Modeling of Software Reusable Component Approach and its Case Study

Kazuo Matsumura, Akihiro Yamashiro, Toshiyuki Tanaka, and Ikumune Takahashi

Systems & Software Engineering Laboratory, Toshiba Corporation
70 Yanagi-cho, Saiwai-ku, Kawasaki 210, Japan

ABSTRACT

Software has become more complex and larger. In order to develop high quality software at a high level of productivity and in a short period of time, it is indispensable to establish software reusability techniques including the software reusable component approach. Our approach is called "standardized components & their composition approach". We have developed a technique which is characterized by a development process and object layers for promoting software component reusability.

Our technique, on condition that the architecture of a product family is fixed for a certain period of time, has the following features: (1) development of software components and that of software products are performed independently, (2) guidelines for software component analysis/design can be obtained using an object layer which is based on shared software resources (called objects). This technique has been applied with excellent results.

1. INTRODUCTION

Software reusability techniques are essential to the development of high-quality software at a high level of productivity to meet short deadlines. Reuse of the software can be defined as "repeated use of software (including documents) and of the knowledge and procedures required for the development thereof through their formalization and standardization". In this sense, we recognized that reused software includes the elements of software classified by Freeman, that is, 1) code fragments (a part of executable program codes), 2) logical structures (internal designs of processes and data structures), 3) functional architectures (external specifications of system functions and data), 4) external knowledge of application domain and development), and 5) environmental-level information (knowledge of utilization and technology transfer).(1)

Recently, attention has been concentrated on reusability techniques at a higher abstract level than that of program codes. The object-oriented design method, proposed by Booch(2), aims at enhancing the reuse of logical structures by finding objects and their classes unique to specific applications. This method is intended to be used in object-oriented languages including Ada. It has been reported that real time SA (Structured Analysis) and modern SA, which are used as a method for upper CASE (Computer Aided Software Engineering), are effective for the maintenance and reuse of functional architectures and external knowledge through the accumulation of information in CASE tools. Osterwell's process model(3) aims at formalizing and reusing knowledge from and for software development.

We have been developing an integrated software management and production support system (IMAP), which aims at the industrialization of software development. In this system, we proposed and implemented design description methods and tools for the purpose of reusing module components at the levels of program code and logical structures(4,5,6). In our experience, we have found that there is no common component set which can be widely applicable to any application domain. We have also found that making reusability techniques effective requires an integrated approach which covers not only a single particular level but levels above and below that level. For example, making code-level reusability effective requires improved comprehensiveness about design information at the logical structure level. On the contrary, once reusability at the logical structure level becomes practical, automatic conversion into and retrieval at the code level become essential as a matter of course. Also, reusability at the logical or code level often assumes a specific system architecture at the functional configuration or external knowledge level. It is no rare case that reusability efficiency is considerably lowered for systems with other architectures.

With our approach, therefore, we start with making a future plan for standardized software components
within the range of the product architecture, then proceed with component design and development so that consistency through each level can be maintained.

This paper presents our new technique to reuse standardized software components featuring a component development process and component-oriented object layers, together with validity demonstrated through applications.

2. DEFINITION OF SOFTWARE COMPONENTS

We call our approach a “standardized components & their composition approach”. In this approach, prepared standard software components (hereinafter referred to as components), are provided for reassembly and the creation of the target software. The missing parts of the software are developed by programmers as component users. The advantages of this approach include improved software productivity and quality due to reuse of software components and the ease of software modification which is accomplished by localizing the changes in system environment to those in the software components. In addition, the provision of software components standardizes and simplifies software development tasks. This approach features the following:

(1) Components are developed and managed not by individuals but by the organization concerned.
(2) Sets of operations and/or processing for each object (to be described later) used to implement a specific product domain are provided as a component.
(3) The procedures and guidelines for component development are provided.

The term “component” can be generally defined as follows:

“... A component is a unit of software selected for reuse. Software can be made by using such components. Normally components include the following:
- Programs (source code, object code, etc.)
- Documents (specifications, design documents, test certificates, etc.)
- Test programs and data.”

Components in general, however, range in different concepts and categories. Figure 1 shows an overview of components depicted on three axes. With our approach, we handle only the components enclosed in boxes as components, that is:

(1) Black box components ... This kind of component is used without manufacturing their contents. Only the external specifications are visible.
(2) Object-type components ... This type of component aims at object-oriented components as their final form. However, since their characteristic is limited to data abstraction out of the many features (class/instance, inheritance, etc.) of object-oriented components. They are called object-type components as distinct from object-oriented components. The term “object” also refers to various kinds of shared resources (in the broad sense of the term, including the hardware, human operator, the operating system (OS), files, data types, and so on) that implement actual products or software. A set of operational components for an object is tied up under the concept of data abstraction. In Ada, therefore, the object-type component can be implemented as a package. In some ordinary procedure-type languages, the object-type component may be implemented as a library. Individual operational components included in a single object-type component assume modules such as functions and procedures.

