like, are all common main routines. It is simple once we know the trick, but if we do not find it, common portions are often overlooked.

When common portions of those that call are found, it is totally up to you how to implement them. It is conventional, however, to build a structure for the common portions of those that actually call in such a way that it involves local main routines and subroutines within. This is comparable to the fact that there are local main routines and subroutines in the common portions of those that are called, i.e. in conventional subroutine structure.

APPENDIX 5 Generalized Construction Technique for a Reuse System of Componentized Applications

In this appendix, I first explain the process by which I derived the generalized construction technique for a reuse system of componentized product on the basis of **the three requirements** for a practical and effective component-based reuse system. I then show evidence for the theorem that if a certain software product can be constructed as a componentized application according to this generalized construction technique, or in other words, if there is an RSCA that complies with this technique, then the system truly can fulfill the three requirements.

The three requirements for a practical and effective component-based reuse system involve the following.

- Software products must be built up solely by combining components;
- The system must be able to meet all customization requests; and
- Large numbers of developers must systematically benefit from component-based reuse.

Also, the generalized construction technique for a reuse system for componentized applications should:

- Partition the area that software products cover into low NCA areas and high NCA areas.
- Cover all low NCA areas by using 'Business Logic Components' or 'Software Components' that have the following quality (i.e. quality required of a 'Black-box Component'):
  - Ability to meet all envisioned requests in areas that must be covered by selecting components from component sets and specifying parameters (generality).
- Cover all high NCA areas by using 'Business Logic Components' or 'Software Components' that have all of the following four qualities (i.e. qualities required of a 'White-box Component'):
  - Easy retrieval of desired components (retrievability).
  - Number of components that must be revised is limited (locality).
  - Size of each component is suitable (suitable size; suitable granularity).
  - Each component is easy to decipher (readability).

Appendix 5-a Process by Which Generalized Construction Technique for a Reuse System of Componentized Applications Was Derived

The generalized construction technique for a reuse system of componentized applications was derived in the following way.

In the business field at least, application programs are rarely built solely by general subroutines because general subroutines cover only low NCA (Need for Creative Adaptation) areas and leave high NCA areas untouched. Thus, to cover high NCA areas, some kind of new "components" like 'Software Components' other than general subroutines are needed. In the meantime, rather than try to cover all the low NCA areas only with general subroutines, we usually need to use them together with common main routines.

Based on those understandings, we forecasted that we would need to use some types of 'Business Logic Components' or 'Software Components' together and, consequently, started to seek 'Software Components' other than general subroutines.

Now, if you consider why high NCA areas cannot be covered by general subroutines, you will probably find that it is because of the premise that they do not allow program customization. If they allow program customization, they should be able to cover high NCA areas.

Therefore, following the above consideration, I decided to classify components into those that never allowed program customization and those for which you are resigned to performing program customization. Then, we named the former the black-box component and the latter the white-box component.

After that, in an attempt to investigate the appropriate use of each component, we tested them on the second and third requirements of the above three.

If you build a component-based reuse system solely using black-box components, large numbers of developers can systematically benefit from that component-based reuse, but the range of customization requests that you can handle will be limited.

On the other hand, if you build a component-based reuse system only using white-box components, any customization request can be met, but it is likely to become a system with which systematic component-based reuse is difficult.

Either way, it was not easy to make the second and third of the three requirements compatible with each other. So, I divided a program into low NCA areas and high NCA areas to grope for a way to handle each of them independently. That is, I planned to manage each of those areas separately either with black-box components or white-box components.

First, I estimated that low NCA areas could be covered by black-box components. Then, I considered what to do to make this happen.

It would be problematic if using black-box components resulted in limiting the range of customization requests. However, since low NCA areas were where we could expect what kind of customization requests are likely (or areas that are easy to handle), we should adequately handle them only with parameter customization. Thinking this way, I concluded that it would be okay if black-box components were equipped with **generality**, the quality that is explained below. Also, I decided to call this quality **the quality required of a 'Black-box Component.'**

- Ability to meet all envisioned requests in areas that must be covered by selecting components from component sets and specifying parameters (generality).

If black-box components have this quality, they can meet any envisioned customization request in areas that must be covered by selecting components from component sets and specifying parameters, which means that black-box components can cover low NCA areas. Also, black-box components with this quality are consistent with the definitions of 'Business Logic Components' and 'Software Components' of this book. In this book, this black-box component is expressed as 'Black-box Component.'

