Management of Object-Oriented Software Components in Distributed Environments

Mika Ohtsuki

Graduate School of Information Science and Electrical Engineering
Kyushu University, 6-10-1 Hakozaki, Higashi-ku, Fukuoka 812-8581 Japan
February 1999

Contents

Abstract

Cooperative software development in distributed environments has become common because of recent popularization of computer networks. In such a development fashion, reuse of software components is necessary as well as in personal or local development, and object-oriented technology is important as its basis. Reuse of object-oriented software components in distributed environments, however, has not become common yet as has been expected. This originates in not only difficulty in understanding characteristics of object-oriented software components whose structures are complicated, but also absence of support mechanisms for obtaining components in distributed environments. In this thesis, first of all, we discuss what are to be handled as reusable object-oriented software components, then propose a management mechanism for promoting reuse of components in distributed environments. We call it a distributed component repository. As fundamental components for class components, we provide an extraction mechanism from source codes, and register a certain library for examination. Furthermore, we consider dealing with knowledge as components, in particular, address to deal with design patterns as components, because they are strongly related with classes and useful to design reusable structures in an application. In this thesis, we propose a framework of structure documents for design patterns, and provide two process mechanisms: a conversion mechanism to browsing format (HTML) and a source code generation support mechanism. The conversion mechanism equals conversion to nodes in the above repository. The source code support mechanism aims at design support as well as comprehension support by relating with source codes in future.

Acknowledgments

For Prof. Akifumi Makinouchi at Graduate School of Information Science and Electrical Engineering Kyushu University, I appreciate his endurance for a long time to supervise and instruct me in this research. Furthermore, I must apologize his worry and trouble for me. And for Prof. Araki and Prof. Hasegawa at Graduate School of Information Science and Electrical Engineering Kyushu University, I would like to express acknowledgment for their appropriate advice and kindness.

For Prof. Norihiko Yoshida at Department of Computer and Information Sciences Nagasaki University, also, I would like to express acknowledgment for not only his generous cooperation and advice in research discussions, but also for advice to my troublesome consultation. Of course, I must not forget Mr. Jun'ichi Segawa, Mr. Shinji Hamada, Mr. Masahiko Sumita and Mr. Shingo Ishimatsu. Without them, our system would have not been built yet. I would like to thank them for their help in implementation of the system and their sharp insights for it.

Then, I would like to thank fellows in our laboratory, who gave a lot of ideas in a lot of discussions.

Finally, I would like to offer my sincere acknowledgment to my family who has supported me for a long time.

Chapter 1

Introduction

1.1 Software Components and Reuse

As software has become larger and more complicated since the 1970s, it has become difficult to maintain productivity and reliability. Software reuse, therefore, becomes more important. Some studies on reuse showed that 40% to 60% of code is reusable from one application to another. And there are benefits in reusing software, which have been argued by many researchers [48, 6, 7, 8]. For example, Sametinger [48] mentions that software reuse improves quality, productivity, performance, reliability and interoperability.

There are many viewpoints on software reuse. Basili and Rombach see software reuse as the use of everything associated with a software project, including knowledge [2]. The knowledge can be represented in such forms as algorithms, software architectures, design patterns. Sametinger does not consider such knowledge as components to be reused [48]. Tracz defines reuse as the use of software that was designed for reuse [58].

In this thesis, we see reuse as using resources associated with a software project. We define software components as all information generated, modified and referred through an application development process, including design ideas, concepts, requirements, specifications, designs, source codes, algorithms, software architectures, and patterns.

As Sametinger already mentioned, algorithms, software architectures and patterns, in particular, are not available in digital form and they are not considered to be reusable components. Some framework is necessary to deal with them as components.

The object-oriented paradigm fosters software reuse. The term "object-oriented paradigm" is explained in the next section. Before the object-oriented paradigm appeared, components include requirements, specifications, designs, memos, source codes, test codes and results, method libraries, algorithms and data structure and software architecture. Components introduced after the object-oriented paradigm include classes, frameworks, class libraries, designs using object-oriented modeling methods such as OMT (Object-oriented Modeling Technique) [46], architectures, design patterns and idioms.

We call the latter components "object-oriented software components". We focus on reuse of the object-oriented software components, in particular, classes and design patterns. Classes are basic constituent elements of other components. Design patterns are descriptions of design ideas.

1.2 Object-Oriented Paradigm

The object-oriented paradigm was proposed to decrease the cost of software development and to offer reliable components. The object-oriented paradigm, in other words, was proposed as the way to arrange complex and large software structurally to make it easy to understand.

In the object-oriented paradigm, variables related to each other are collected into one structure with operations related to them as an interface, and the structure is called an "object". In other words, the term "object" means a package of a data structure and behaviors. The object-oriented paradigm is a paradigm to make application development easy, by minimizing relationships among structures which consist of objects. Due to independence and modularity among structures, effects are expected such as, readability, testability, and reusability of the structure as components.

There are many object-oriented programming languages [56, 39, 31, 60, 19, 18, 24]: pure object-oriented programming languages such as Smalltalk [56] and hybrid object-oriented programming languages such as C++ [19], which are usual procedural programming languages extended with object-oriented functions.

In both kinds of languages, such relationships as dependencies and method calls can be separated from each other and arranged by collecting procedures and data structures dispersed in procedural programming languages into objects or classes as templates of objects. Furthermore, the objects can be easily reused by such new concepts and functions as encapsulation, inheritance, dynamic bindings and polymorphism. In other words, to accomplish the modularity, they are used.

Encapsulation is a mechanism which allows accesses to variables and methods of an object only through interface methods defined as public, therefore prevents the occurrence of unexpected relationships among objects. Inheritance is a function to extend a class, that is, to add interface and change implementation. The derived class shares interface and implementations of the base class. Dynamic binding is, in a language with type checking, when a variable is declared as a class but can be assigned an object of any derived class at runtime, to deal with the variable as the derived class object. In other words, when a method call for the variable is called, an implementation of the derived class is executed. It can realize polymorphism. Polymorphism means behaving in different ways in the same name. It includes derived classes with a common interface in the base class but different behaviors using dynamic binding.

Object-oriented programming languages foster software reuse by providing concepts and functions mentioned above. By using object-oriented programming languages, in practice, more reusable components are provided. They include componentwares which are components especially designed to be reused, and frameworks which contain structures to be exchanged later by using polymorphism.

However, object-oriented programming languages do not solve such reuse issues as how to construct reusable components and how to use them. Object-oriented programming languages provide only functions to construct "objects", but not principles for designing to make them easy to be reused. As application becomes larger and the number of objects,

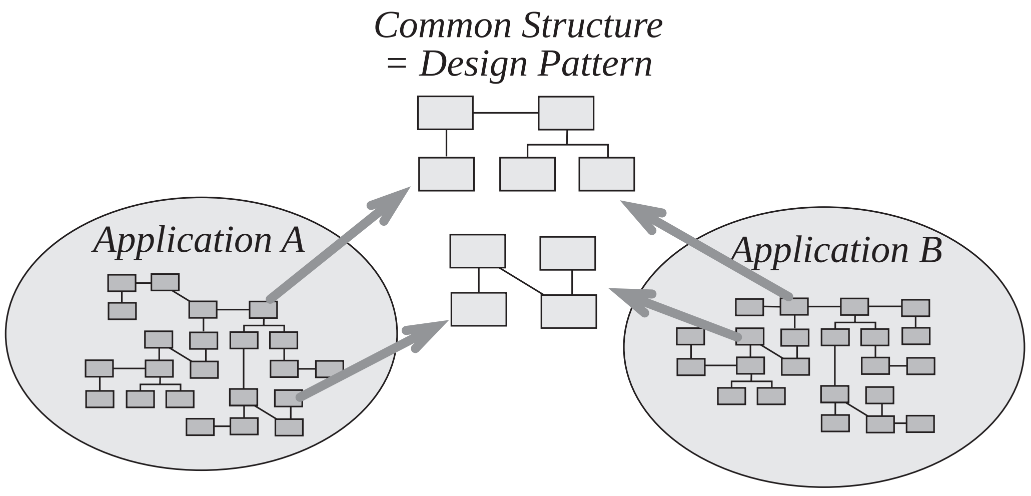


Figure 1.1: Concept of design patterns

classes increases and relationships increase, it become more complex. It results in difficulty in finding out what to be reused, and prevents the use of reusable components.

For decreasing difficulty of design especially for making reusable components, and difficulty in understanding an expanding number of objects and complicated message passings among them recently, many analysis and design techniques such as OMT [46] are proposed and used widely [46, 5, 49, 15]. These are different from each other in their expression ability and have merits and demerits. Currently, UML [44] is proposed to be a framework for integrating them by OMG (Object Management Group). These methods are almost independent from object-oriented programming languages. They, therefore, can be applied to usual procedural programming languages with simulating several object-oriented functions.

However, object-oriented analysis and design methods do not provide concrete principles for making reusable structures either. Modeling methods which can express layered nested structures are useful for software reuse, because an application can be separated and expressed as layered nested structures. UML proposed to integrate other methods also provides the function.

In this thesis, we focus on knowledge called a “Pattern Language” [16, 62] which is collecting attentions in the object-oriented design area recently. A “Pattern Language” is a description of structure and solution pattern appeared commonly in applications, and is generally independent from the problems which the application solves (Figure 1.1). “Pattern Languages” are classified into the following: “Architectures”, “Design Patterns” and “Idioms” [16].

“Architectures” are roughest structural patterns of software. For example, they include MVC (Model View Controller). “Design Patterns” are finer patterns than “Architectures”. They consist of classes removed overlapped ones. For example, “Iterator Pattern” provides common interface for iteration among several aggregates. “Idioms” are very concrete codes to realize an general function using a certain language. They, therefore, are also called “Coding Patterns”. Ones for C++ are known well. For example, there is a coding pattern to realize reflection which is used to obtain an object’s class name from the object.

When using them for design, generally, “Design Patterns” can be used for implementation of an “Architecture”, and to use “Idioms” for implementation of a “Design Pattern”. In these components, in particular, “Design Patterns” are effective principles to build reusable structure such as frameworks [43]. Because most of the current design patterns aim to decrease dependencies among objects using object-oriented facilities such as inheritance and dynamic binding. W. Pree explains how to decrease dependencies by design patterns into several essential patterns, and points out they are effective for making hot-spots which are exchangeable modules of a framework [43]. Here frameworks are a set of classes designed to be adaptable structures for common problems in a certain problem domain. Application-dependent parts of a framework are designed as classes or modules which can be replaced later. The exchangeable classes or modules are called “Hot Spots”.

1.3 Object-Oriented Software Components and Relationships among Them

Object-oriented software components in this thesis are as follows:

- Source Codes

  Concretely, they are files containing codes written in object-oriented programming languages. These are dependent on a concrete language. They can have various extents of dependency on an application.

- Concepts for handling source codes

  The following are conceptual units for classifying and packaging source codes. These units are used for an application design. The most fundamental unit is class. All others are composed of classes. In more concrete, they contain references to classes. Because they are ways to classify source codes, they are dependent on a certain language.

  - Classes

    They are fundamental units in object-oriented software. They are depend on a certain language and the extent of dependency on an application is various, as same as source codes.

  - Modules

    They are sets of classes which are classified and packaged according to a certain function. They can be contained in both libraries and applications. How to classify classes is not fixed and dependent on the designer.

  - Frameworks [30]

    They are parts or whole of an application or libraries, which are applicable to other similar problem domains with replacing partial implementation (classes). They consist of several classes strongly related to each other.

  - Libraries

    They are sets of classes separated and collected from applications. The target of this research is not method libraries but class libraries consisting of classes. A library can contain several frameworks.

  - Applications

    Applications can consist of either a class or thousands of classes. The target of this research is such larger applications as which consist of thousands of classes. Such applications can consist of several modules. They also can use classes and frameworks in libraries. They are dependent on problems of itself.

- Design Ideas and Concepts

  These are components for design principle and program understanding support of source code components, which collect attention. They are provided on papers in plain text generally. Frameworks to use them efficiently on computers in electric forms is necessary.

  - Algorithms and Data Structures [32, 33, 34]

    Ideas to be foundations for designs of data structure and methods of classes. Independent from both languages and applications.

  - Software Architectures [69, 70]

    Ideas to be foundations for designs of rough structure of applications. Independent from both languages and applications.

  - Pattern Languages [16, 62]

    ◇ Architectures
    The most language-independent, and the most difficult to formalize.

    ◇ Design Patterns
    Dependent partially on functions provided by object-oriented languages, such as inheritance.
    Independent from concrete languages and applications.

    ◇ Idioms
    In other words, coding patterns. Language-dependent but application-independent.

- Other General Description Documents

  They may be formatted in each company. “Literate Programming” [35] is a trial to integrate source codes and any kinds of documents such specification, design concepts and memo. A good example is NoWeb [29]. From a view point of the object-oriented paradigm, model descriptions by OO modeling techniques should be paid attention. Some of them are designed for certain CASE tools and dealt with in electric forms. In CASE tools they can be checked for consistency.

  - Requirements
  - Specifications
  - Designs

    Model descriptions are important. Usually, the descriptions are language-independent.
  - Test Codes and Results
  - Memos

Table 1.1: Components classification

	Executable	Application Dependent	Language Dependent
Libraries	○	×	○
Frameworks	○	△	○
Applications	○	○	○
Algorithm and data structure	×	×	×
Software architecture	×	×	×
Architecture (Pattern Language)	×	×	×
Design Patterns	×	×	△
Idioms	×	×	○
Specifications	×	○	×
Requirements	×	○	×
Test codes and results	△	○	○
Memos	×	○	△

Components listed above can be classified as Table 1.1

Next, how components mentioned above relate with each other is considered. The followings are examples of such relationships.

When deciding the rough structure for solving a problem, usual software architectures and architectures in pattern languages can be reused. Which one to use is decided by analyzing requirements roughly. In general, such large structure consists of sub structures. When designing more detailed structure to realize the structure, the sub structures can be realized by using finer architectures and design patterns. What to be used is by picking up finer requirements. Design patterns are basic concepts to construct frameworks in the application. Some of the frameworks, such as ones included in a GUI library, may be reused in the application. Of course, others may be expanded from prepared ones or created anew according to requirements. Design patterns are structures only to increase availability and maintainability. Therefore, data structure and algorithms to solve the given problem are necessary. Data structure and algorithms as components, generally, are provided in the form of documents. Some of them may be implemented as a library. You can use them in libraries or build them up anew.

Figure 1.2 shows the relationships around an application in focusing design patterns and classes. In this Figure, the boxes put in *Application* box are attributes indicating aggregates. For example, *Classes* is an aggregate attribute including references to classes. Thus, the arrows from the box to *C (Class)* circles express the references. The boxes of *Modules*, *Frameworks*, *IKCs* and *Documents* are same. Here, "IKCs (Instantiated Knowledge Components)" in the figure are concrete instances of knowledge components, which are necessary to relate the knowledge components to an application. In this research, an instance of a design pattern is called "IPS (Instantiated Pattern Structure)".

Keeping generated relationships as cross-references can be very effective for understanding an application. The generated relationships are shown as Table 1.2.

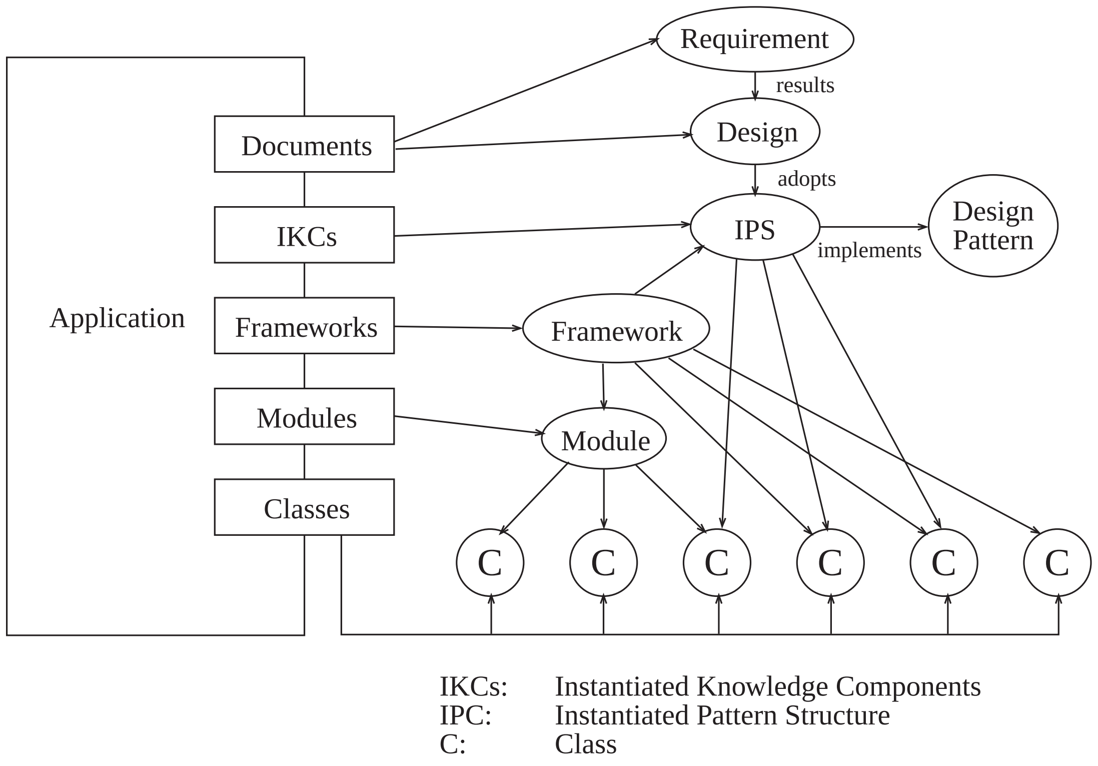


Figure 1.2: Relationships among components

Table 1.2: Kinds of relationships among components

among any documents	refers	referred
among requirements and designs	reasons	results
among concepts and collections of class	implements	implemented
among coarser concepts and finer concepts	adopts	adopted
among classes	refers inherits	referred inherited

1.4 Approaches to Issues of Software Reuse

Reuse issues are categorized as follows: software development for reuse and software development with reuse.