(3) Domain-oriented components: This kind of components can be applied to limited application domains.

The reasons for adopting object-type components are that they can be implemented by many procedure-type languages, and that the necessary operational components are grouped under the labeled object, and they allow both component developers and users to have a good degree of foresight for combinations of the operational components. The use of black box components is intended to show who is responsible for quality assurance without allowing the users to tamper with them.

We have proposed a design description technique for these components under the name TFF (Technical description Formula for Fifty steps/module design) and have developed support tools(4,5).
3. MODELING OF OUR APPROACH

3.1 Assumptions and Hypotheses

Our approach is based on some assumptions (applicable conditions) made through experience and some hypotheses (not yet proven but demonstrated by actual application cases).

(1) Product and components development

Assumption 1: It is possible to fix the architecture of a product family for fixed periods.

Hypothesis 2: It is possible to develop components in advance based on the product architecture and technology forecasting.

Hypothesis 3: A good component system can be classified into layers by the unit of the object-type component.

Hypothesis 4: The usability of components requires interface compatibility with their lower layers.

Components depend strongly on the product architecture concerned. It is rational that different operating systems, languages and peripheral devices require differing basic, lower-layer components. Differences in product functions should also be reflected in the higher-layer components. There are no components that can be always used for any application domain. It is realistic and effective to use components within the specific product family by assuming the life cycle of a product architecture to be from two to five years. When ease of modification to a component system is a prime requirement, correlations (especially invoking correlations) among the components and a layer structure based on the correlation are of particular importance to localize the component interface.

(2) Development environment

Assumption 5: Providing a standard development environment is essential to efficient product development.

Hypothesis 6: It is possible to standardize a way of developing software through the same design methodology in a software development team.

If we lack uniformity of our development environment including a development machine, OS, language, design methodology, etc., it results in a very low level of efficiency in the product development work. Expansion of software component reusability from the personal to the team level requires a consistent design methodology (design procedures, criteria and description methods) and the basic concept of support tools.

(3) Development organization

Hypothesis 7: The development of components and the products can be performed independently.

Hypothesis 8: The development of components and the products can be assigned to different groups.

We have already reported on the possibility of assigning the work of component production, maintenance and management to independent organizations (4, 5).

3.2 Component-oriented Development Process

Figure 2 shows a development process which permits developing components independently with an emphasis on the product architecture.

At the planning stage, a new product development project is always discussed in terms of a new product family with a certain life. Specifically, problems with similar products developed in the past are sorted out and a basic system design is made that is based on user needs, demand forecast and hardware (including peripheral input/output devices) and software technology forecasts. The methods available up to this point for software development include SA or real-time SA for function analysis and definition, and the selection of its implementation category (hardware, software, or humanware). Also the target system environment, system operation and maintenance policies, and extraction of technical factors to be newly developed are of great significance, as they affect the direction of subsequent component standardization.

At the system design stage, it is necessary to define a product architecture. There are the following items of architecture which must be designed with the use of components in mind:

- Hardware configuration
- OS, language, common tools
- Functional configuration
- User interface
- Software configuration

Common tools include a data base, windowing, communication software, and other extended OS functions. In recent years, common tools and user interfaces have been increasing in importance and have had a large impact on the development of software components. Functional configuration is designed in detail by means of real-time SA and other methods. During this design process, it is clarified whether each
function is a standard or optional function. For software configuration design, the overall software configuration is designed, including not only the tasks, processes, files and common data in application software, but the implementation environment for them as well - i.e. OS, drivers, managers, system call libraries, and so forth.

The subsequent stage divides the work into component development and product development. In the case of a first product, the development of components goes ahead of the development of the product. In the component planning stage, object-type components are extracted from the software configuration mentioned above. At this point, the main work is confined to the generation of an object-type components classification table and the listing of the operations or functions needed for each object. The most typical method to discover objects is to extract the external entities and data store from the data flow diagrams in the SA[2]. There are also some other methods: to extract, as objects, hardware components such as peripheral devices, abstract objects resulting from special processing algorithms (e.g. the segment concept in graphics processing), or objects obtained by grouping similar types of functions. In component planning, product development specialists are involved in the discussion, and the lack of any necessary components is checked for by the component users.