Next, I assumed that high NCA areas needed to be covered by white-box components so that any customization request could be met. The problem here was how I could make white-box components "systematically reusable for large numbers of developers."

Breaking down this problem, the following four qualities came up, and I thought that if white-box components had all of them, a large number of developers would be able to reuse them systematically. Combining together these four, I decided to call them **the qualities required of a 'White-box Component**.'

- Easy retrieval of desired components (retrievability).
- Number of components that must be revised is limited (locality).
- Size of each component is suitable (suitable size; suitable granularity).

- Each component is easy to decipher (readability).

If we can build a program consisting of 'White-box Components' with all four qualities, systematic reuse by large numbers of developers becomes possible. This is because, in response to a certain customization request, desired components are retrieved quickly, the number of them is limited to one or two, each one of them is not too large, and they are easy to decipher. In other words, thanks to retrievability, we no longer go astray due to modules that the original developers arbitrarily partitioned, and thanks to locality, suitable size, and readability, we no longer have to painstakingly decipher the whole program. Although the decipherment of a program is still sometimes necessary, it is within the permissible range for large numbers of developers to reuse it systematically. Conversely, it would be easier to understand if you recognize that we require sufficient retrievability, locality, suitable size, and readability of 'White-box Components' in order to make the decipherment work within the permissible range.

Therefore, if such a 'White-box Component' with all of these qualities exists, systematic reuse by large numbers of developers is possible. Then, because white-box components are components for which you are resigned to performing program customization, they can cover high NCA areas. Also, 'White-box Components' with all of these qualities are consistent with the definitions of 'Business Logic Components' and 'Software Components' of this book. So, in this book this type of white-box component is noted as 'White-box Component.'

Although I listed the four qualities above as the qualities required of a 'White-box Component' and created the component equipped with all of them, there might be other appropriate combinations of qualities. But, I did not go too far with this. If you are interested, you can go further.

This is the process by which the generalized construction technique for a reuse system of componentized applications (RSCA) was derived.

Please note that precisely what covers low NCA areas can be 'Black-box Components,' 'White-box Components,' or even the combination of the two. However, it is desirable to cover them with 'Black-box Components,' if possible, since it is better to reduce the need for the maintenance of a program. To cover low NCA areas, we should use as few 'White-box Components, with which maintenance of a program is needed (or may be needed), as possible, and use 'Black-box Components' whenever you can (you should be able to).

As described earlier however, high NCA areas cannot be covered by 'Black-box Components.'

Appendix 5-b Proving Theorem of Satisfying the Three Requirements

In this section, I show evidence for the theorem that if a certain software product can be constructed as a componentized application according to the generalized construction technique, or in other words, if there is an RSCA that complies with this technique, then the system truly can fulfill the three requirements.

The first of the three requirements, the one that says that **software products must be built up solely by combining components**, is satisfied. This is because low NCA areas in the areas covered by software products are covered by black-box components that meet the qualities required of a 'Black-box Component,' and high NCA areas are covered by white-box components that meet the qualities required of a 'White-box Component.'

Hearing this abstract explanation, you may not picture concrete images of "what it means to divide software product coverage into low NCA areas and high NCA areas," or "whether the fact that components cover certain areas is the same as the idea that a software product is built up solely by combining components." If you feel that way, refer to the examples in **"5.2 Technique for Constructing Component-Based Reuse Systems and an Actual Example"** to aid your understanding.

As for the second of the three requirements, which states that **the system must be able to meet all customization requests**, let's consider it in terms of both the low NCA area and the high NCA area.

With regard to the low NCA area, you should be able to speculate as to what customization requests are likely. Then, because the black-box components that cover that area meet the "qualities required of a 'Black-box Component,'" or have generality, "(ability to meet all envisioned requests in areas that must be covered by selecting components from component sets and specifying parameters,)" the second requirement is fulfilled.

The high NCA area, on the other hand, is covered by white-box components for which you are resigned to performing program customization, so that any request can be handled. Consequently, the second requirement is met.

For the last of the three requirements, the one that claims that **large numbers of developers must systematically benefit from component-based reuse**, let's consider it in terms of both the black-box component and the white-box component.

With the black-box component, since we can utilize it only by specifying declarative information with clear-cut meaning when viewed externally without deciphering the internal program, the third requirement is satisfied.

As for the white-box component, although the decipherment of a program sometimes may be required, depending on customization requests, by meeting the "qualities required of a 'White-box Component,'" the systematic reusability by a large number of developers can be guaranteed. Therefore, the third requirement is fulfilled.