Gibbs argues that a well organized software community (groups of designers and developers sharing knowledge and experience) is necessary for software reuse in [23].

"In our ideal scenario, applications would be based on generic software components accumulated by a software community familiar with the application domain. To build a new application, a developer would collect requirements according to an existing, well-defined model of the domain, select generic software components according to these requirements, and initialize and compose the selected components to construct the running application. By analogy, lawyers would like to handle all regal cases as though they were slight variations on textbook cases."

Gibbs defines reusable object classes as experience, then argues that software information systems for managing and accessing to class collections is necessary.

It is difficult to use them effectively without understanding assumptions of the class collections as mentioned in [22]. Research in the education of object-oriented design techniques reports that for using frameworks (reusable structure consisting of several classes) it is effective for beginners to learn appropriate application examples [51]. [51] also reports that learning using structural information (hierarchy-based, including design patterns) is not effective for beginners because of their unfamiliarity to such information, but would be necessary to use frameworks more effectively.

Based on these studies, in order to reuse such components as frameworks effectively, it is necessary to provide an environment to refer to descriptions of assumptions, ones of structures and application examples. In other words, it is necessary that descriptions and examples are formatted in a form that is easy for a user to read, and he/she can understand the semantics structure of them by following them. It enables he/she to obtain them in an electric form.

Most of CASE tools can relate designs/models to source codes by generating source codes continuously. Products in CASE tools are stored in a storage called a "repository". Some of the tools provide interfaces for following relationships among components. The tools aim to manage every application through its life cycle, but not to support understanding and use of such reusable components as frameworks.

On the other hand, as networks are popularized and techniques related to distributed environments are developed, independent distributed systems (peer-to-peer servers) will increase later from now. Software communities can be built up in a cooperative community of several corporations as a "virtual corporation". Furthermore, broad on such distributed systems beyond it. In such environments, a mechanism for sharing components over usual local and centralized repositories.

CDIF (CASE Data Interchange Format) is proposed for interchanging data of CASE tools among corporations. However, it is a standard to exchange data in a certain repository into a common format. Relationships of data in CASE tools are usually closed inside the repository. In other words, to handle relationships among repositories connected to each other but managed independently, in addition to the standard format, a framework for dealing with external relationships is required.

In this research, we consider the framework for dealing with distributed and loosely coupled data and relationships among them, then aim to realize the mechanism as a prototype. The mechanism is called a "distributed component repository" in the rest of this thesis. The distributed component repository is shown as Figure 1.3 conceptually.

The following issues must be considered.

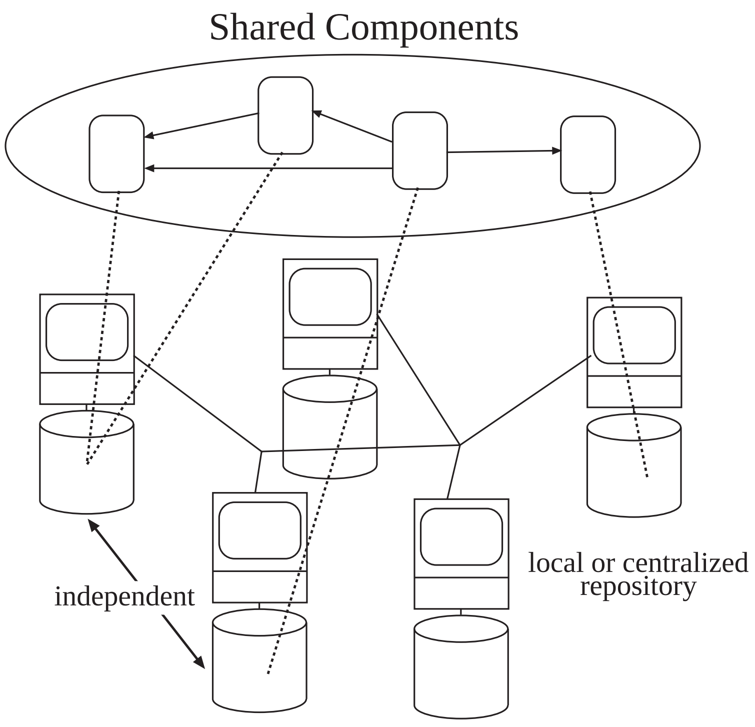


Figure 1.3: Concept of shared components in distributed environments

- Consistency management of data name space

  Providing a mechanism for naming distributed components uniformly.

- Distribution transparency

  It is the ability to obtain distributed components without concerning their location. Providing a mechanism for corresponding component IDs to their locations and proposing them to a user.

- Management of mutual relationships among data

  It must be solid for location changes.

- Configuration management

  Version management and configuration management are necessary because components are updated concurrently.

First of all, we aim to address the above issues by implementing the distributed component repository to deal with only class components. A class is the fundamental unit for dealing with object-oriented software, but all source codes are not separated into the unit class. Dealing with classes as the fundamental units of object-oriented software independent from source codes makes it easy to deal with relationships to other conceptual components and documents. For example, it becomes easy to see that a certain library contains A class and B class or a certain function of a certain specification is implemented in a certain class.

In [23] Gibbs argues that a class should support multi views. In this research, however, we consider a view point for sharing. Class components can be accessed in an access level which a developer requires. In other words, when he/she uses them as off-the-shelf parts without modification, he/she gets only public interfaces of them, and when he/she needs to refer or modify inside codes of them, he/she can get private implementation of them.

On the other hand, as for the issue of how to build reusable structures, Garlan et al. proposes for "developing sources of architectural design guidance" [22]. As one approach for the guidance, he mentions design patterns. However, Usual CASE tools do not provide a framework for dealing with such unformatted information as components.

This thesis focuses on design patterns as knowledge. The reason for this is that design patterns are can be used because they are language-independent, not far larger than classes and strongly related to classes as basics for designing frameworks. In other words, using design patterns as components contributes to the comprehension of frameworks by relating them to their classes based on design patterns.

For this reason, it is necessary to convert them into electric forms in order to be processed in computers. The electric forms can be also used for code generation, then it foster software reuse. Additionally, we aim to foster understanding frameworks by relating them to classes based on design patterns.

From the above consideration, the rest of this thesis is separated two themes; one is construction of the distributed component repository for class components, and the other is a proposal of a structured document framework to deal with design patterns as components and construction of mechanisms to process them.

First, as for the distributed component repository for class components, we describe how to construct the distributed component repository, then describe how to extract class components from source codes and how to handle them. Secondly, as for design pattern components we begin by discussing how to describe design patterns as structured documents and propose a framework. Then we propose a source code generation support system for design support and relating them to class components. At the last, we analyze and evaluate all the results to make a conclusion.

The component management server as a basic system of the distributed component repository for class components and a structured document framework for design pattern components are language-independent frameworks. The above functions are implemented as exchangeable client programs or modules. Therefore, we can implement ones for any languages later.

When implementing our experimental prototype system in this research, the class component extraction mechanism targets C++, and the source code generation support mechanism targets Java. As for the former, because usual libraries are provided in C++, we considered that it is suitable as subjects for the class components extraction mechanism. As for the latter, because Java is simpler than C++ as object-oriented language, we considered that it is suitable as subjects for the source code generation mechanism.

However, The selection of languages is neither essential nor final one. When future integration and practical use of the development environment, any languages could be integrated.

Based on the above discussion, this thesis consists of following chapters.

- Distributed Component Repository
- Class Components
- Design Pattern Components
- Source Code Generation Support
- Conclusion

Chapter 2

Distributed Component Repository

Our aim is to support cooperative development of object-oriented software in a distributed environment. Suppose a team of several developers working together to build a software system. When its members share libraries and components, the following problems must be addressed for distributed cooperative work, in addition to problems already solved in current software development support:

- Integrated Management of Various Kinds of Components

  As mentioned in Chapter 1, at software development, various kinds of components are necessary: not only source codes but also specifications, requirements, memos, test codes and so on. There are several proposals to integrate such various kinds of components, for example, one from Tektronix [4], which tried to connect components with each other using the hypertext mechanism.

- Transparency Management

  Transparency means the ability to access information without paying attention to physical location, as in distributed computing, distributed database and so on. In the distributed computing field, OZ++ [26] for example.

- Version Management

  This is an old problem in software development, therefore, there are many research results and commercial products. Representative version management systems are SCCS [45], RCS [57] and so on. Furthermore, in practice, CVS [14] are used as a version management system for concurrent development.

- Access Control

- Cooperative Work Support

  Until now, cooperative work support has been considered in the fields of Groupware or CSCW. Recently, there are several trials [61, 36] for applying the products to software development.

In these issues, we focus on the first and second ones, integrated and transparency management. In transparent management, it is crucial not to let users be concerned about actual (physical or geological) locations of components. There are several distributed components

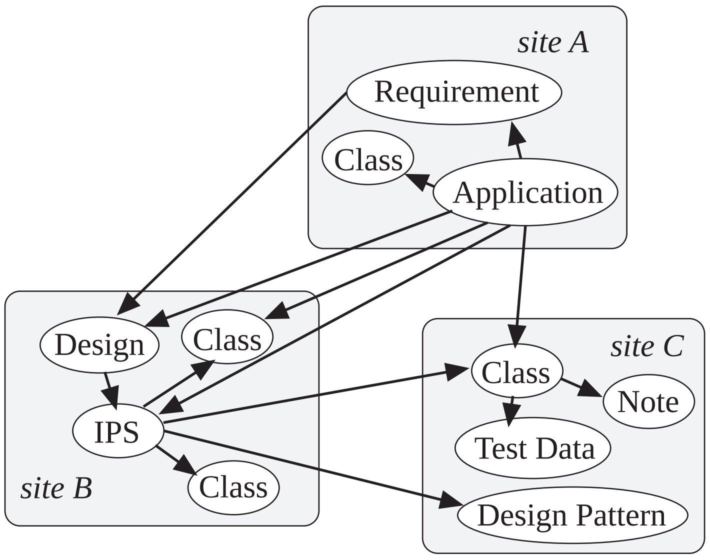


Figure 2.1: Distributed hypertext structure of various software components

management systems: OZ++ [26] which aims to realize distributed computing, and RIG which aims to realize interoperability of libraries (the sharing of assets among reuse libraries [9]) and so on. On the other hand, there are several proposals for integrated CASE tools environments: Chimera [1], PCTE (Portable Common Tools Environment) [68], CAIS (Common APSE[1] Interface Set) [41].

Furthermore, among these above problems, we also consider integration of management. Various forms and relationships of components are unformatted. Therefore, it is difficult to manage them formally by using databases. A hypertext is the most effective alternative to manage such unformatted information. In other words, using the hypertext mechanism, we consider managing such various components distributed among remote sites and related with each others, as illustrated in Figure 2.1.

2.1 Hypertext

A hypertext [52] stores information in the form of network consisting of nodes connected by links. A node is an unformatted container of an information unit, which may be anything that can be stored in computers, for example, texts or image data. A link represents an arbitrary relation between nodes. Searching a node is done by tracing links successively in hypertext network. Using a hypertext, software components can be managed integratedly, by nodes representing components and links representing their relations.

In the standard reference model of hypertext called Dexter [25], a hypertext consists of three layers: the storage layer, the run-time layer and the within-component layer. The storage layer prescribes basic structures of the hypertext, i.e. atomic components, composite components and behaviors of links. The run-time layer prescribes interface from applications

[1]APSE stands for “Ada Programming Support Environment”

to the storage layer. This paper focuses on the within-component layer, i.e. how components should be stored in nodes and how their relations should be formed as distributed transparent links.

Transparent implementations of distributed hypertexts have already existed: The WWW (World Wide Web) is a typical example.

2.2 Location Management

Some mechanisms of WWW in part are utilized to implement a distributed hypertext. Node locations on a distributed environment are identified using URL (Universal Resource Locator) of WWW. However, WWW enables us only to trace links but not to manage node locations. Furthermore, the indicated location is mere the location of the file, and does not reflect such structures as modules. For example, in CORBA [40], an identifier consists of a repository ID, an interface ID and so on. In OZ++, a scope called school is used. In RIG, currently a URL is used but URN (Universal Resource Name) will be used in future.

Our approach is similar to CORBA. A table is prepared to manage the names and the locations of nodes using a database in order to locate nodes efficiently. In more detail, the table to made a node ID correspond to a URL of a prepared node file, or to a CGI (Common Gateway Interface) program which generates an HTML (Hyper-Text Markup Language) file as the node.

All the generated node HTML files contain links corresponding to dependencies of them in the form of anchor tags using CGI. The form is not URL itself in order to make location management easy when the node files are relocated.

CGI is an interface mechanism to call external procedures from HTML files. A CGI procedure with an argument indicating a component ID is placed at the entry of a link, and the procedure resolves location of the component by looking up the Component-URL table.

Every site, i.e. every workstation in the network, has a table and its management routine which acts as a server for other parts of the system. There is neither a centralized server nor a database in the system, but all the site servers collaborate in a decentralized manner to achieve overall management of nodes and queries in the distributed hypertext.

2.3 Representation of Distributed Hypertext Nodes

An overview of the systems is shown in Figure 2.2, where "httpd" (**h**yter**t**ext **t**ransfer **p**rotocol **d**aemon) is a WWW server, "cmpd" (**c**omponent **m**anagement **p**rotocol **d**aemon) is a table management server to manage the Component-URL table, and "add_component", "list_component" and "query_component" are client routines respectively. "Add_component" is the input transformation part in Figure 2.3, which analyzes a given component resource and extracts necessary information, then creates corresponding components including links automatically. Both "list_component" and "query_component" are the output transformation part in Figure 2.3, which deal with reference requests and inspection requests. It converts components into HTML files (if necessary) or other requested hypertext nodes.

The following protocols are delivered between the server and the clients,

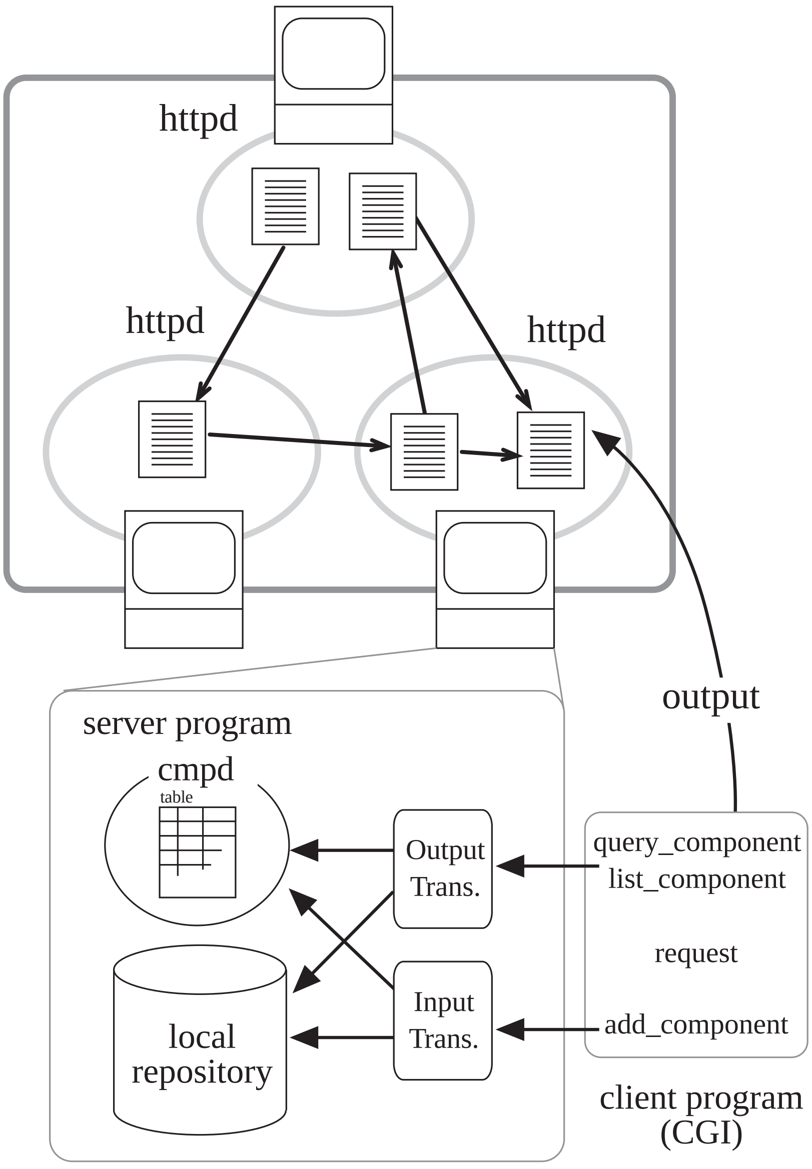


Figure 2.2: Structure of a component management server

- ADD, UPDATE, DELETE
  Registration/update/elimination of type/class information

- LIST, QUERY
  List/query request of type/class information

As mentioned above, there is no centralized server in the system, and all the site servers collaborate in a decentralized manner to achieve overall management. This is a reasonable approach because the size of distributed environments gets much larger, and centralized management cannot work well. WWW servers provide functions to make links transparent, however, we ourselves must provide functions to make the Component-URL tables transparent.