At the component design specification stage, the specifications for object-type components are determined through detailed design, including that of interfaces. The component-oriented object layer (hereafter sometimes referred to as the component layer) and the component design guidelines using that layer will be described in Section 3.3. The key design process is this: the interfaces for the objects extracted in the component planning stage are designed sequentially from the lower to the higher layers, and the designed object-type components are configured as units in component layers with the invoking relationship among components specified, so that the integrity of the overall component system may be determined. This process requires careful performance forecasting. Concerning component specifications, it is also important to define how to use a component. Therefore, the component specifications must include knowledge about component combinations and a description of the performance.

The components are released to the product development section after the final testing, certification and documentation of their performance and reliability are completed.

The following describes the process of development to be initiated after components for a specific product family are prepared and product development based on those components is started along the right lines at the product development section. It is common that, within a single product family, there are slight differences in user requirements for individual jobs. Therefore, at the stage of system design, it is necessary to ensure that these additional user requirements and their modifications are within the scope of the product architecture concerned.

If that is not a new job, it is usual that components and products are developed in parallel because of a shorter development period. At the stages of component planning and component design specification, a joint committee comprising members from the component and product sections is organized to identify and prepare only the necessary components other than those already developed. Whether or not a specific component is essential and could be matched with other components is checked by using component layers.
3.3 Component-oriented Object Layers

Components are to process and/or operate certain objects. They can therefore be classified by the objects to be processed into component-oriented object layers (hereinafter referred to as component layers) and assorted by an invoking relationship. (See Figure 3.)

They are broadly divided into: target environment and product application. The target environment provides an environment for the target computer system to run the application program concerned. Each layer is explained with examples as below.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Layer name</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Job-unique layer</td>
<td>Product application</td>
</tr>
<tr>
<td>4</td>
<td>Product family domain layer</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Application domain layer</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Target tool layer</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Target basis layer</td>
<td>Target environment</td>
</tr>
</tbody>
</table>

*Figure 3 Component-Oriented Object Layers*

(1) Target basis layer
Deals with the CPU, peripherals and other hardware (such as the CRT or printer), OS, languages, and firmware.

(2) Target tool layer
Deals with data base management systems, user interfaces (such as windowing, error messages), communication protocols, some basic data types and their operations, etc.

(3) Application domain layer
Deals with processes unique to each application domain, e.g., business data processing systems, process control systems, microcomputer embedded systems. A process control system includes, for example, exclusive task control and synchronous task control, digital I/O, and analog I/O.

(4) Product family domain layer
Deals with the standard functions and common data for specific product families, e.g., common display screens, output forms, data bases and status flags.

(5) Job-unique layer
Deals with components that provide additional functions unique to specific jobs.

Component layers are, as a rule, partitioned in accordance with the invoking relationship. In other words, the layers are designed so that components are invoked from within the same or an upper layer. That is one-way direction concerned with invoking.

In analyzing and designing components, it is necessary not only to perform their functional decomposition in a top-down manner but also to conduct a well-balanced bottom-up analysis from the target environment. The bottom-up analysis is important in the light of performance, future expansion, and porting.

To design components, component interface specifications are defined from lower to higher layers on a step by step basis.

The result of component design for each object-type component is arranged in component layers, and is evaluated including the invoking relationship among them (the concept is shown in Figure 4). When the invoking relationship fulfills the rule mentioned above, the layer is subdivided into detailed layers (see the 3rd layer in Figure 4). If a reverse invoking relationship, such as the one from the 1st to 2nd layers in Figure 4, exists, the rule should be regarded as unsatisfactory and the design must be retried by re-dividing the object-type component. If it is unavoidable to admit the design for some reason such as performance upgrading, it should be remarked on for the component system.

The advantages of the design approach based on the component layer concept lie in that it clearly defines the range of purpose-oriented provision of components and allows us to evaluate the facility of component creation and modifications.
4. CASE STUDY AND REVIEW

This section discusses and reviews the results of our software component development approach applied to three different cases.
These are:
Case A: Process monitoring and control software (mainly FORTRAN)
Case B: Distributed information processing system software (COBOL)
Case C: Microcomputer embedded product control software (C and assembler)

These are medium-sized examples of software of several tens of thousands of steps.