To describe this in depth, in response to a certain customization request, desired components are retrieved quickly, the number of them are limited to one or two, each one of them is not too large, and they are easy to decipher; therefore, a large number of developers can systematically reuse the components. As explained earlier, due to retrievability, we no longer go astray due to modules that the original developers arbitrarily partitioned, and thanks to locality, suitable size, and readability, we no longer have to painstakingly decipher the whole program. Although the decipherment of a program is still sometimes necessary, because they are easy to decipher, and the range is limited, decipherment is acceptable.

With all the explanations described above, I have shown evidence for the theorem that if we can build a software product as a componentized application according to this generalized construction technique, or in other words, if the reuse systems of componentized applications (RSCAs) based on this generalized construction technique exist, then that system must fulfill **the three requirements.**

What should be noted here is that it is not guaranteed that RSCAs like this can be built in every field. Rather, I can only say that there exist actual cases of RSCAs based on the generalized construction technique in the business field, and that we can build RSCAs according to this generalized construction technique, at least for general application programs in the business field. However, we cannot guarantee,

for example, whether the software that supports the human genome project can be built based on this generalized construction technique. Similarly, even in the business field we may not be able to build RSCAs based on the generalized construction technique in specific areas. Regarding this issue, you can refer to the explanation of the **component-enabled portion** in Chapter **5 "What are 'Business Logic Components?"**

Finally, let's consider the theorem that I just described above in reverse order, that is, the proposition of whether or not we can say that it is needed to employ **the generalized construction technique** in order to make the component-based reuse system meet **the three requirements**. Unfortunately, this has not been proven. In this book, I only explained the process by which **the generalized construction technique** was derived. Although I presume that only this generalized construction technique, or its somewhat revised versions, can satisfy the three requirements, this presumption has not been officially proven. So, there may be other construction techniques. If you are interested, why don't you study them? Then, if you come up with a new construction technique, try to prove that your construction technique can satisfy **the three requirements** too.

References

Some of the following references are written only in Japanese. Consequently, these references have Japanese characters in it. Although mainly Japanese language editions were referenced, the corresponding English language editions are shown if possible.

Note that some of the following references are bad examples which should not be followed.

Besides the following references, manuals of software products (packages and tools) are referred to while writing this book. These manuals are not in the following list, because showing them is the same as directly speaking ill of them. It is very difficult to find a practical and effective component-based software reuse system.

1. Schrödinger,E., What is Life?: The physical Aspect of the Living Cell, Cambridge University Press, 1944.
2. 小野寺力男著：グラフ理論の基礎, 森北出版, 1968.
3. 藤田広一著：基礎情報理論, 昭晃堂, 1973.
4. Dawkins,R., The Selfish Gene, Oxford University Press, 1976.
5. 岡田節人著：動物の体はどのようにしてできるか, 岩波書店, 1981.
6. Brooks,F.P.,Jr, "The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, Addison-Wesley, 1975.
7. 電気通信学会編：データ通信ハンドブック, オーム社, 1984.
8. Hofstadter,D.R., Gödel, Escher, Bach: an Eternal Golden Braid, Basic Books, 1979.
9. 丸山武編：ソフトウェア構造, オーム社, 1985.
10. 土井輝生著：著作権の保護と管理, 同文社, 1985.
11. Gould,S.P., Ever Since Darwin: Reflections in Natural History, W. W. Norton & Company Incorporated, 1977.
12. 石井慎二編：進化論を楽しむ本, JICC 出版局, 1985.
13. 清水博, 餌取章男著：生命に情報をよむ, 三田出版会, 1986.
14. Kimura,M., The Neutral Theory of Molecular Evolution, Cambridge University Press, 1983.
15. 樋渡宏一著：性の源をさぐる, 岩波書店, 1986.
16. Dreyfus,H.L. and Dreyfus,S.E., Mind over Machine: The Power of Human Intuitive Expertise in the Era of the Computer, Free Press, 1986.
17. Simon,H.A., The Sciences of the Artificial second edition, MIT Press, 1969.
18. 酒井博敬著：情報資源管理の技法, オーム社, 1987.
19. Dawkins,R., The Extended Phenotype: The Long Reach of the Gene, Oxford University Press, 1982.
20. 長屋宏著：アレルギー, 中央公論社, 1988.
21. Ohno,S., Evolution by gene duplication, Springer-Verlag, 1970.
22. 河田雅圭著：進化論の見方, 紀伊國屋書店, 1989.
23. Budd,T., A Little Smalltalk, Addison-Wesley, 1987.
24. Winograd,T. and Flores,F., Understanding Computers and Cognition: A New Foundation for Design, Addison-Wesley, 1987.
25. Wiener,R.S. and Pinson,L.J., An Introduction to Object-Oriented Programming and C++, Addison-Wesley Professional, 1988.
26. 門内淳, 赤堀一郎著：C++ プログラミング, 日本ソフトバンク, 1989.
27. Hofstadter,D.R., Metamagical Themas, Basic Books, 1985.
28. Norman,D.A., The Psychology of Everyday Things, Basic Books, 1988.
29. 柴田隆介著：会社もけっこう面白い, 日本経済新聞社, 1990.