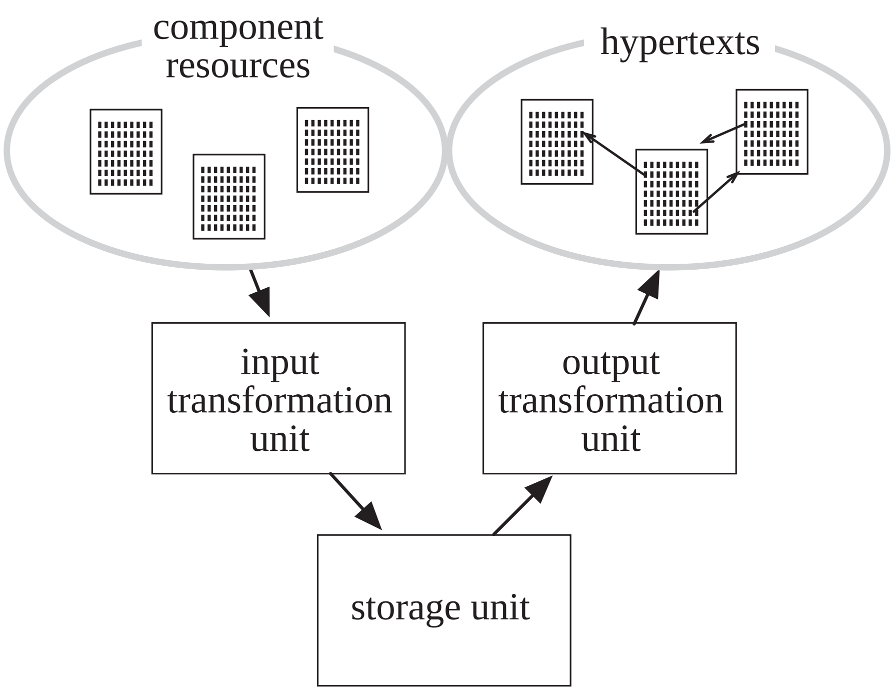


Figure 2.3: General processing model for hypertext

Therefore, it is crucial to maintain consistency of the Component-URL tables of all the sites. In other words, it is necessary to provide protocols to realize transparency by exchanging data among distributed repositories in addition to the protocols for server-clients mentioned above.

In general, there are several methods of maintaining distributed consistency of a shared resource:

- One site serves the resource (server approach),
- Each site has a read-only copy of the resource (copy approach),
- Each site has a partial copy cached on demand (cache approach).

One method should be selected out of these according to access patterns to the resource. As for the Component-URL table, writing accesses for addition and update are rare. Reading accesses for reference are very frequent. They can occur at the search request as well as at the addition and update requests (due to parsing given source code files) and at the compilation (due to collecting constituent classes). In this situation, the copy approach is the best. Identical copies of the table are placed on all the sites. There are protocols between servers as well as the ones between a server and clients:

- ADD_FROM_PEER
- UPDATE_FROM_PEER
- DELETE_FROM_PEER

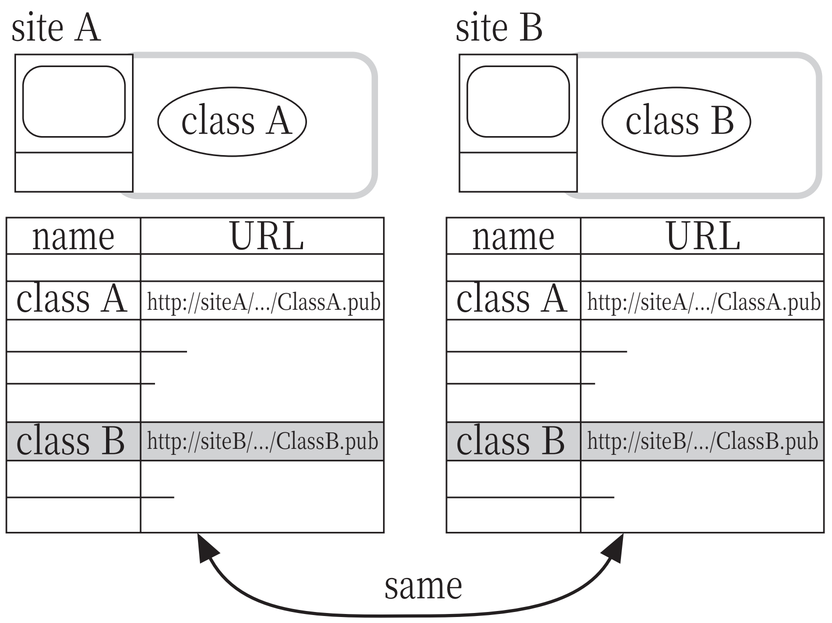

site A
class A
site B
class B
name
URL
class A
http://siteA/.../ClassA.pub
class B
http://siteB/.../ClassB.pub
name
URL
class A
http://siteA/.../ClassA.pub
class B
http://siteB/.../ClassB.pub
same

Figure 2.4: Overview of distributed tables

On the network, there must be dangers of message loss and collision. There is no collision on “update” and “delete” of a class, because only the owner’s site can do these, while there is a possibility of name collision on “add”. The system does not try to resolve this, but only invalidates “add” and reports this to the users who issue “add” simultaneously. Resolution of the collision is left to negotiation between the users, and it can be done by providing an aid such as an electric discussion system outside of the system.

2.4 Prototype Implementation

Our prototype targeted C++ class components as the first step towards our aim. A class component is converted to HTML files at registration, and saved into an accessible data area of a WWW server. Details of the extraction and registration mechanism are described in Chapter 3.

Therefore, the location management table manages URLs of these HTML files, and the client program which resolves component ID and returns an HTML output does not convert for the output but returns the file corresponding to it as is. Moreover, it is implemented as a script which indicates URL in the form of “Location Header” obtained by requesting a component ID to the table manager. HTML files generated at registration already contain links represented by CGI using the program with a component ID argument. Furthermore, the servers synchronize by exchanging information of tables among them as shown in Figure 2.4. The module structure of the servers is shown in Figure 2.5.

Details of C++ source code components are described in Chapter 3 as follows: how to analyze source codes, what to extract from them and how to generate HTML files from them.

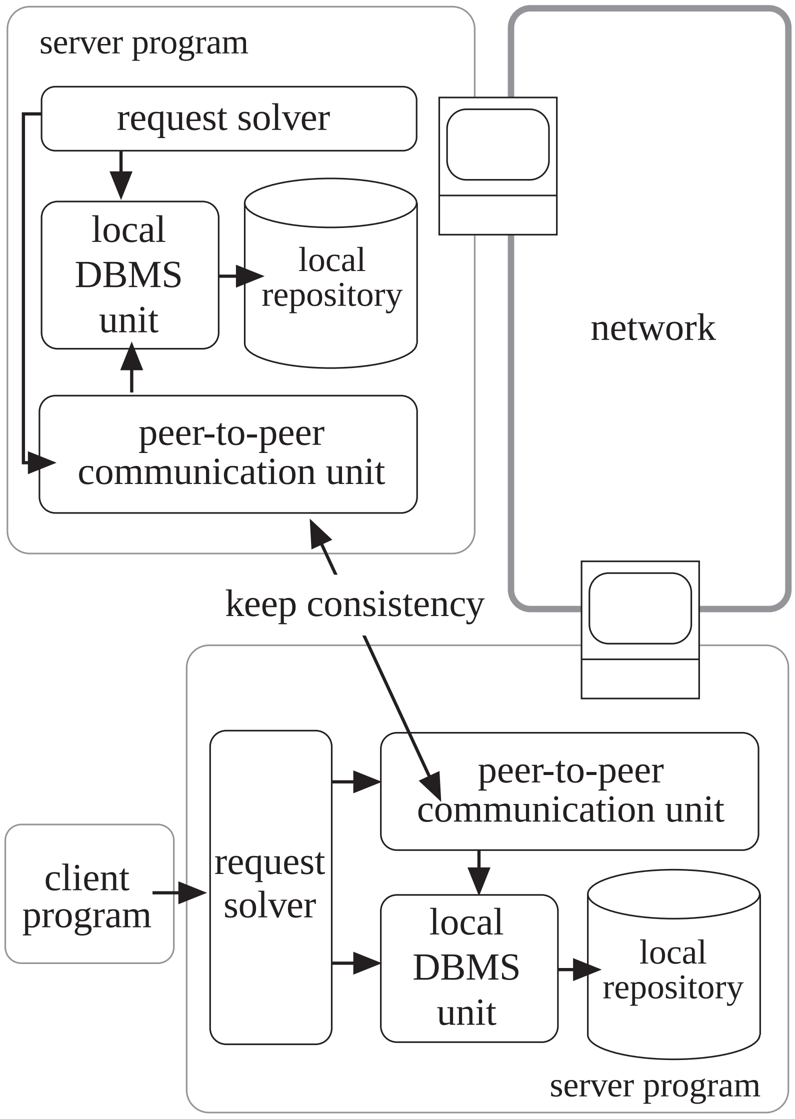


Figure 2.5: General distributed repository system model

2.5 Related Works

Related works that attempted to apply hypertexts to component management for software development support include the following examples:

RIG RIG(Reuse Library Interoperability Group) [9] aims at interoperability of libraries. It manages to share libraries in distributed environments. In other words, the unit of its component is a library, therefore coarser than components we deal with. And kinds of libraries are limited to object-oriented components.

RIG manages library components using a virtual repository called NHSE (National HPCC[2] Software Exchange). NHSE works in the same way as our management system: name assignment and resolution, exchange and interpretation of catalog information and so on.

Dynamic Design Dynamic Design developed at Tektronix [4] is a CASE environment for C programs. In its hypertext, nodes represent requirements, specifications, design notes, design assumptions, comments, sources, objects (machine codes), symbol tables, documents and reports, and links represent relations of "leads to", "comments", "refers to", "calls procedure", "follows from", "implements" and "is defined by".

The Dynamic Design is built on a virtual machine for hypertexts called Hypertext Abstract Machine (HAM) [13]. HAM manages and handles graphs, contexts, nodes, links and attributes as elements of a hypertext. A graph represents a network structure consisting of nodes and links, and it is used for visualization of the network structure. A context represents the order of nodes, and it is used for version management. An attribute is an annotation for other elements. Those graphs and contexts are usefull concepts for handling nodes and links. However, HAM has a centralized management mechanism, and therefore does not suit distributed transparent management in wide-area network.

Chimera Chimera [1] is a part of the Arcadia system which is an integrated distributed development support environment for Ada programs. It provides interfaces to development tools using a hypertext as an abstract data type, and it acts as a middleware system to deal with contents of distributed component databases. As opposed to our system, Chimera focuses more on hypertext modeling in heterogeneous distributed environments and implementation of application programming interface (API), rather than how to manage software components and their relations in an integrated and transparent fashion in distributed environments.

2.6 Concluding Remarks

To provide transparent mechanism for connecting a lot of kinds of components to each other in distributed environments, we have implemented a distributed management system of object-oriented software components presented in this chapter on Sun SPARCstation with Solaris OS. And we have realized the system that several machines in the network share the ID-URL table. This system is actually used for managing classes in the GNU C++ library and ones in our system itself. Furthermore, by cooperating with the source code components generation mechanism, it can handle the components transparently.

The system is still the first prototype, and the following issues should be examined:

[2]High Performance Computing and Communication

- Representation of other functionalities required for a repository

  Implementing ownership management, access control and version management.

- Integrating more abstract components such as design patterns and frameworks.

  The current prototype targets only class components based on C++, but, is enough to make it easy to refer from other documents such as specifications to class components in distributed environments. The management mechanism for other components is the same for class components. The system must be designed so as to exchange input and output transformation units flexibly. For implementation of such design, using design patterns such as Abstract Factory may be effective to absorb influence by class exchange.

  We have been extending our system to store design patterns and to aid source code generation from the patterns. This is described in Chapter 4.

- Introducing cache mechanism

  The current prototype is not efficient, because, at each reference to a component, the component must be obtained from network. Therefore, it is necessary to extend the component management system to cope with cache mechanism.

- Applying the system to real software development to evaluate it.

Lastly, we consider re-implementation using Java language for increasing portability.

Chapter 3

Class Components

In this thesis, a class is a fundamental component for object-oriented software components. It is an element of several kinds of structure components to handle an application and a library structurally: frameworks, modules and instantiated design patterns. In addition to them, any documents can refer to a class. Several components consisting of class(libraries, frameworks, modules and instantiated design patterns) are future issues. In this thesis, we concern application components in order to execute an application making test in practice.

It is able to generate class components from scratch. However, we consider providing a mechanisms to extraction class components from source codes written in a certain languages and construct relationships among them. It aims at fostering reuse of existing libraries as components. Therefore, we select C++ as the target language for extraction.

Class components have relationships with not only original source codes but also machine codes compiled form the source codes and construction information for generating the machine codes. For uniform interface, the class-related components must be "browsable" in the hypertext format. Furthermore, they must be obtained in compilable format. There are requests to each component as a hypertext node.

- Class Components
  - They must have an ID and interface information
  - They need to express different interface views according to access authority.

    Two views (nodes), public and non-public interfaces, are provided. The public interface node includes interfaces indicated as public generally. The non-public interface node includes interfaces except for them, indicated as protected or private, for example.
  - They must be connected corresponding to dependencies.
- Application Components
  - They must have the necessary information to build up an application.

    There must be a starting point class of an application. Development and test environment information are necessary.
  - They need to be related to constituent classes.

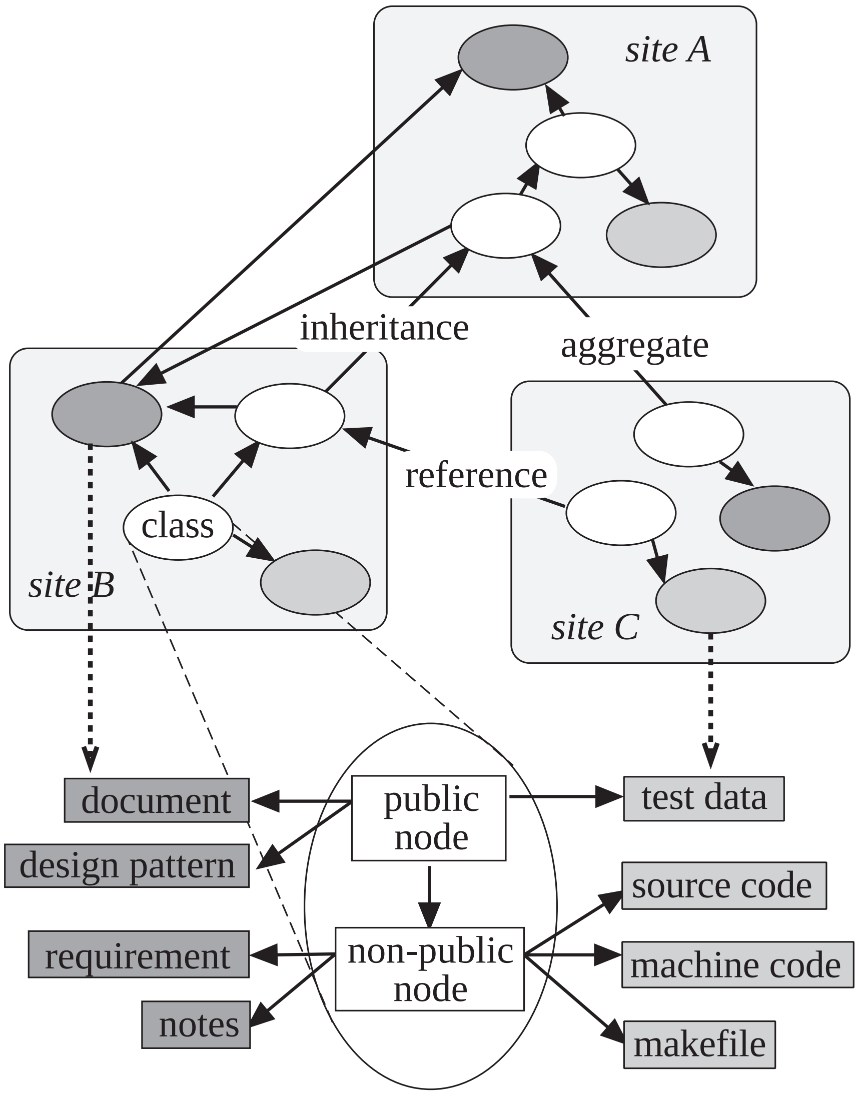


Figure 3.1: Hypertext structure of class-related information

- Source Codes, Machine Codes

  They must be provided in formats which can be processed (compiled and executed) in computers. To handle machine codes, problems of machine-dependency must be solved.

- Construction Information

  It is how to construct a class from several separated source code files in such a language as C++. It is described for each class.

From the above requests, nodes and relationships among them are shown in Figure 3.1.

It is necessary to provide both a hypertext generation mechanism for the stored class components and a mechanism to construct an application for using them. For application

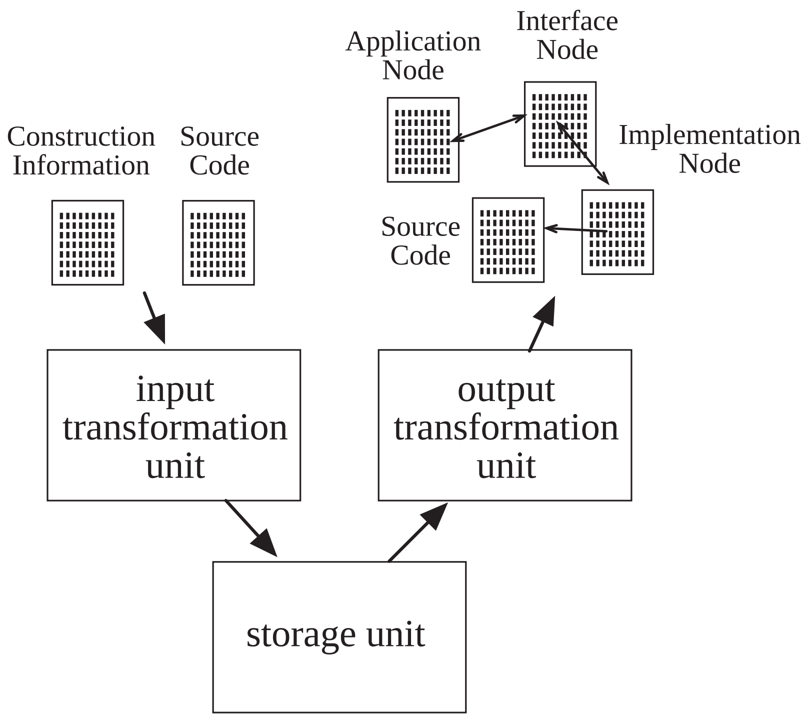


Figure 3.2: Process model for class-related components

construction, it is necessary to keep construction information and to provide an input mechanism for the information. An application component contains only the starting point of the application which is necessary to build it up. Representing the application components as hypertext nodes and relating them to others are future issues.