(1) Evaluation of development process

In each case it was possible to conduct the work of component development as an independent process. In the cases of A and B, we assumed that the life of the product architecture would be five and three years, respectively. The development of the components was started about half a year ahead of the product development work. Case C is a microcomputer embedded product expected to grow and change further. We did not therefore fix its product family. Since its development period was limited, we developed components in parallel with the product development work. These cases pointed to the following factors:
(i) It is desirable that the component development group be led by persons with a good knowledge of the applications concerned.
(ii) Librarians should be assigned about 5% of all processes for continued component management.
(iii) The design of the product architecture and the forecasting product performance are much more important when components are developed in parallel with the product concerned.

(2) Evaluation of component layers

Table 1 gives the results of component reuse for case A and Figure 5 shows the percentage breakdown of the use of standard components by layer also for case A. (It must be noted that the number of components was counted for each individual operational component belonging to each object-type component.) Case C showed a pattern similar to that of case A. In case B, more than 90% of the components used were accounted for by those belonging to the product family domain layer. This was because in case B, components equivalent to target environment components were supplied mostly as basic software of a quasi-OS. As is known from these cases, the percentage breakdown of component layers varies with product policies and the scope of the basic software, etc. In each case, the number of prepared components was found to be several hundred.

Table 1  Results of Component Reuse (case of project A)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of standard</td>
<td>489 items</td>
<td></td>
</tr>
<tr>
<td>components prepared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of programs</td>
<td>151 programs</td>
<td></td>
</tr>
<tr>
<td>Frequency of component</td>
<td>2,910 times</td>
<td>5.36 times/</td>
</tr>
<tr>
<td>use</td>
<td>(*)</td>
<td>component</td>
</tr>
<tr>
<td>Usage of components</td>
<td>69% (*)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5 shows that in case A, while target environment and application domain related components accounted for about 50% of the prepared components, their use accounted for as much as about 85% of all the use. The use of these components was quite effective. By contrast, product family domain components were less frequently used, and about half of them were never used in any job. Effective use of upper-layer components is our future challenge.

The component layers were designed to allow the higher layers to call the necessary components only in lower or the same layers. This made it possible to develop the component layers effectively also in the next-generation product family whose target machines and OS were different from those of the previous family. That is, about 30% of all components could be transported to the new product family without changing component design specifications (interfaces), and a considerable number of target environment components were covered by new basic software. As a result, more than half of the components could be used in the next-generation product. Our approach proved to be very
effective with respect to the reuse of documents, e.g.,
component design specifications, as well.

(3) Evaluation of productivity

In all of our examples, the ratio of reuse of components was about 60% or more. (See Table 1 for case A.) From the viewpoint of productivity, however, it seems that the effectiveness can be better evaluated in terms of the frequency of use of each component. In each case, the leaders felt that the labor of making the components would pay off if these components can be used three times or more on the average.

Let us look at one job in case A. Suppose (i) it takes 1.3 man-days on the average to develop one operational component, and (ii) the probability of developing similar components is 0.8 when no special effort is made to encourage the use of standard components. Then, it is estimated that a total of 2,390 man-days (1.3 man-days × 2,910 times × 0.8 = 1.3 man-days × 489 components) can be saved. If the number of jobs utilizing the components increases, the effectiveness of this process will also increase.

(4) Evaluation of product quality

When it comes to product quality, the benefits of our approach can be summarized as follows.

First, the occurrence of bugs was decreased by about 20 to 30% at the time of software system integration testing, thanks to component-conscious design and severe testing.

Second, limiting the size of each component (the 50 steps/module concept) helped us greatly. The evaluation of a certain case showed that the occurrence of bugs was about 8 to 10% for modules of 50 steps or less but about 90% for modules of more than 50 steps.

Third, standardization of the design concept and description method resulted in a standard deviation of the quality metrics of the components (the number of steps, the frequency of comments, the McCabe's complexity measure, etc.) in the range of 2 to 7 while showing small variances with persons and components.

Some problems were pointed out even though very rare, e.g.: a function may fail to perform with the desired response due to wrong usage of components; when a bug is found in the process of product development, it would be difficult to determine whether or not it is attributable to a defect in the components themselves.

5. CONCLUSION

We have discussed our software reusability technique featuring a component development process and component-oriented object layers, designed to promote the reuse of standardized components. This technique has been applied to some projects with excellent results.

We plan to apply this technique to other cases too, thereby making the classification of objects in individual component layers more systematic, and to improve computer support for component management and metrics systems.

ACKNOWLEDGEMENT

The authors would like to thank Y. Ohfude and F. Teshima for their discussions. Further more, they wish to acknowledge the helpful support from J. Inoue and M. Nakashima in gathering evaluation data.

REFERENCES

(2) G. Booch, "Object-Oriented Development", IEEE Trans. on SE-12, 2 (1986)