30. Shu,N.C., Visual programming, Van Nostrand Reinhold, 1988.
31. Brown,A.W. and McDermid,J.A.: Learning from IPSE's Mistakes, EEE SOFTWARE, pp.23-28, March 1992.
32. Casti,J.L., Paradigms Lost: Images of Man in the Mirror of Science, William Morrow and Company, 1989.
33. 吉永良正著：ゲーデル・不完全性定理, 講談社, 1992.
34. 黒川利明著：ソフトウェアの話, 岩波書店, 1992.
35. Wegner,P., Concepts and Paradigms of Object-Oriented Programming, OOPS MESSENGER – A Quarterly Publication of Special Interest Group on Programming Languages, VOLUME 1 NUMBER 1, ACM press, 1990
36. 牧野由彦著：考える遺伝子, 東京ベル印刷, 1992.
37. 山本隆雄, 渡辺理人, 山本聡, 嶺町優司著：コンピュータ・ウイルス, 講談社, 1993.
38. 板倉稔著：スーパーＳＥ, 日科技連, 1993.
39. 畑中正一著：ウイルスは生物をどう変えたか, 講談社, 1993.
40. 玉置彰宏著：なぜオブジェクト指向なのか, 日経コンピュータ, 1993 年 9 月 20 日, No.319, pp.132-133.
41. Cringely,R.X., Accidental Empires: How the Boys of Silicon Valley Make Their Millions, Battle Foreign Competition, and Still Can't Get a Date, Addison-Wesley, 1992.
42. 竹下亨著：CASE を探索しよう －ソフトウェアの再利用－, Vol.25, No.9, bit, 1993.
43. Gausw,D.C. and Weinberg,G.M., Exploring Requirements: Quality Before Design, Dorset House, 1989.
44. 木村泉著：ワープロ作文技術, 岩波書店, 1993.
45. Dawkins,R., The Blind Watchmaker, Norton & Company, Inc, 1986.
46. Ellen,T., Artificial Life: Explorer's Kit, Sams Publishing, 1993.
47. 吉田征著：技術の伝承と移転, 日科技連, 1994.
48. Webster,B.F., Pitfalls of Object Oriented Development, M & T Books, 1995.
49. Wiener,J., THE BEAK OF THE FINCH : A Story of Evolution in Our Time, Vintage Books, 1995.
50. 岡本龍明, 太田和夫編：暗号・ゼロ知識証明・数論, 共立出版, 1995.
51. Sigmund,K., Games of Life, Oxford University Press, 1993.
52. Norton,P. and Davis,H. and Davis,P., Peter Norton's Visual BASIC 4 Programming, Prentice-Hall Canada, 1995.
53. Hoff,A. and Shaio,S. and Starbuck,O., Hooked on Java: Creating Hot Web Sites With Java Applets, Addison-Wesley, 1996.
54. Waldrop,M.M., Complexity: The Emerging Science at the Edge of Order and Chaos, Simon & Schuster, 1993.
55. 立花隆著：脳を究める, 朝日新聞社, 1996.
56. Vaughn,W.R., Hitchhiker's Guide to Visual Basic & SQL Server, Microsoft Press, 1997.

Index

Rather than showing the page number, this index gives the section number where each term appears. If the section number is in **gothic** then please refer to it first.

Abbreviation **r.t.** means related term. Please refer to it.

The words **in parentheses** following the main term have the same meaning or explanation for assistance.

[D]

[E]

[F]

[G]

[H]

[I]

[K]

[L]

[M]

[N]

[O]

[P]

[Q]

[R]

[S]

[T]

[U]