In the following sections, what information to be extracted for expressing class components as hypertext nodes, when to extract it, and how to generate application automatically are discussed. Implementation of our prototype for C++ is described in the following section.

3.1 Hypertext Node Generation

The following information in source codes is necessary to be reflected into class components.

- Access level of a class

  For encapsulation. 'Public', 'protected', 'private' and so on. There are more kinds of access level in some languages and fewer in others.

- Name

  Identifier of a class.

- Interface information

  - Member functions and member variables

  - Access specifiers

    For encapsulation. 'Public', 'protected', 'private' and so on. There are more kinds of access specifier in some languages and fewer in others.

- Dependencies

  Inheritance, whole-parts and reference relationships

The above information must be updated corresponding to updating source codes.

Figure 3.2 shows an overview of the system which saves source codes as components and shows them as hypertext nodes referred by users. Frequency of update determines the method of converting and keeping components: extracting at the input transformation unit and generating and keeping class components inside the repository, or keeping source codes as is and converting them at the output transformation unit for each request. In other words, if reference is more frequent than update, conversion should be done at registration. If not, the conversion should be done at request. If the target components are enough stable and updated not often, they should be extracted and generated at registration. On the other hand, because application-dependent or unfinished components are often updated, they should be converted at request.

Dependency Linkage The generated hypertext nodes of class components must contain links to others, corresponding to dependencies of the class. Links correspond to identifiers of class components. Hypertext nodes, in practice, are outputs in the form of HTML. In the HTML outputs, CGI anchors are placed at the positions of links, to obtain a target class component from the repository using a given component ID, and convert it into a HTML output representing a public interface node. The reason for not using an absolute address of URL rather than CGI is to make the management of transparency easier.

3.2 Application Construction in Distributed Environments

There are two alternative ways to construct an application in distributed environments: using distributed computing, or gathering components and constructing it on a site. In the former, the constructed application is executed on a distributed system, therefore, machine dependency does not matter.

The latter is our target. In this case, a system which gathers components corresponding to dependencies among them and arranges them automatically to construct correctly is necessary.

On application construction, a compiler must acquire dependency information among class files. There is a UNIX tool named "make" [20] for this. Dependency information is described in a file called "Makefile," and "make" compiles the source files according to the content of the makefile. A programmer must understand and describe all the dependencies among files in an application to use the tool. Distributed environments make application development more difficult than usual. Such an application development method requires functions to locate components transparently and to collect components automatically.

In our system, the class components, as mentioned above, contain dependency information among components, making it easy to extract information. Because there are several languages which separate interface descriptions and implementations, a component author must give such concrete dependencies among source codes for each class.

A centralized makefile, i.e. a makefile for a whole application is build up from the source codes dependency information and the dependency information among class components. Here, the file to describe the source code dependency information is called ".make" file.

Details of the procedure are as follows. We make the assumption that all the nodes are homogeneous, and can share the same machine codes. This is reasonable because we discuss distributed cooperative development of one application.

i) Identify necessary classes

   The system identifies classes to be compiled by tracing dependency links in the class components. It starts tracing from the start class indicated in the application information when compiling the whole application, and from a class when compiling it only.

ii) Collect class source codes distributed on network

   Machine codes, headers and supplementary information of the identified classes are collected from node files. If a class is not compiled and does not have a machine code, its source code is fetched and compiled at the collector site.

The above procedure is repeated recursively until all the necessary classes are identified and collected.

3.3 Prototype Implementation Based on C++

Based on the above discussion, a prototype system based on C++ is constructed, which contains the following functions: class component information extraction from source codes, HTML file generation and registration, extraction of dependency information among class, and automatic Makefile generation. In this prototype, dealing with finished components is assumed, therefore, extraction is done at registration, and class components are generated and saved in the repository at the same time. In the following subsections, details of implementation are described.

3.3.1 Extracting Information

Extracting type/class information is further divided into two processes: partitioning information of a class into nodes and grasping relationship among nodes. C++ is not a pure object-oriented language, and contains various features not for object-oriented programming in order to keep compatibility with C. Here we deal with information concerning classes only, and ignore global functions or data outside of classes.

The following information is extracted for partitioning a class into nodes:

- Class (including struct and union) declarations,
- Other type declarations (typedef, enum),

- Declarations parameterized as templates.

Typedef and enum declarations are extracted to avoid name conflicts among types, due to the fact that a class is a kind of type in C++. In the current implementation, signature and name space declarations are ignored.

The topmost part of C++ grammar description in YACC to analyze and extract these information is as follows:

```
program:
    /* empty */
    extdefs

extdefs:
    extdefs extdef

extdef:
    fndef
    datadef
    template_def
    ......
```

This means that a program is a series of definitions, and a definition is either a function definition, a data definition, or a template definition.

A type declaration is in a typed declaration specifier which is a part of a data definition, and a class declaration is in a struct declaration in a type declaration. The following information of a class is extracted out of these:

- Name
- Interface information

  Member functions, member variables and access specifiers – private, protected and public
- Dependencies

  Inheritance, whole-parts and reference relationships

A grammar description concerning this is as follows:

```
structspec:
    ......
    class_head { opt.component_decl_list }
    ......

class_head:
    ......
    aggr identifier
    aggr identifier : base_class_list
    ......
```

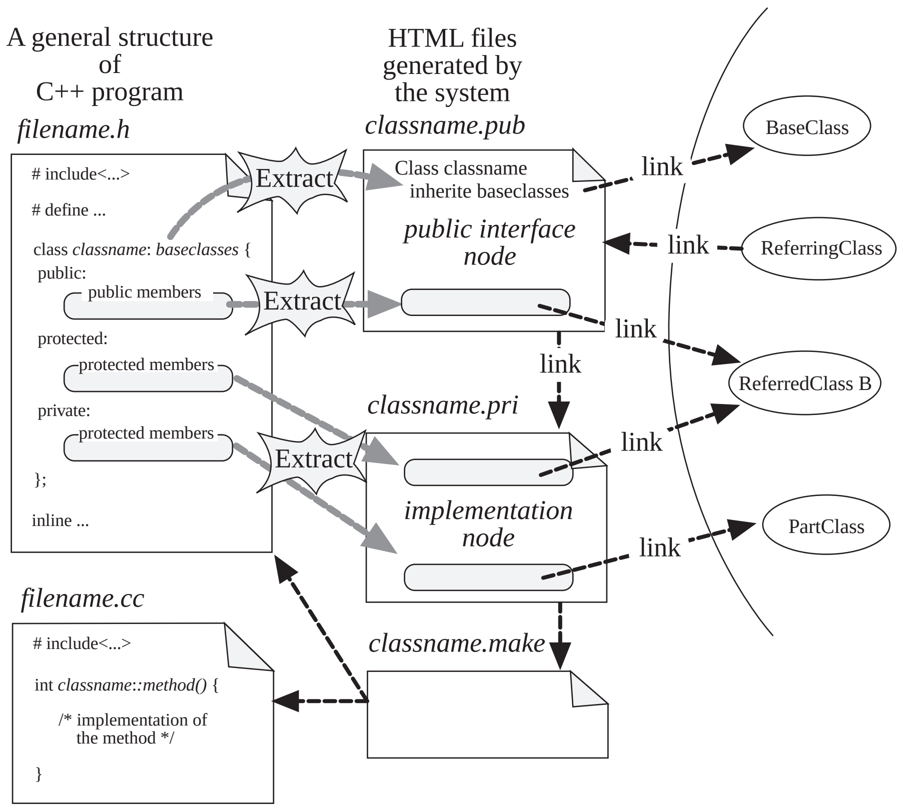


Figure 3.3: Correspondence between a C++ source code and nodes

```
aggr:
    class
    struct
    union

base_class_list:
    base_class
    base_class_list , base_class

base_class:
    typename
    access_list typename

access_list:
    ......
    visspec another_specs
    ......

opt.component_decl_list:
```

```
                /* empty */
                component_decl_list
                opt.component_decl_list visspec :
                          component_decl_list
                opt.component_decl_list visspec :

            visspec:
                private
                protected
         public
```

A class name (`identifier`) and its super class name (`base_class_list`), as well as super class access authorities are in the first part, `class_head` of the class declaration. Interface declarations, functions and variables, are in `component_decl_list`, and references to other types are in function and variable declarations. Variables can be either of an aggregate relation or a reference relation, however, they are not distinguished.

In this prototype, the format to save class components is HTML files. Source code components are given source codes. Therefore, the output transformation unit merely returns two HTML files corresponding to the requested component ID, a public interface node file and an implementation node file. The HTML files are created based on the extracted information. Only public super classes and members are stored in the public interface node file, and protected and private super classes and members are stored in the implementation node file. The public interface node file is an entrance of the class. Correspondence between a source code and nodes reflecting the code is shown in Figure 3.3. After the HTML files are generated, these are saved in the repository and the class ID and the location of them are registered in cmpd.

Figure 3.5 and 3.6 show examples of views for the HTML outputs generated from the source codes shown in Figure 3.4. The former is an example of a public interface node, and the latter is an example of an implementation node. They are linked to each other. Moreover, the links to other class components are placed on the name of the components which are types of class variables. In the figures, links are indicated by underlines.

3.3.2 Distributed Automatic Construction of Makefile

Suppose that ClassA depends on ClassB and ClassC, and ClassB depends on ClassD. The declaration part of ClassA is in the file ClassA.h, and the implementation part is in ClassA.cc and etc.cc. For example, the source code information is described as follows (in C++ case).

Declaration	ClassA.h
Definition	ClassA.cc etc.cc
OtherMachineCode	libxxx.a

```
/* This is a simple example for my paper.
   There is no entity corresponding to these classes. */
#include <String.h>

class ExampleColoredPoint: public ExamplePoint {
public:
    enum color_code { red, blue, yellow };
    static String color_names[];
private:
    color_code color;
public:
    void set_color(color_code vc) { color = vc; }
    String color_name() { return color_names[color]; }
};

String ExampleColoredPoint::color_names[] = {
    "RED", "BLUE", "YELLOW"
};
```

Figure 3.4: Example source codes

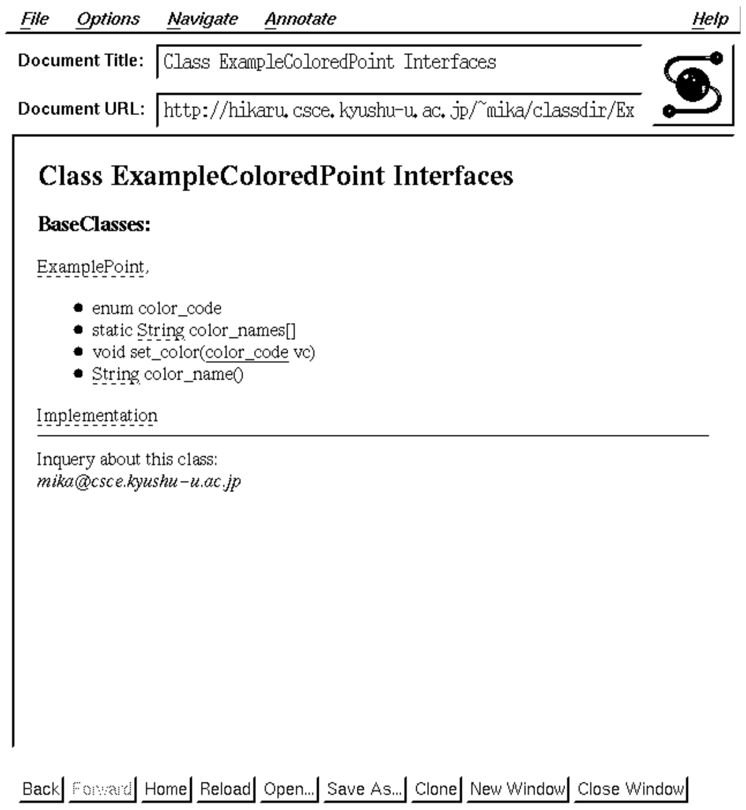
This information is converted as follows:

```
ClassA.o: ClassA.cc ClassA.declarations
etc.o: etc.cc ClassA.declarations
ClassA.declarations: declaration(in ClassA.make)  ClassB.declarations
                     ClassC.declarations
ClassB.declarations: declaration(in ClassB.make) ClassD.declaration
ClassC.declarations: declaration(in ClassC.make)
```

Furthermore, in the case where the starting class is ClassA and the application name is test, the following **Makefile** and **main.cc** are created automatically from these.

Makefile

```
all: test
test: main.o ClassA.o etc.o ClassB.o ClassD.o ClassC.o
main.o: main.cc ClassA.h
    $(CXX) -c ClassA.cc

ClassA.o: ClassA.cc ClassA.h ClassB.h ClassD.h ClassC.h
    $(CXX) -c ClassA.cc
etc.o: etc.cc $(HEADERS)
    $(CXX) -c etc.cc
ClassB.o: ClassB.cc ClassD.h
    $(CXX) -c ClassB.cc
ClassC.o: ClassC.cc
    $(CXX) -c ClassC.cc
```



Class ExampleColoredPoint Interfaces

http://hikaru.csce.kyushu-u.ac.jp/~mika/classdir/Ex

Class ExampleColoredPoint Interfaces

BaseClasses:

ExamplePoint,

- enum color_code
- static String color_names[]
- void set_color(color_code vc)
- String color_name()

Implementation

Inquery about this class:
mika@csce.kyushu-u.ac.jp

Figure 3.5: Public interface node

main.cc

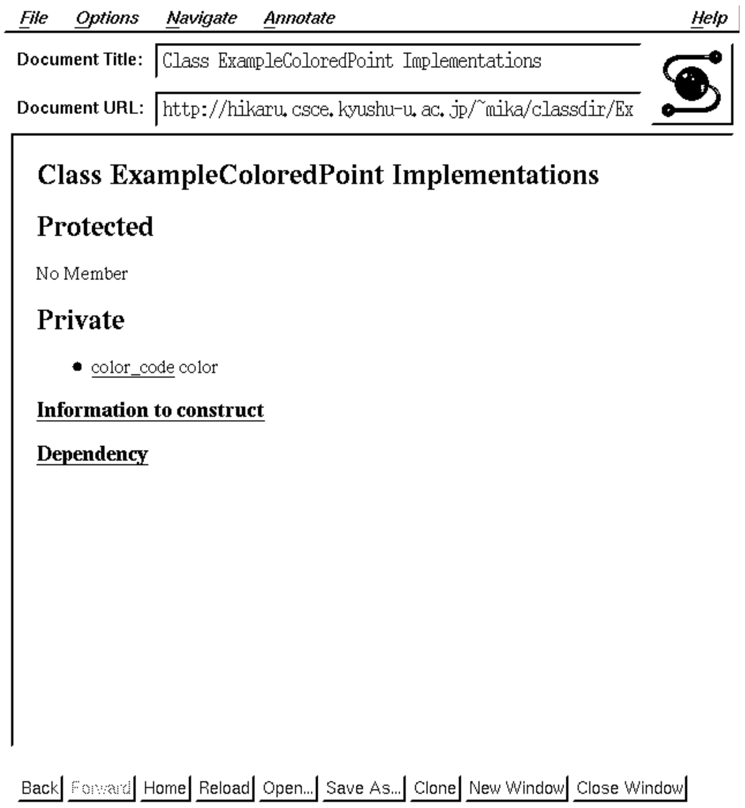
```
/***
 * This file is auto generated.
 */
#include "ClassA.h"

int main(char **args) {
    ClassA.main(args);
}
```

Since this system is still under construction, it must be examined in detail.

3.4 Related Works

C++ Source Code Repositories There are many source code repositories for C++ [63, 54], including commercial ones. Most of them support to debug and trace programs. However, most of them do not focus on aspect as components and not support to reuse.

Object Make In Object Make [53], makefiles are described in the unit of a source code file (class) as in our system. Our system provides a pre-processor to construct a makefile for


File Options Navigate Annotate Help
Back Forward Home Reload Open... Save As... Clone New Window Close Window

Document Title: Class ExampleColoredPoint Implementations

Document URL: http://hikaru.csce.kyushu-u.ac.jp/~mika/classdir/Ex

Class ExampleColoredPoint Implementations

Protected

No Member

Private

- color_code color

Information to construct

Dependency

Figure 3.6: Private implementation node

the whole application, however, Object Make provides a tool called “omake” to build the whole application by constructing components tracing dependencies.

3.5 Concluding Remarks

To provide a mechanism to deal with class components, which are the most primary components among many kinds of object-oriented software components, we discussed the design and the implemented prototype for a class component generation mechanism and an automatic Makefile generation mechanism to use the class components for C++ source codes. By cooperating the distributed component management system proposed in the previous chapter, the connections among several class-related components (source codes, construction information and application components) are provided, and it provides the basis for connectivity to other object-oriented software components.

There are some difficulties in implementing the prototype based on C++. The machine codes of C++ language are strictly dependent on machines. There are many variants of compiler and language specification for C++, therefore, the flexible circulation in distributed environments makes the component management difficult. To reuse components of C++, development environments and other conditions must be arranged. For extraction, it is difficult to deal with elements except for object-oriented ones. Therefore, we are considering to adopt Java language.

For automatic makefile generation mechanism, it is necessary to discuss how to describe dependencies in detail. Furthermore, it is necessary to discuss how to describe application and library components.

Furthermore, for effective program understanding support of class components, a mechanism for reversed reference to specifications and other components is necessary. A proposal is provided for automatic reversed reference generation by generating application/class components from design patterns. This proposal will be discussed in more detail in Chapter 5.

Chapter 4

Design Pattern Components

"Design patterns" are collecting attentions in the area of object-oriented analysis and design. A design pattern is a description of a typical problem and a solution for it which frequently appears in applications but is independent from problems which the applications target. The benefit of design patterns is to share solutions over areas, to reuse the solutions as well as to enable developers to communicate more easily using the patterns.

Design patterns are design know-hows to modularize applications mainly, therefore, these are not necessary to solve problems the applications target. In other words, without design patterns you can make applications only to solve current problems. If a part of an application or a library implements a design pattern, the meaning of the structure is difficult for others to understand. So, the prepared structure in the application or the library may not be used, or in the worst case, even destroyed. To prevent the above problem, collaboration between design patterns and applications and libraries in which these are used is necessary.

Currently, however, design patterns are treated as documents and there is no uniform framework to manage the patterns in a catalog on computers. We propose building a catalog of design patterns upon the results of our preceding research [42] as described in the previous chapters. Current design patterns are described using plain texts, figures and pseudo codes. For example, patterns in a book such as "Design Patterns [21]," are not able to be managed in computers. Therefore, we aim to build a framework for dealing with plain texts, figures and pseudo codes uniformly based on SGML (Standard Generalized Markup Language [67]). SGML is a generic term for different languages to describe structure of different kinds of document using marks called "tag". Furthermore, we address to code generation support from design patterns to relate them with source codes.

By expressing patterns in SGML, it is possible to deal with plain texts, figures and pseudo codes uniformly, and it is easy to manage consistency of meanings and to relate them with source codes. On the other hand, because of the integration of plain texts, figures and pseudo codes in a unit, it has to be separated into texts, figures and pseudo codes. In addition, because figures are converted into structured documents to relate patterns with source codes, the figures has to be generated from the structured documents. At code generation, it is necessary to supplement information by an user.

In the following sections, design patterns are explained briefly. Next design patterns using SGML are proposed. Furthermore, the conversion mechanism from the description into hypertext nodes and supporting system for source code generation using the description and implementation of a prototype using Java are explained. Then related works are described in the last sections.

4.1 Design Patterns

A design pattern is a description of a typical problem and a solution for it which frequently appears in applications but this pattern is independent from problems which the applications target. The term "design pattern" originates in the field of Architecture, and used to name structures used in architectures frequently. Gamma applied the term to structures in object-oriented applications in his doctoral thesis, and the way to describe is based on one in the field of Architecture.

In the area called Software Architecture in the field of Software Engineering, there are a lot of researches for solutions and algorithms used in software. In object-oriented programming, a program is separated into components per object, the structure of the program is more similar to an architecture. Most of structures proposed as design patterns use features of object-oriented technology, in particular inheritance and polymorphism. W. Pree [43] states that design patterns support to set up hot spots, which are several components of an application to exchange at extension in the future, when designing "frameworks" in object-oriented design.

Recently many researchers have proposed and discussed design patterns. [21] is one of the most famous catalogues of design patterns, which contains 23 patterns which are found mainly in GUI area. There are other patterns in P. Coad's paper [17] and several papers in [16, 62]. When designing the format of SGML for design patterns, we based on [21]. Some of patterns in the book are used as test cases.

A design pattern includes at least four essential elements; a *name*, a *context*, a *solution* and a *consequence*. A *name* expresses characteristics of the pattern and may contain other names as aliases. A *context* includes problems described as concrete examples or in abstract expression and sometimes includes several selections and costs for them. A *solution* includes the reason to apply it and practical design forms and rules. A *consequence* is a description of what happens by applying the pattern. In addition to the four components, a design pattern contains a combination with other patterns and relations such as which pattern to be more specified.

In [21], a design pattern contains the following items,

Name	:	"name", "also known as"
Context	:	"intent", "motivation", "applicability"
Solution	:	"structure", "participants", "collaborations"
Consequence	:	"consequences"
Others	:	"known uses", "related pattern"

Most of these items are explanation described in plain texts, and sometimes figures are inserted using OMT description to supplement the explanation. For *solutions*, charts, which may include message passing, must be supplemented. In the next section, we will discuss an SGML framework to deal with this kind of documents.

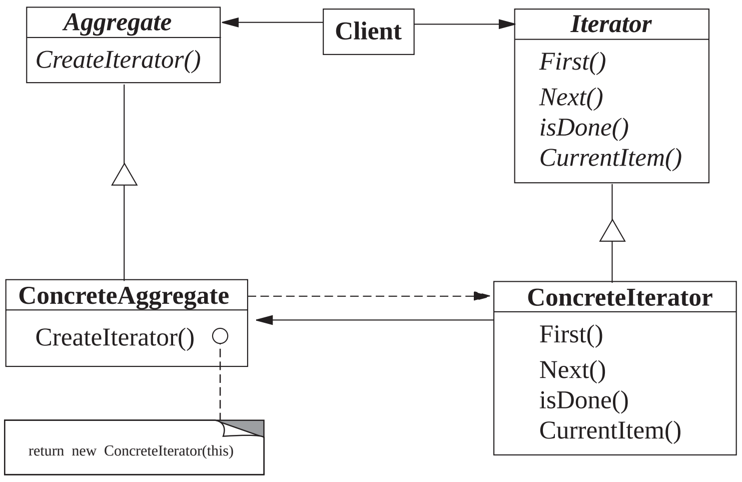


Figure 4.1: Example of design pattern

Example of Design Pattern: "Iterator" As an example of design patterns, Iterator Pattern is explained briefly.

The Iterator Pattern is one of the closest design patterns to concrete data models. It is a pattern which enables the exchange of both types of aggregate and methods for iteration, by separating structure to iterate from an aggregate and making interface common, when a certain set to be iterated is given. The Iterator Pattern, therefore, consists of four classes (roles): an abstract and a concrete classes for aggregate, and an abstract and a concrete classes for iterating the aggregate classes. The concrete classes corresponds to each other. A method named *CreateIterator*, which is a design pattern called "Factory Method [21]," makes the concrete classes correspond to each other and conceals implementation (concrete iterator name) from outside. Iterator has four methods to represent iteration functions generally, however, in several implementations (for example, java.util.Enumeration in Java library), they may be packed into two or three methods.

4.2 SGML Framework for Design Patterns

To make a catalogue of design patterns on computers, we separate a design pattern description into three parts, *explanation* such as context and result in plain text, *structure information* expressed as a class diagram and *pseudo codes* included in the diagram. Explanation is dealt with as structured texts separated into items. Structure information is described also as structured texts. On the other hand, pseudo codes are described using a simple language. The reason for dealing with structure information and pseudo codes not as a chart but as structured texts is to use the information for interactive code generation.

Here pseudo codes are a description for supplementing methods in OMT charts in [21], which are expressed like codes in a simple programming language. We design a simple internal expression to generate code fragments in a concrete programming language.

To integrate these elements uniformly, SGML (Standard Generalized Markup Language) is adopted. SGML is a generic term for different languages to describe the structure of

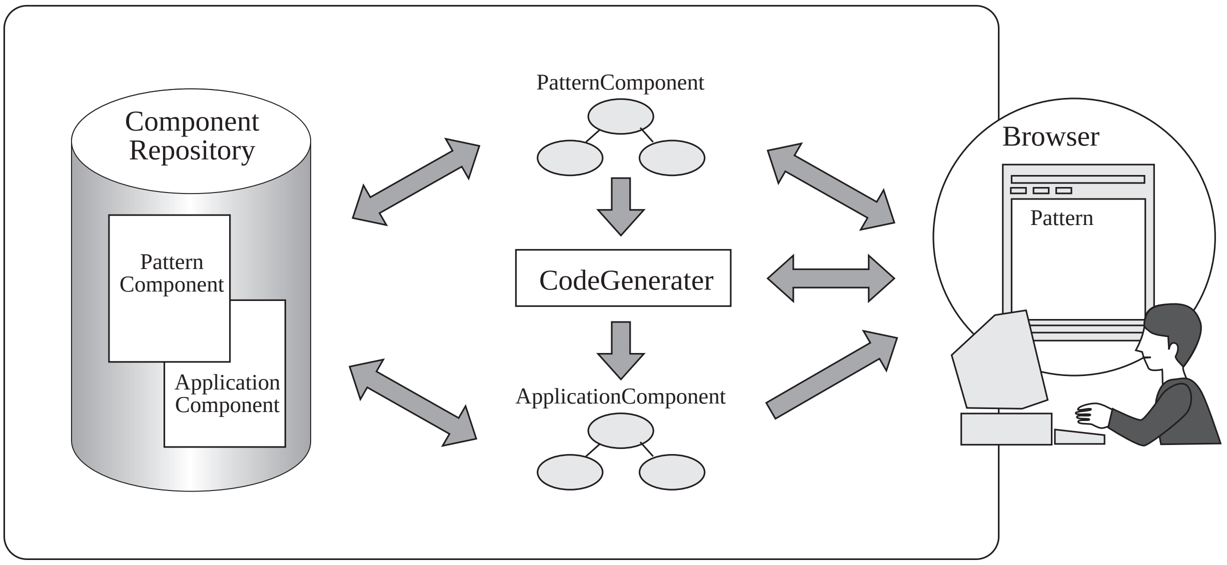


Figure 4.2: System overview for design pattern components

documents using marks called “tags”. Originally, SGML was proposed for sharing documents such as manuals [67]. The documents had similar structures but were described in different formats. When sharing them among companies, the standard format was required. SGML is a framework to define formats for the documents. One of the most representative databases using SGML is a manual database of the Canadian Department of National Defense in USA. HTML (Hypertext Markup Language) used in WWW was the most popular instance of SGML, but recently HTML focuses more on layout information rather than structure information. So HTML is getting apart from SGML.

The “tag” is expressed as a string enclosed by “`<`” and “`>`”. When a document has certain structure such as an recipient address of a letter, it is expressed by enclosing the address string by a start tag (`<to>`) and an end tag (`</to>`) as “`<to>` Name `</to>`”. Texts enclosed by tags can contain texts enclosed by tags recursively, so a hierarchy structure can be expressed. Tags can have attributes such as a name. DTD (Document Type Description) describes rules to construct a certain structured document by defining tags [67].

The language that we designed to describe design patterns is named PIML (Pattern Information Markup Language).

There are two required processes for PIML documents as the following (and as shown in Figure 4.2),

- converted into hypertext nodes
- related with source codes generated from them

This results in the following requirements for the PIML structure.

- explanation texts

  should be separated from the whole document and displayed as hypertext nodes

- structure information

  should be expressed in a figure as a hypertext node

  needs to be structured for source code generation

- pseudo code

  is not converted to a hypertext node (inserted in an OMT chart as is)

  is analyzed at source code generation

In the following sections, whole structure of PIML will be described, as well as structure information and pseudo codes. DTD (Document Type Definition) of PIML and an example of PIML description of *Iterator Pattern* are attached at the end of thesis: Appendix A and C.

4.2.1 Basic Framework

The items in [21] are used almost as is to determine the tags of PIML. The context items, "intent", "motivation" and "applicability", and a consequence item "consequences" are converted into the tags without any change. Three of the solution items, "structure", "participants" and "collaborations", are re-constructed into structure information separated into classes. The others of solution items, "implementation" and "sample code", are converted into tags without any change as the context and consequence items. In the future, the "implementation" item will be integrated into structure information to select trade-offs.

In this paper, items used in [21] are used without change, but there are some other formats. Essentially they are also based on definition of design patterns proposed by Alexander, therefore PIML documents can be converted into the format. For example, "intent" can be omitted because it is merely a resume of the context items. "Motivation" and "applicability" may be merged into one item.

The example of "Iterator Pattern" described in PIML is shown in Figure 4.3. As can be seen in the example, `<pattern>` has a name of a corresponding design pattern as an attribute, in this case "Iterator", and may have the following ten tags:

- `<also_known_as>`
- `<intent>`
- `<motivation>`
- `<applicability>`
- `<consequences>`
- `<implementation>`
- `<samplecode>`
- `<known_uses>`
- `<pattern_relations>`

- `<structure>`

Eight tags of them, `<also_known_as>`, `<intent>`, `<motivation>`, `<applicability>`, `<consequences>`, `<implementation>`, `<samplecode>` and `<known_uses>`, correspond to eight items in [21]: "also known as", "intent", "motivation", "applicability", " consequences", "implementation", "sample code" and "known uses". These tags include texts such as:

```
<intent>
   Provide a way to access the elements of an aggregate object sequentially
   without exposing its underlying representation.
</intent>
```

On the other hand, the `<structure>` tag integrates information of three items, "structure", "participants" and "collaboration".

The exact grammar of PIML is described in DTD (Appendix A). The principle behind the design of PIML is that elements including long texts are expressed as tags and elements such as an identifier and a boolean value are expressed as attributes.

4.2.2 Structure Representation

Structure information, consisting of classes and relations among classes, should be described in a structured document not in a figure, so that can be used for code generation mainly. Therefore the items in [21], "structure", "participants" and "collaborations", are re-constructed into structure information separated into classes. In the concrete, structure information contains explanation included in the "collaboration" item, classes and relations among classes. A class contains a name, explanation for the class included in the "participants" item and interfaces. An interface contains a name, an explanation about the interface and pseudo codes included in a figure of the "structure" item. A relation between classes contains one kind of inheritance, aggregate, reference or creation. It also contains names of target and origin classes. A label is needed if it is aggregate or reference.

Furthermore, layout information to generate an OMT chart and cloneable information to support source code generation are added (Table 4.1 indicates outline and tags of the structure part of PIML). Details of layout information and cloneable information are described in Section 4.3 and Chapter 5 respectively.

We call classes constructing a pattern "roles" in order to distinguish classes of a pattern which express roles in the pattern from the actual classes constructing an application in our system (Figure 4.4). In the rest of this paper, we call classes of a pattern "roles" and classes of an application "classes".

Figure 4.5 is a brief description of structure information of the "Iterator Pattern" in PIML.

4.2.3 Pseudo-Code Representation

All characteristics of a pattern can not be described only by the structure of roles. Behaviors of all roles determine the behavior of the whole pattern. In [27], the functions which must be executed in a method of a role are described as "contracts" among roles using a formal

```
<pattern name="Iterator">
 <intent>
  Provide a way to access ...
 </intent>

 <motivation>
  An aggregate object such as a list ...
 </motivation>

 <applicability>
  Use the Iterator pattern ...
 </applicability>

 <consequences>
  The Iterator pattern has ...
 </consequences>

 <implementation>
  Iterator has many implementation ...
 </implementation>

 <samplecode>
  We'll look at the implementation ...
 </samplecode>

 <known_uses>
  Iterators are common in ...
 </known_uses>

 <related_patterns>
  ##p:Composite##: Iterator are ...
 </related_patterns>

 <structure>
  ...
 </structure>
</pattern>
```

Figure 4.3: PIML example

expression. On the other hand, in [21] it is described in a simple programming language in a box connected to a method name to help intuitive understanding. We intend to generate code fragments in a practical language reflecting behaviors of roles. Therefore, we design a simple language which satisfies requirements to generate code fragments.

Table 4.1: Structure part of PIML syntax

Structure	notes: Notes;
	relations: Relations;
	roles: Roles;
	cloneables: Cloneables;
	layout: LayoutInfo
		Notes is a comment. Relations and roles are necessary structure elements. Layout is necessary to generate OMT chart. Cloneable is necessary for source code generation support.

Roles	Role*	Set of roles.

Role	syslabel: Identifier;
	abstract: Boolean;
	notes: Notes;
	operations: Operations;
		'Syslabel' is an identifier. 'Abstract' is a flag attribute to indicate if the role is `abstract` or `concrete`. 'Notes' and 'operations' are child elements.

Relations	Relation*	Set of relations.
Relation	Inheritance | Aggregate | Reference | Creation	Relation is one of inheritance, aggregate, reference or creation. All relation have two IDs to indicate a target class and a origin class.
Inheritance	target, origin: Identifier	Inheritance dose not have an identifier.
Aggregate	target, origin, syslabel: Identifier; number: Number	Aggregate has an identifier and the 'number' attribute to indicate if the target is a collection of the class (`many`) or the class itself (`single`).
Reference	target, origin, syslabel: Identifier; number: Boolean	Reference has an identifier and the 'number' attribute to indicate if the target is a collection of the class (`many`) or the class itself (`single`).
Creation	target, origin: Identifier	Creation dose not have an identifier.
Operations	Operation*	Set of operations.

Operation	syslabel, return_value: Identifier;
	abstract: Boolean;
	access: AccessSpec;
	notes: Notes;
	code: PseudoCode;
		Operation has four attributes and two elements. The 'return_value' attribute is a return value of the operation. The abstract attribute indicates if the operation is abstract or concrete. The 'access' attribute can contains an access level, `public`, `protected`, `private`, `privateprotected`.

Here, the meanings of marks in the second rows are as follows: "*" is a set, "|" is a selection and ";" is a connection. In addition, Syntax attributes are described in the format as "*attribute name*: *syntax element name*".

In other words, the language can express minimal control constructions, method calls and

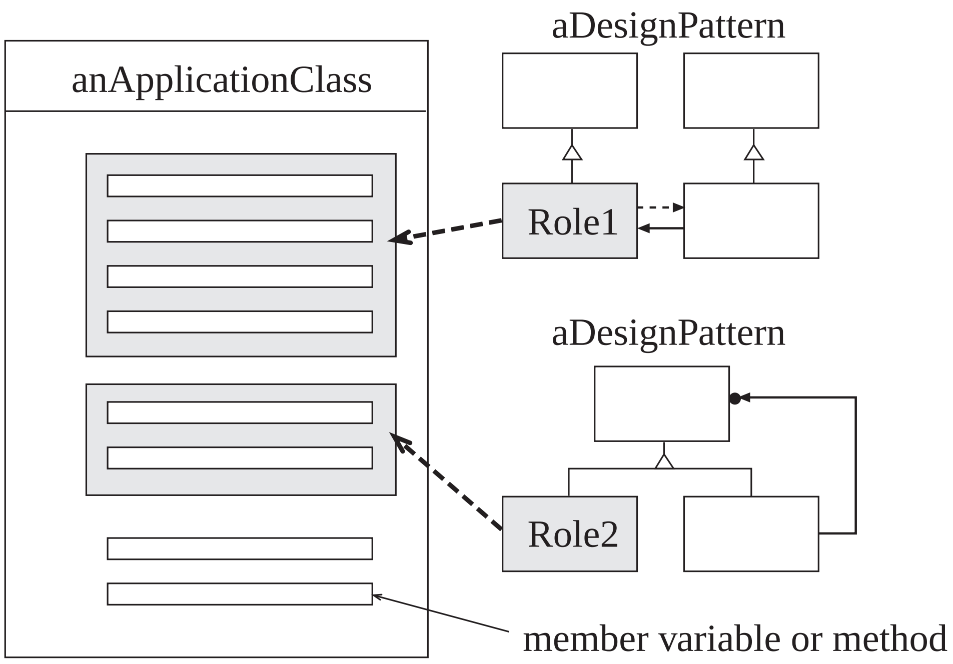


Figure 4.4: Correspondence between a class and roles

assignments, and does not include complex statements such as nested classes and expression statements for concrete operations. The language having such characteristics are designed as a language like Java. An iteration statement is used only for broadcasting to aggregate of objects, so there is only "forall". Pseudo codes described in the simple language are analyzed and converted into real code fragments by replacing names and so on. In our current research Java is adopted as the target language, but languages are not limited if grammars of them satisfy the above characteristics.

The following is an example of pseudo code for *CreateIterator* operation in *ConcreteAggregate* role of Iterator Pattern.

Example: ConcreteAggregate::CreateIterator() of Iterator Pattern

```
<pseudocode>
  return new "ConcreteIterator" ( "this" )
</pseudocode>
```

Also the whole syntax of the pseudo code is attached in the last of this thesis as Appendix B.

4.3 Hypertext Node Generation for Browsing

A PIML description is separated and converted into hypertext nodes as follows:

- Contents table

```
<structure>
 <notes>
  Collaboration description
 </notes>
 <cloneables>
  <cloneable>
   <celem type=role id=...>
   ...
  </cloneable>
 </cloneables>
 <relations>
  <inheritance from="ConcreteIterator"
   to="Iterator">
  ...
 </relations>
 <roles>
  <role syslabel="Iterator">
   <notes>
    Explanation for the Iterator class.
    (corresponding to a part of
     a "Participant" item)
   </notes>
   <operations>
    <operation syslabel="First" ..>
   </operations>
  </role>

  <role syslabel="ConcreteIterator">
   ...
  </role>
 </roles>
</structure>
```

Figure 4.5: Structure part of PIML example

- Explanation items
- Class diagram (OMT chart)

The table of contents is generated dynamically by extracting all item tags included in a PIML documents, which consists structure of items of a pattern.

Explanation items except `<structure>` are converted into HTML outputs by enclosing extracted texts with HTML tags. Cooperation with other software components must be represented as hypertext links. The first problem is to represent links to other patterns and other explanation items. Also, links to source code components are important.

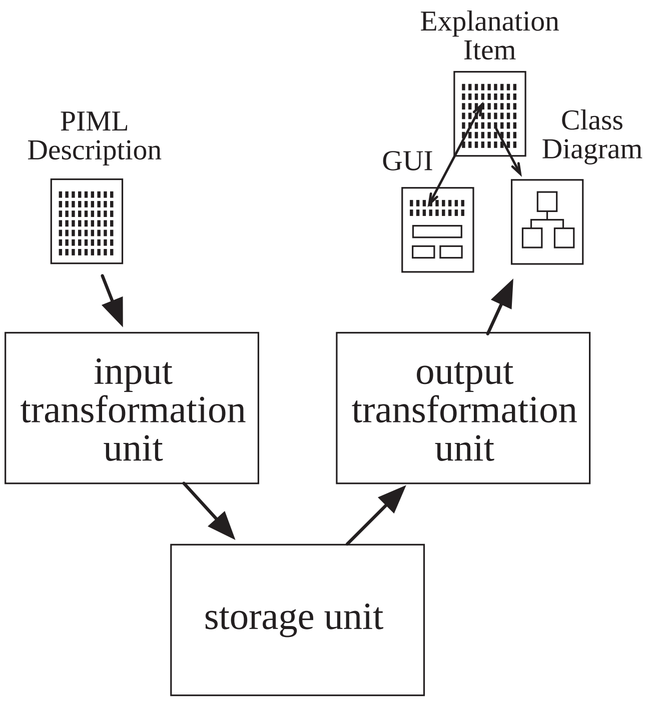


Figure 4.6: Process model for design pattern components

The contents page and explanation item pages are generated dynamically, therefore, it is difficult to denote absolute URLs of pattern components into them directly, as in the class component case. Furthermore, it is difficult to insert CGI procedures generating the pages in PIML documents, because URLs of CGI procedures are variable. Consequently, we devised "abstract anchor" mechanism.

When extracting each item from a PIML document and generating HTML outputs, identifiers put in the document as anchors are converted into CGI procedures with the identifier as an argument. By invoking the CGI procedures, a new HTML converting process for generating a contents page corresponding to the given identifier is executed, By this mechanism, when selecting an identifier in a current document, a component corresponding to the identifier is displayed.

To perform this function, identifiers marked by special marks (distinguishable from tags) must be inserted in the document beforehand. As the mark, currently `##` is used.

For example,

Description of Iterator Pattern

Related Patterns Composite: Iterators are often applied to ...

PIML description for the above:

```
<related_patterns>
##p:Composite##: Iterators are often applied to ...
```

Generated HTML Page:

```
<head> Related Patterns </head>
<body> ...
<a href="http://webserver/cgi-bin/getcompo?pattern&Composite&index">
 Composite </a>: Iterators are often applied to ...
```

The format of class diagrams used in this paper follows [21] which is a simplified version of class diagram in OMT (Object Modeling Technique). OMT is a modeling technique proposed by J. Rumbaugh et al [46]. The class diagram in [21] consists of boxes expressing roles and arrows expressing relations among roles. A box of a role contains the name of the role and methods. A method may have pseudo codes in a box beside a role box. In this case, a line is drawn between the method string and the pseudo code box. A relation must be either inheritance, aggregate, reference or creation. A kind and multiplexity of a relation arrow can be distinguished by the shape of terminal points of the arrow. Aggregate and reference relations may have their names; in this case, the name is attached beside the line of the arrow.

The reason to generate class diagrams from structure information but not to prepare as a figure drawn by hand is to reflect structure information described in PIML. When drawing a diagram practically, layout information is necessary in addition to structure information.

In automatic chart generation, generally, one of the biggest research issues is automatic arrangement of constituent elements. However, because of the limited number of elements and low update frequency, it is adopted to describe layout information in a design pattern description and use it for layout. Therefore, to describe layout information, the new tag `<layout>` is introduced as an element of `<structure>` tag. There would be an alternative to attach layout information into `<role>` tag, but it is not adopted because layout information should be independent from semantic information.

Concretely, `<layout>` tag is a tag to express a two dimension array. It has a number of rows and columns as attributes, and boxes (`<box>`) as elements. A `<box>` tag contains an identifier of the target role and an index of the two dimension array to be located as attributes. In other words, a descriptor must decide where to locate role boxes in a two dimension array beforehand.

For example, the PIML description of “Iterator Pattern” has the following layout information:

```
<layout rows="2" columns="3">
  <box name="Aggregate"
       row="1" column="1">
  <box name="ConcreteAggregate"
       row="2" column="1">
  <box name="Iterator"
       row="1" column="3">
  <box name="ConcreteIterator"
       row="2" column="3">
  <box name="Client"
       row="1" column="2">
</layout>
```

The flow of automatic generation of OMT chart is as follows:

Analyze layout information (done at analysis of PIML)
↓
Generate boxes and linkage them to roles in correct position.
↓
Calculate size of boxes by using font size and the number of characters.
↓
Calculate size of the layout field by using the max hight of boxes in a row and the max width of boxes in a column.
↓
Draw boxes on the field by using the above location data
↓
Draw lines between boxes corresponding to relationships

4.4 Prototype Implementation

Figure 4.7 is a class structure of a mechanism to convert pattern components stored in PIML format into HTML. The location information can be obtained from a server managing an identifier and location table for software components.

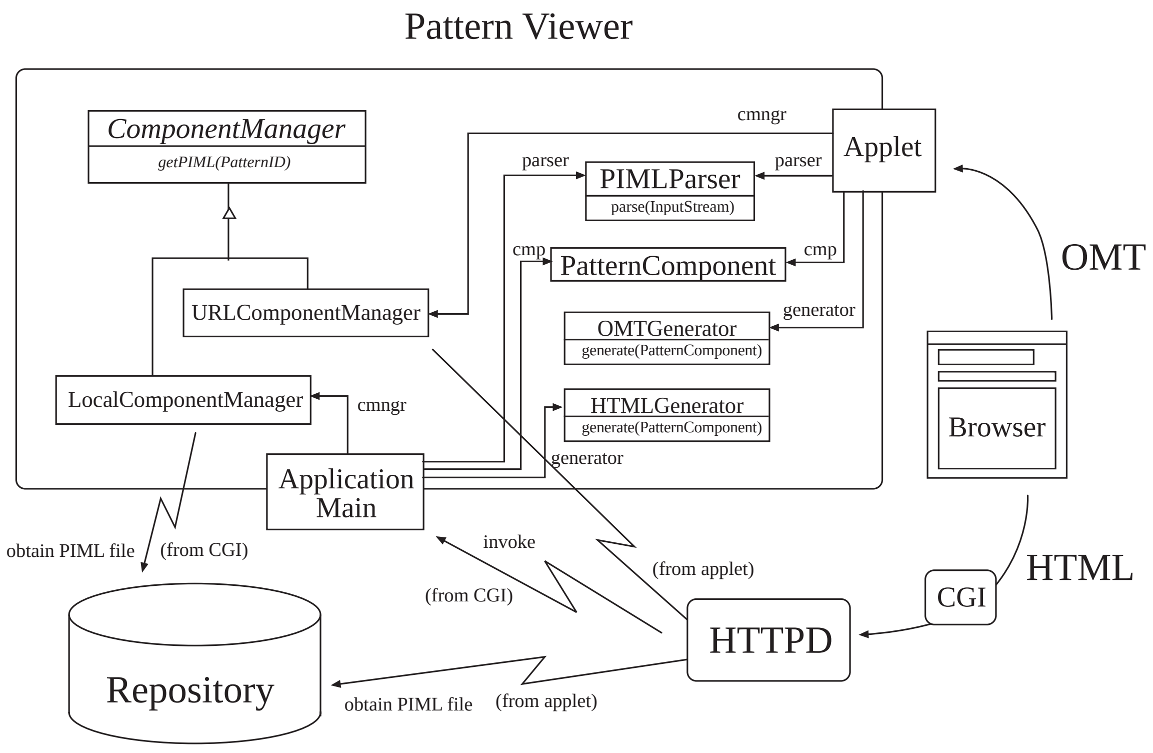
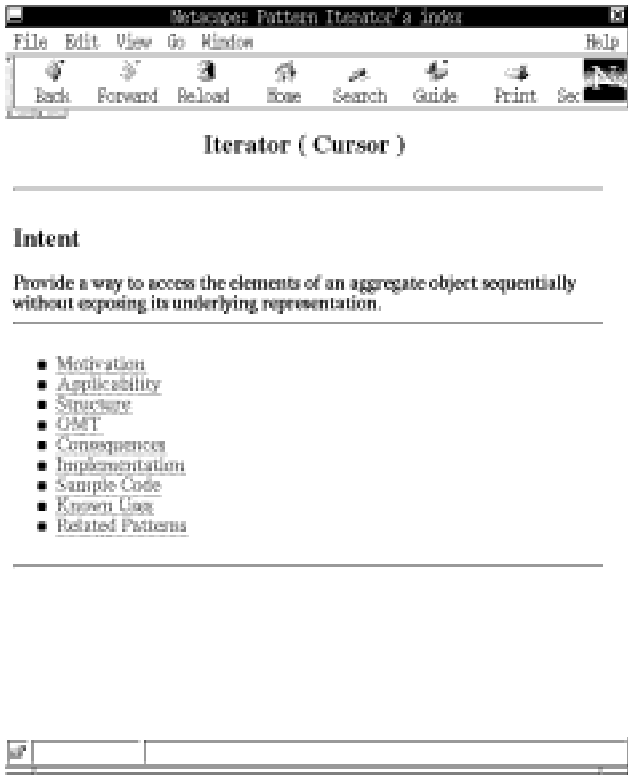


Figure 4.7: Overview of converting system (for view)

`Pattern` in the figure is the object converted from PIML file obtained using location information as abstract syntax tree (a variant of "Composite Pattern"). `Pattern2PIML` in the figure convert a `Pattern` object into HTML by scanning the `Pattern` tree, and corresponds to `ConcreteVisitor` in "Visitor Pattern". In interactive code generation, converters corresponding to `ConcreteVisitor` are used.


Netscape: Pattern Iterator's index
File Edit View Go Window Help
Back Forward Reload Home Search Guide Print Sec

Iterator (Cursor)

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- Motivation
- Applicability
- Structure
- OMT
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

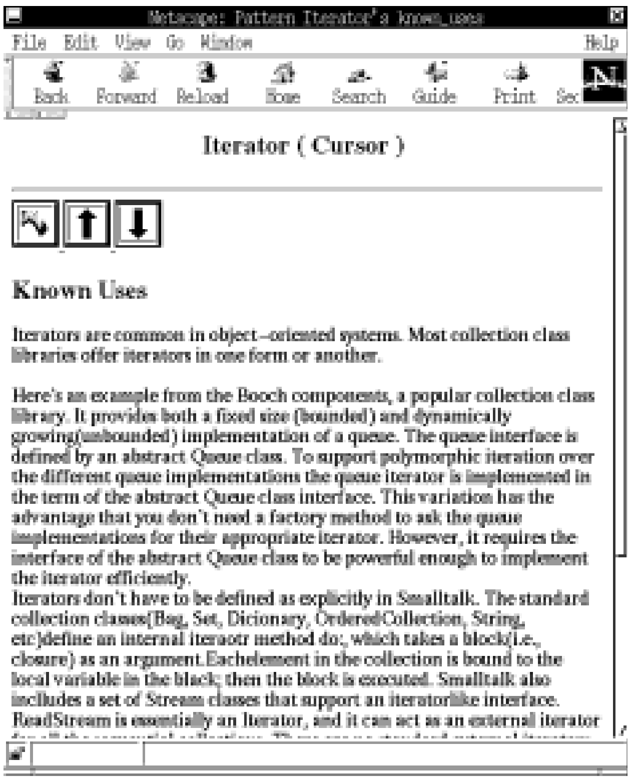
Figure 4.8: Index page

Figure 4.8, 4.9 and 4.10 indicate respectively the content table, the item "Known Uses" in explanation items and the OMT chart expressing structure information generated from the PIML description of *Iterator Pattern.*

4.5 Related Works

There are two approaches for cataloging design patterns on computers: one is to focus on a document aspect of a design pattern, and the other is to focus on a solution aspect. The latter means formalization of design patterns not for developers to apply by hand but for computers to support it. In this approach, *context* and *consequence* descriptions of patterns are still in the form of texts, and its document aspect is preserved. In other words, what to formalize is only *solution* such as structures, pseudo codes and so on.

Approaches to managing patterns as documents is categorized according to management granularity. There are two extremes; one is to merge all patterns into one file, and the other would be to manage per role or method. The two approaches are managing per pattern


Netscape: Pattern Iterator's known_uses
File Edit View Go Window Help
Back Forward Reload Home Search Guide Print Sec

Iterator (Cursor)

Known Uses

Iterators are common in object –oriented systems. Most collection class libraries offer iterators in one form or another.

Here's an example from the Booch components, a popular collection class library. It provides both a fixed size (bounded) and dynamically growing(unbounded) implementation of a queue. The queue interface is defined by an abstract Queue class. To support polymorphic iteration over the different queue implementations the queue iterator is implemented in the term of the abstract Queue class interface. This variation has the advantage that you don`t need a factory method to ask the queue implementations for their appropriate iterator. However, it requires the interface of the abstract Queue class to be powerful enough to implement the iterator efficiently.
Iterators don`t have to be defined as explicitly in Smalltalk. The standard collection classes(Bag, Set, Dicionary, OrderedCollection, String, etc)define an internal iteraotr method do:, which takes a block(i.e., closure) as an argument.Eachelement in the collection is bound to the local variable in the black; then the block is executed. Smalltalk also includes a set of Stream classes that support an iteratorlike interface. ReadStream is essentially an Iterator, and it can act as an external iterator

Figure 4.9: Item page

and managing per item of a pattern. The former makes management easy, but requires dynamic decomposition. The latter does not require decomposition of patterns, but makes management cumbersome.

There are already several examples focusing on a document aspect of patterns. Portland Pattern Repository [64] and a pattern DB at MIT [65] are repository systems to share design patterns and other related documents on the Internet. [64] has only a guideline to describe a design pattern, to manage imperfect documents. Currently, it is used by researchers to communicate patterns with each other. [65] is a simple text database, which manages patterns in the unit of item and can be search. The OMT charts are placed in documents in the form of bitmap image.

On the other hand, a prototype system made in IBM T. J. Watson Research Center [12] is an example focusing on a solution aspect of patterns. Regarding the formalization of patterns, differences between [12] and ours are as follows:

i) [12] does not formalize structures, but only presents a bit-mapped pictures, while our system generates figures of structures dynamically from structure descriptions in PIML:

ii) On code generation, [12] uses macro expansion, while our system uses pseudo codes in a simple language, and has the ability to generate source codes in several languages.

    [12] focuses on supporting C++ code generation, but not on a repository to manage patterns and source codes. On the other hand, we aim at constructing a repository to

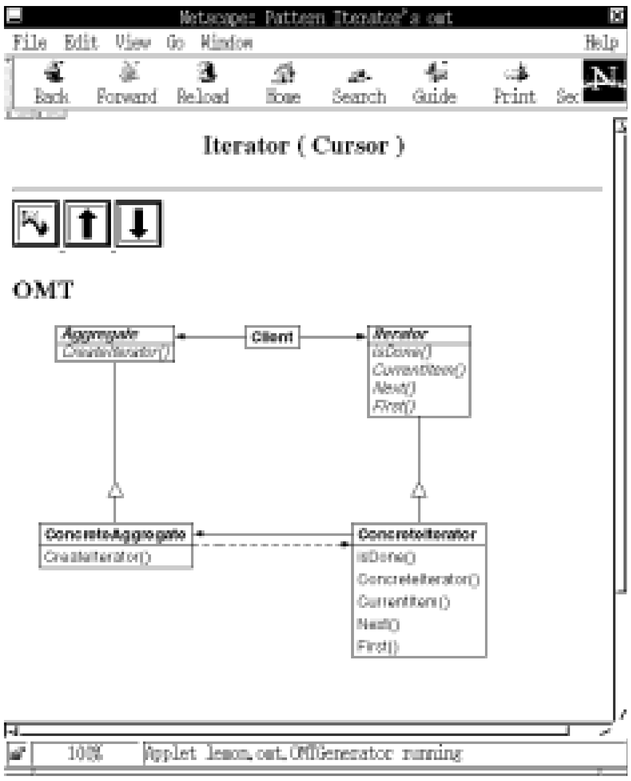


Figure 4.10: OMT chart

manage patterns and source codes integratedly.

Regarding structure formalization, there are related researches also in the areas of Software Architecture [69, 70] and Adaptive Programming [37]. In these areas, management of structure information, source code generation and visualization are important themes, which are closely related to our research. However, it is crucial that design patterns are more general and more abstract structure description which are not specific to certain applications.

4.6 Concluding Remarks

In this chapter we propose a framework for managing design patterns as SGML documents. A design pattern is divided into three elements; explanation texts, structure information and pseudo codes. Structure information is formatted in SGML. We also designed a simple language to describe the behaviors of methods, and are implementing a generator of code fragments in a concrete language. The current target concrete language is Java. In addition, we provided the function to convert the structured documents into HTML format. Converting is carried out per item.

By describing design patterns in the form of SGML documents, it becomes possible to deal with design patterns on computers. Our prototype provides examples of how to deal with design patterns on computers, browsing them and using them for source code generation support. Furthermore, there are other possibilities resulting from being SGML documents such as circulation flexibility and integration with SGML database systems.

In the future, we will study the following issues:

- PIML writing support

  Support to write PIML is necessary, because description, in particular one for structures, is complicated and difficult to keep integrity. Support to write PIML makes it easy to describe experimental design patterns, and will result in contributing further promotion of design pattern. A prototype for support mechanism is almost implemented.

- Support for structure variations

  The structure of a design pattern is in fact abstract, and has a lot of variations depending on its usage. In our current system, the structure is fixed, and it is difficult to tailor such variations. Therefore, it is necessary to investigate allowable variations of the structure and to extend our system to tailor them.

- Augmentation to an integrated CASE tool

  From the standpoint of CASE tools view, our system, as a repository in particular, needs a lot of functions such as version management, access control and so on. Because some of these functions are already realized in existing CASE tools, we plan to integrate such tools into our system. For example, to build a software automatically, the "make" command is useful in managing dependency information. For version management, we investigate to integrate existing version management applications such as RCS and CVS.

Details of source code generation support are discussed in the next chapter.

Chapter 5

Source Code Generation Support Using Design Patterns

There are two reasons for supporting source code generation from design patterns: one is to promote the use of design patterns, the other is to support the comprehension of source code components by referring to design pattern components.

When using design patterns in practice, there are a lot of constraints and alternatives. Users are often confused and have to repeat adapting them too often. Our goal is to promote use of design patterns by providing constraints and alternatives opportunely and adapting them in users' places.

On the other hand, structures based on design patterns are difficult to be understand from the standpoint of an application, because the structure is independent from problems which the application solves. This causes increasing maintenance costs and increases the possibility of unexpected destruction of such structure due to misunderstanding. To prevent such problems, we aim at providing comprehension support by integrating source code components and design patterns.

In general, there are two alternatives for integrating source code components and design pattern components: one is to generate source codes from design patterns, the other is to relate given source codes to design patterns. We focus on the former and the latter remains as a future issue.

In this chapter, source code generation supporting system is discussed.

5.1 Issues for Source Code Generation Support

The description syntax of information needed for code generation must be determined by considering issues as follows:

- Multiple and duplicated design patterns

  An application can contain implementations generated from several patterns (including ones generated from the same pattern)

- Duplicated roles

  Several classes can be generated from one role

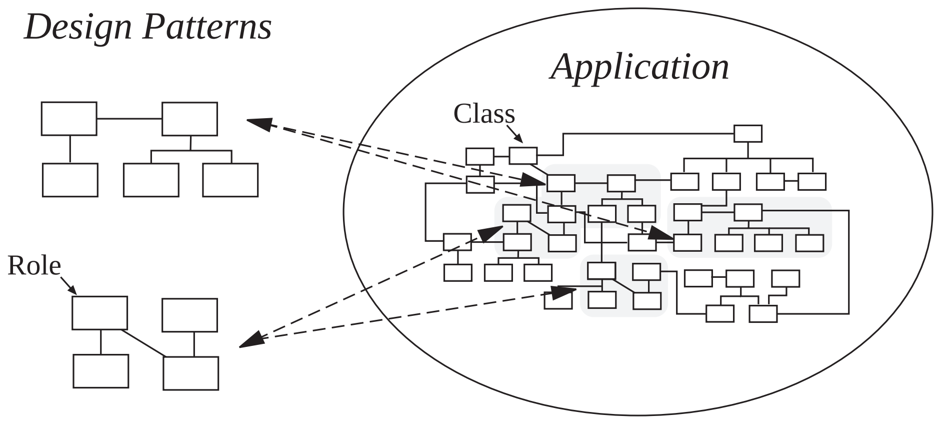


Figure 5.1: Generation of classes from design patterns

- Alternatives at implementation

  It is necessary to distinguish language-dependent ones and not.

- Variants

  There are several variants for a pattern, which structure is slightly different. Alternatives at implementation can be considered as a kind of variant.

- How to describe relationships between patterns and classes

  Relationships between patterns and classes should be maintained for comprehension support.

There are relationships between patterns and classes (applications) generated from them as indicated in Figure 5.1.

Intermediates are needed between design patterns and classes, because of the many to many relationships. In our system, a structure called "Instantiated Pattern Structure" is introduced as the intermediation. The instantiated pattern structure is called IPS in short. Here to "instantiate" means to make an instance from a design pattern. An IPS expresses not only an instantiated design pattern, but also a part of a class which behaves a certain role in the pattern. In other words, an IPS keeps one of many to many relationships between patterns and classes(Figure 5.2). An IPS must be distinguished by an identifier, when the same pattern is instantiated to another set of classes.

5.2 Code Generation Process

In this section, the process to generate concrete code fragments from structure information and pseudo codes are described. To generate class codes from a given pattern, it is necessary to build up a class tree with an ID and other information obtained by asking users while tracing the a design pattern tree. In short, the process is as follows: First, a user selects a pattern at the top node to include all classes to be generated, then he inputs one or more

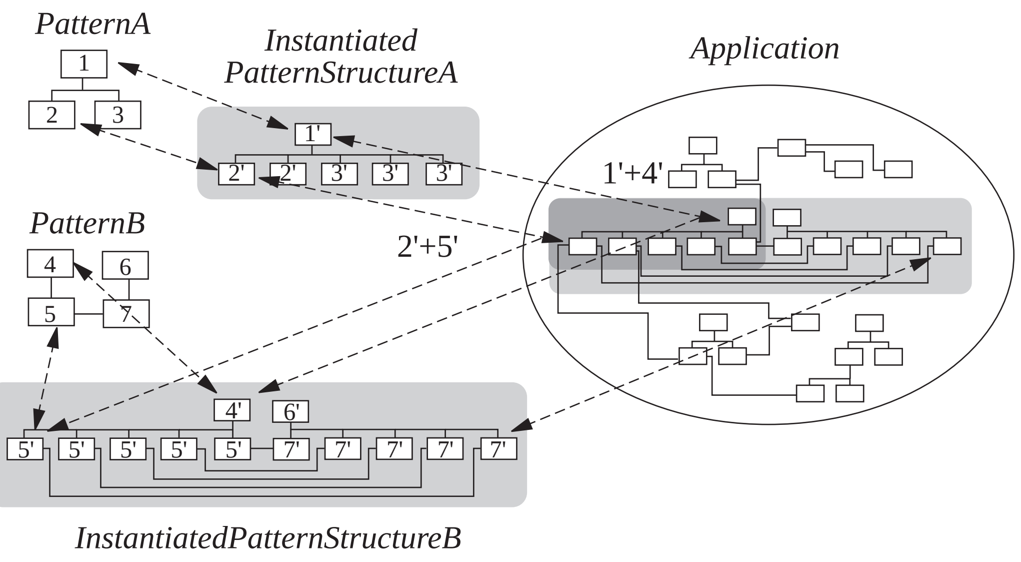


Figure 5.2: Relationships among design patterns, IPSs and classes

class identifiers for each role. For each identifier a class node is generated. Second, at each class node, the user inputs identifiers for aggregate and reference relations and methods of the corresponding role. For each identifier, a variable or a method node as a child of a class node is generated. Third, at each method node, the user inputs identifiers for arguments of the method.

To be independent from a certain language, the above information is processed separately from the one below. Then after the above information is satisfied, the user can select a language. The user supplements necessary specifiers for classes and methods, and edits method codes generated from pseudo codes.

As a result, information which must be supplied by a user. For example, identifiers for classes, aggregate and reference relations and methods as language independent information; specifiers, additional methods (option) and implementation of methods as language dependent information. In addition, there are selections which may modify the structure of the pattern briefly. In our research, the selections are not dealt with yet. After the two kinds of information are specified, application description files and codes are generated.

Start to make an application

$\downarrow$

Select a pattern

$\downarrow$

Instantiation

Input language independent information (name etc.)

$\downarrow$

$<$ IPS (Instantiated Pattern Structure) $>$

$\downarrow$

Specialization

$$\begin{tikzcd}
\text{Input language dependent information (specifiers etc.)} \arrow[d] \\
< \text{Internal expression of the application} > \arrow[d] \\
\text{Source code generation}
\end{tikzcd}$$

5.3 Cloning

When a class is cloned, sometimes other classes and methods must be cloned at the same time. A role in a pattern is a template. Therefore, it corresponds to several classes in the practical application. For example, `ConcreteAggregate` and `ConcreteIterator` in Iterator pattern correspond to several concrete classes such as, `SimpleList` and `SimpleListIterator`, `SkipList` and `SkipListIterator`, and so on. And an operation in a role may correspond to several methods in a class. For example, `CreateProduct` operation in `AbstractFactory` role corresponds to several methods corresponding to kinds of instantiated classed from `AbstractProduct` role. We call generating several classes/methods from roles/operations "cloning", and consider the necessary description and procedure to realize it. In other words, we consider how to describe the cloning constraints among roles and operations, and how to process the constraints.

5.3.1 Name Space

Before considering cloning constraints, name spaces to keep relationships among cloned elements (classes/methods) must be considered. It is necessary to relate relationships between roles (inheritance, aggregate, reference) and reference in operations to generated class and method names at code generation. Therefore, a pattern must be instantiated in the unit of roles and operations defined in the pattern. The unit is called "System Label Space", which is, in practice, a table containing pairs of a system label and a class or method.

When cloning, a new name space is generated. Elements cloned at the same time are related to their system labels in the new name space. Other elements are taken over from the previous name space. By this process, confusion by cloning can be prevented.

5.3.2 How to Describe Cloning Constraints

Two methods to describe the cloning constraints are considered. One is a method which describe the consequences from an element to another element by a constraint, this is, when a certain element is cloned, another elements must be cloned. The other is a method which describe a set of elements to be cloned at the same time as a group.

The latter group is gotten by resolving consequences along constraints. By resolving, it is easy to process cloning, since circulation of consequences is removed. In the group description, however, a pattern author must be careful to satisfy constraints.

The constraints are as follows:

- Constraint from a parent class to child classes in an inheritance relationship

  Whenever a super class is cloned, all sub classes must be cloned. On the other hand, when a sub class is cloned, a super class is cloned only if there are special constraints.

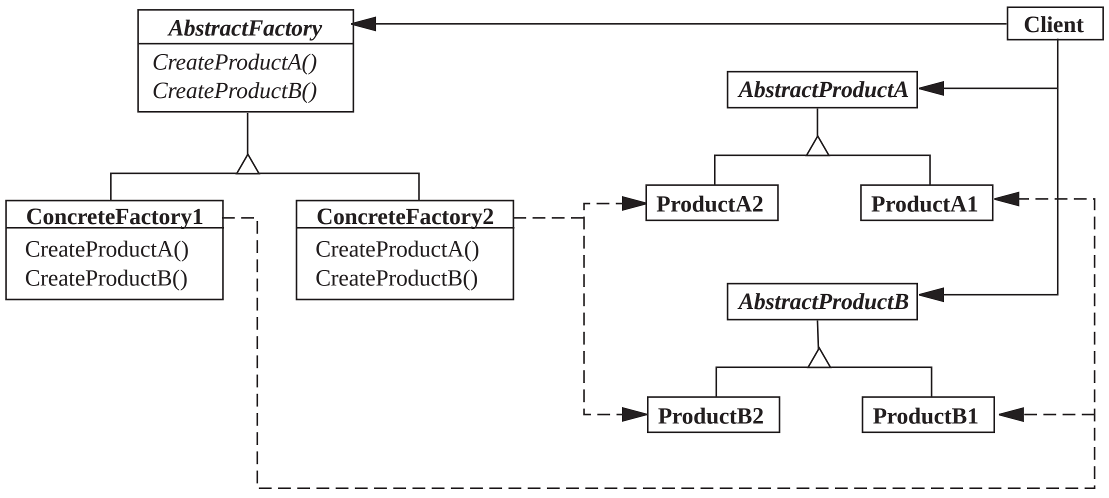


Figure 5.3: OMT chart for Abstract Factory Pattern

- Constraint from methods of a parent class to methods of child classes in a inheritance relationship

  Whenever an overridden method is cloned, methods overriding it are cloned. In the reversed case, the overridden method is cloned.

- Constraint from a class to methods included in it

- Constraint from a method to methods called from it

- Constraint from a construction method to the class containing it

In these constraints, only the ones related to method call can be defined by a pattern author. The others originate from the characteristics of object-oriented programs.

5.3.3 Example Using Abstract Factory Pattern

The following two kinds of constraint are selected concerning about Abstract Factory Pattern, which is one of patterns containing the most complex constraints. Two cases are evaluated, tracing consequences along the constraints and expanding them as groups.

Abstract Factory Pattern consists of four roles, `AbstractFactory`, `ConcreteFactory`, `AbstractProduct` and `ConcreteProduct`. `ConcreteFactory` inherits `AbstractFactory`. `ConcreteProduct` inherits `AbstractProduct`. `AbstractFactory` has an abstract operation named `CreateProduct`, and `ConcreteFactory` overrides it. `ConcreteFactory` creates objects of `ConcreteProduct` by using the operation. Fig. 5.3 is an OMT chart for the Abstract Factory Pattern.

There are two extra constraints written by a descriptor for the Abstract Factory Pattern. One is "when cloning `CreateProduct` operation in `AbstractFactory`, then `AbstractProduct` must be cloned. Vice versa." The other is "when cloning `ConcreteFactory`, `ConcreteProduct` must be cloned. Vice versa." These constraints cannot be satisfied at the same time. If the constrains can be satisfied at the same time, cloning `AbstractProduct` invokes cloning `ConcreteProduct`, then it invokes cloning `ConcreteFactory`.

The above constraints can be described by using expressions as follows. $C[label](First, Second)$ means "cloning First invokes cloning Second".

- By inheritance constraint (* means all elements)
  - $C[inherit1](AbstractFactory, ConcreteFactory*)$
  - $C[inherit2](AbstractProduct, ConcreteProduct*)$
- By override constraint
  - $C[override1a](AbstractFactory.CreateProduct, ConcreteFactory*.CreateProduct)$
  - $C[override1b](ConcreteFactory.CreateProduct, AbstractFactory.CreateProduct)$
- Constraints defined by a pattern descriptor
  - User Constraint 1
    - ◇ $C[userdef1a](AbstractFactory.CreateProduct, AbstractProduct)$
    - ◇ $C[userdef1b](AbstractProduct, AbstractFactory.CreateProduct)$
  - User Constraint 2
    - ◇ $C[userdef2a](ConcreteFactory, ConcreteProduct)$
    - ◇ $C[userdef2b](ConcreteProduct, ConcreteFactory)$

In this case, the remarkable point is the reason both $C[userdef1a,b]$ and $C[userdef2a,b]$ must not be applied at the same time. If they are applied at the same time, relationships between the methods called in `ConcreteFactory.CreateProduct()` (the constructor `ConcreteProduct()`) and its classes (`ConcreteProduct`) would be superfluous. Not only a constructor but also each method call, which has a one-to-one relationship to a class, would have the same problem. From this view point, one solution, to mark the relationships corresponding to classes and to clone them only if they include the methods.

The above constraints, indeed, can be converted into definitions of groups for cloning. In this thesis, such group is called CloningTemplate. For example, Abstract Factory Pattern contains two CloningTemplates as follows (Fig. 5.11):

```
CloningTemplate A (AbstractFactory::CreateProduct,
                   ConcreteFactory::CreateProduct,
                   AbstractProduct, ConcreteProduct)
CloningTemplate B (ConcreteFactory, ConcreteProduct)
```

To describe these template in PIML documents, three tags are introduced into PIML. The grammar for the tags is indicated in Table 5.1.

For example, the above CloningTemplates in Abstract Factory Pattern are described as follows:

Table 5.1: Cloneable part of PIML structure syntax

Cloneables	Cloneable*	Set of cloneable templates.
Cloneable	CloneElem*	A cloneable template consists of elements which must be cloned at the same time.
CloneElem	type: Type; id: Identifier	An element has attributes type (`role` for role or `op` for operation) and id for the target identifier.

```
<cloneables>
 <cloneable>
  <celem type="role" id="AbstractProduct">
  <celem type="role" id="ConcreteProduct">
  <celem type="op" id="AbstractFactory::CreateProduct">
  <celem type="op" id="ConcreteFactory::CreateProduct">
 </cloneable>

 <cloneable>
  <celem type="role" id="ConcreteProduct">
  <celem type="role" id="ConcreteFactory">
 </cloneable>
</cloneables>
```

Steps for cloning using the templates are as follows:

i) Select the target CloningGroup.

ii) Name each element to be cloned and clone them.

iii) At the same time, clone the name space which elements belong. The cloned name space contains other elements except for cloned elements

iv) Then, if the name space contains other cloning templates, relates them to the cloned elements.

If there are same elements in the instantiated CloningTemplates, CloningGroups are expanded or build newly. If there is not the same element in the ICTs, a new CloningGroup to contain each new ICT is created and added to the IPS.

v) The expanded or cloned CloneableGroups are again proposed to the user.

Table 5.2: Input data for Iterator pattern

Role name	→ Class Name
Iterator	→ Iterator
Aggregate	→ List
ConcreteIterator	→ SimpleListIterator
ConcreteAggregate	→ SimpleList
Client	→ IteratorTest
Operation Name	**→ Method Name**
CreateIterator	→ createIterator
First	→ first
Next	→ next
isDone	→ isDone
CurrentItem	→ getCurrentItem
Relation Name	**→ Variable Name**
aggregate	→ list
iterator	→ iterator
target	→ target

5.4 Prototype Implementation Using Java

Based on the above analysis, a Java source code generation support system from design patterns was designed and implemented. In concrete, from the separated process model, the language-dependent part and the language-independent part are implemented separately, and the language-dependent part can be replaced later (Figure 5.4). The three data structures for a design pattern, IPS and an application are defined using Composite pattern, and classes to trace and convert each structure are implemented using Visitor pattern. Using these patterns, localization and flexibility are increased. For example, the process of outputting final source codes is localized in ApplicationVisitor class which traces the structure for an application, and converts each node by tracing down recursively the tree structure whose top is Project.

There are two problems for converting pseudo codes into concrete languages. The first is how to describe and convert basic types in each language, and the second is how to describe and convert aggregate types. The latter problem, in particular, is related to expansion of `forall` statements, that is, statements expressing iteration. The statements must be replaced with concrete statements for each selected type. To solve this problem, we prepared a class called Aggregate, which defines actions of iteration for each type. The `forall` statements are replaced with the defined actions in it.

As for the former, as the latter, we prepared a class defining basic types for each language.

5.4.1 Input and Output Example for Iterator Pattern

Suppose inputs as Table 5.2 for Iterator pattern.

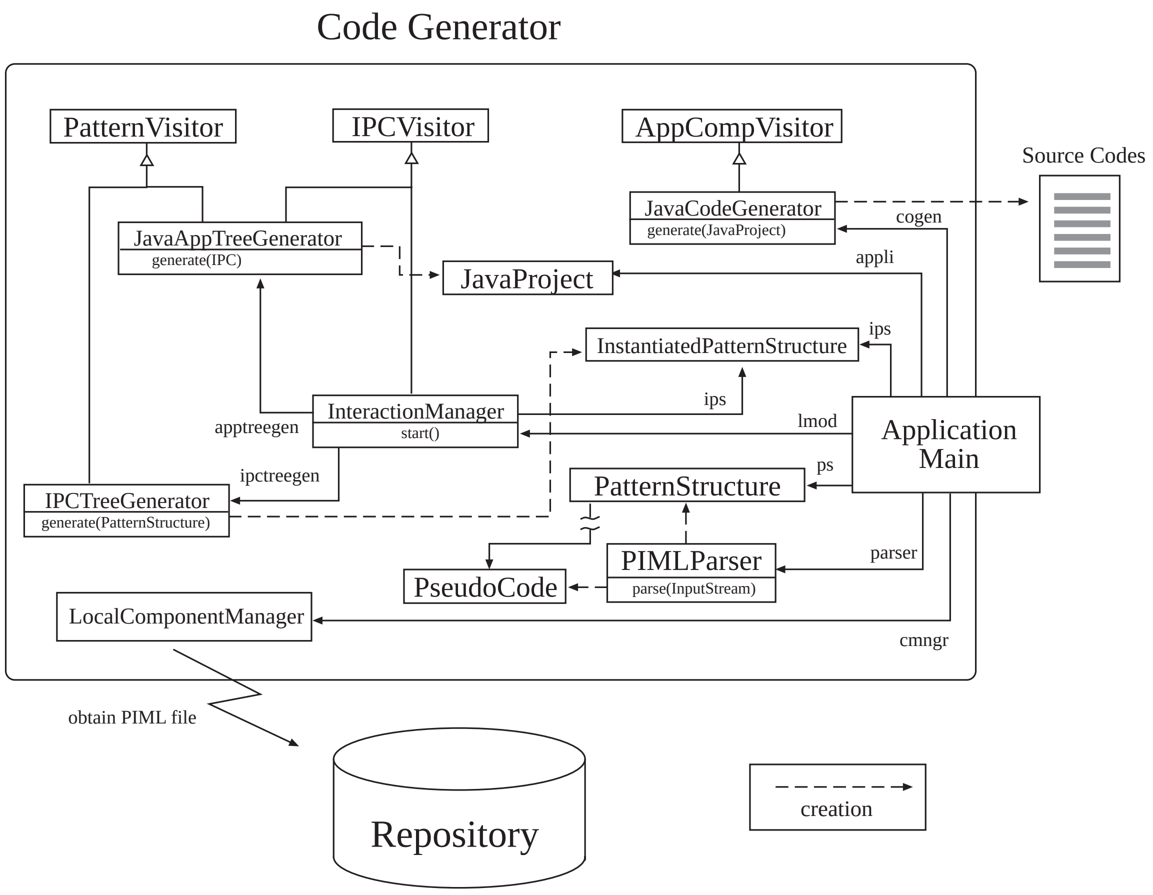


Figure 5.4: An overview of code generator

```
JAVAC = /usr/local/share/java1_1_1_ja/bin/javac
JAVA = java
CLASSPATH = -classpath .:/usr/local/share/java1_1_1_ja/lib/classes.zip
JAVAOPT = $(CLASSPATH)
CLASSES = SimpleListIterator.class SimpleList.class \
List.class Iterator.class IteratorTest.class
%.class: %.java
        $(JAVAC) $(JAVAOPT) $<
all: IteratorPatternTest
IteratorPatternTest: $(CLASSES)
clean:
        rm -f *.class *~
```

Figure 5.5: Generated Makefile

Output files are generated as shown in Figure 5.5, 5.6, 5.7, 5.8, 5.9 and 5.10 . These results indicate that correct names corresponding to relationships are given to classes, valiables, operations, and correct operation codes from pseudo codes. This code does not have application

```
public interface Iterator  {
  public abstract boolean isDone ();
  public abstract void next ();
  public abstract void first ();
  public abstract Object getCurrentItem ();
}
```

Figure 5.6: Generated Iterator class

```
public interface List  {
  public abstract Iterator createIterator ();
}
```

Figure 5.7: Generated List class

```
public class SimpleListIterator implements Iterator  {
  SimpleList target;
  public boolean isDone () {
    }
  public SimpleListIterator (SimpleList list) {
    target = list;
  }
  public void next () {
    }
  public void first () {
    }
  public Object getCurrentItem () {
    }
}
```

Figure 5.8: Generated SimpleListIterator class

```
public class SimpleList implements List  {
  public Iterator createIterator () {
    return new SimpleListIterator(this);
;
  }
}
```

Figure 5.9: Generated SimpleList class

dependent codes at all, they must be supplimented by the developer.

In addition, these data can be input using an input method on a web browser.

5.4.2 Cloning Example Using Abstract Factory Pattern

```
public class IteratorTest  {
  Iterator iterator;
  List target;
}
```

Figure 5.10: Generated IteratorTest class

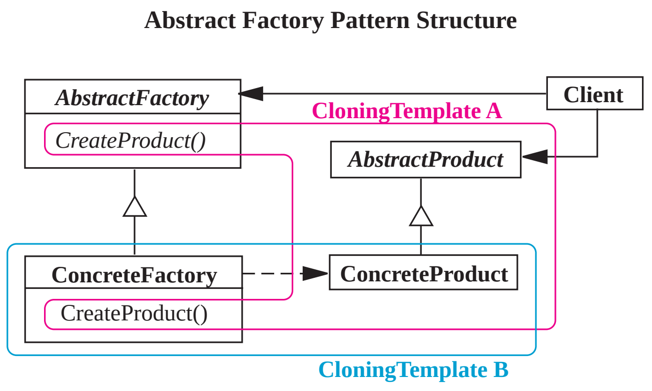


Figure 5.11: Cloning templates in Abstract Factory Pattern: A and B

First, suppose that a cloning template of Abstract Factory Pattern is instantiated as shown in Fig 5.12. For convenience, the instantiated template is called ICT (Instantiated Cloning Template). In addition to ICT, a group is introduced, which contains one or more than one ICTs and provides a constraint for users. Moreover, when cloning a pattern containing several cloning templates, if a cloning template is instantiated and its roles or methods are shared with other templates, at the next cloning, all contained templates must be cloned around the shared parts. By describing the constraints as cloning groups, all the roles and methods can be cloned.

For example, when instantiating `AbstractFactory`, suppose that `WidgetFactory` which exchanges window systems are instantiated such as in Figure 5.12. Then instantiated cloning templates (ICTs) are indicated as ICT A1 and ICT B1 in Figure 5.13.

This time, cloning groups (CGs) are indicated as in Figure 5.14.

For example, imagine what happens if you select CG B1, and clone it, then change two classes of them to `PMWidgetFactory` and `PMWindow`. We must pay attention not only to ICT B1 but also to ICT A1 that is cloned and the cloned classes are replaced. Thus, because the new ICT A2 shared a part of A1, the new CG A1 must include both ICT A1 and ICT A2.

All classes at this time are as shown in Figure 5.15. ICT A2 and B2 are indicated in Figure 5.16.

This time, cloning groups are shown in Figure 5.17.

In this state, suppose that CG A is selected and cloned. The elements to be named except for doubled ones are developed as follows:

```
CloningGroup A {
  WidgetFactory::CreateWindow,
  MotifWidgetFactory::CreateWindow,
```

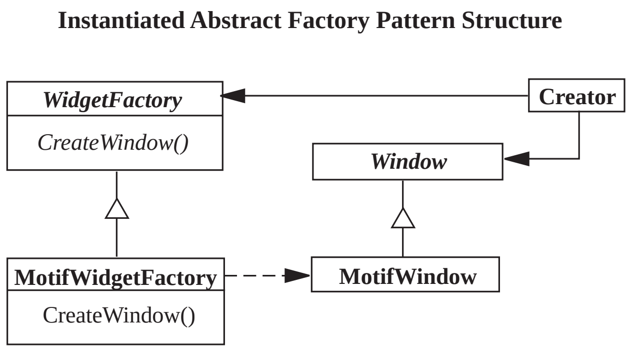


Figure 5.12: Cloning step 1

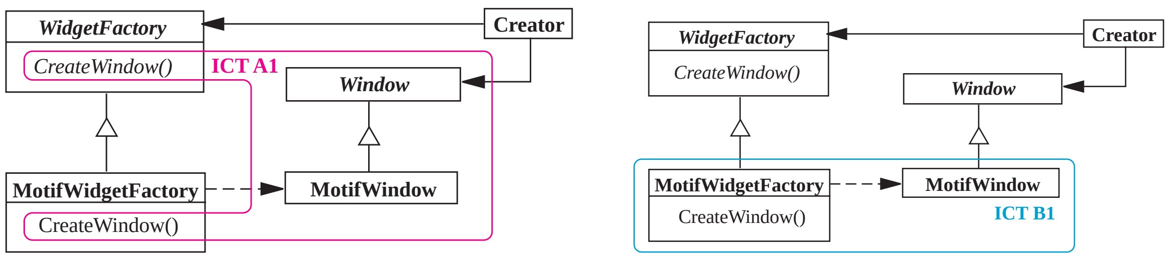


Figure 5.13: Instantiated cloneable templates, A1 and B1

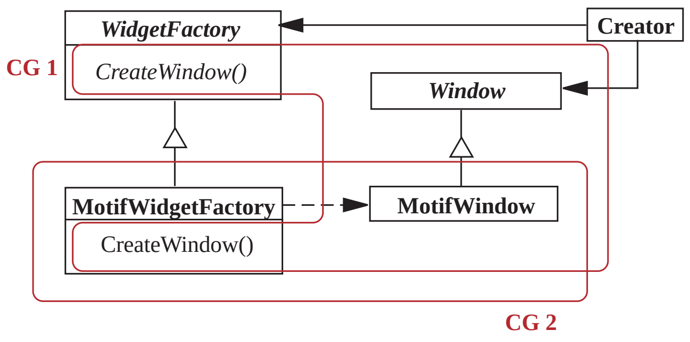


Figure 5.14: Cloning groups at step 1, CG 1 and CG 2

```
    PMWidgetFactory::CreateWindow,
    Window,
    MotifWindow,
    PMWindow
  }
```

In other words, these are elements related to “Window”. Then if “Window” is replaced “ScrollBar” and cloned, they are shown as follows. It shows complete cloning without name

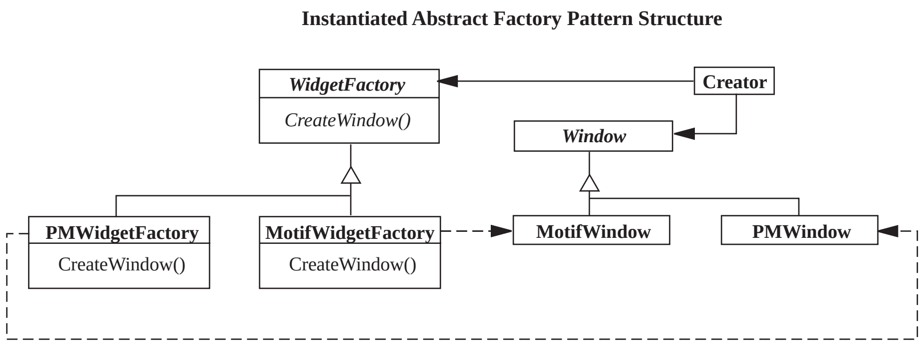


Figure 5.15: Cloning step 2

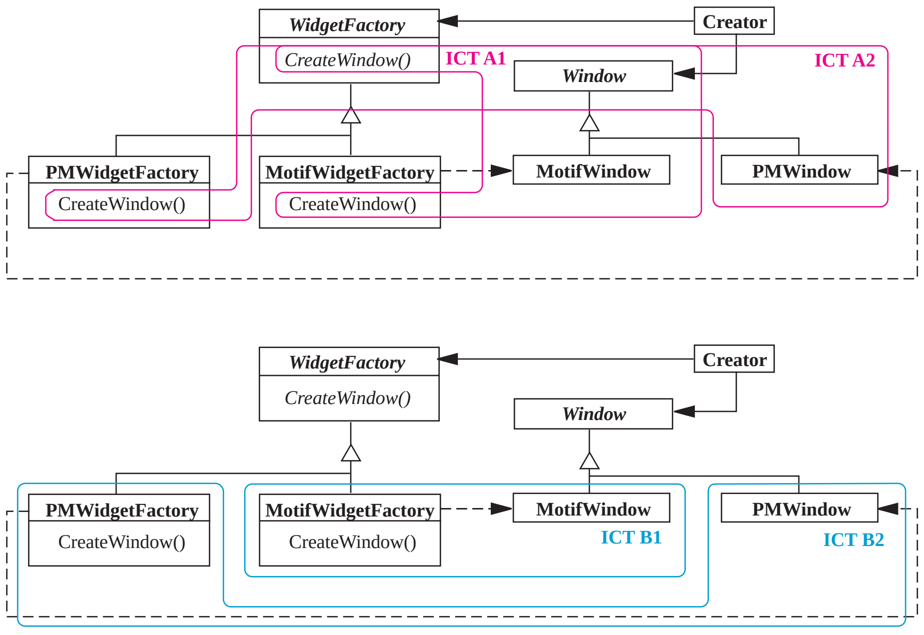


Figure 5.16: ICTs at cloning step 2, A1, A2, B1 and B2

confusion.

Figure 5.19 shows current ICTs, A1–4 and B1–4.

5.5 Related Works

Usual code generation systems, for example, CASE tools such as ROSE [44], support code generation using designing models for problems handled by an application. On the other hand, the code generation system using design patterns supports introducing problem-independent structures such as frameworks. Both problem-dependent and problem-independent structures should be supported. However, the former has already been studied in CASE tools,

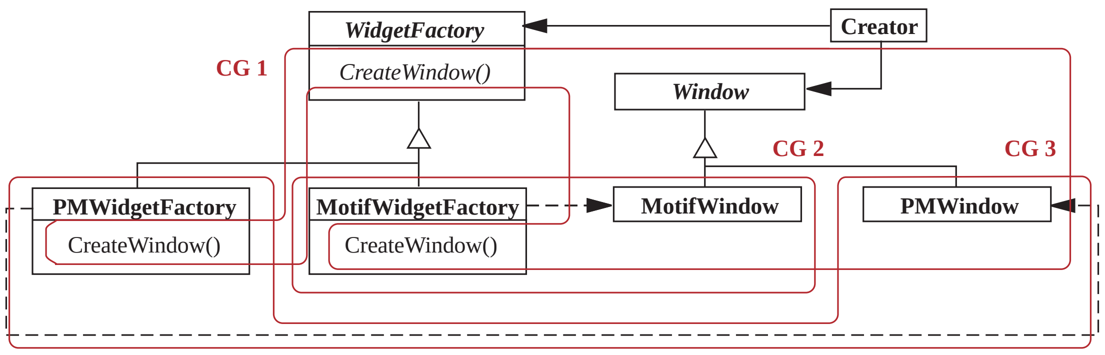


Figure 5.17: Cloning groups at step 2

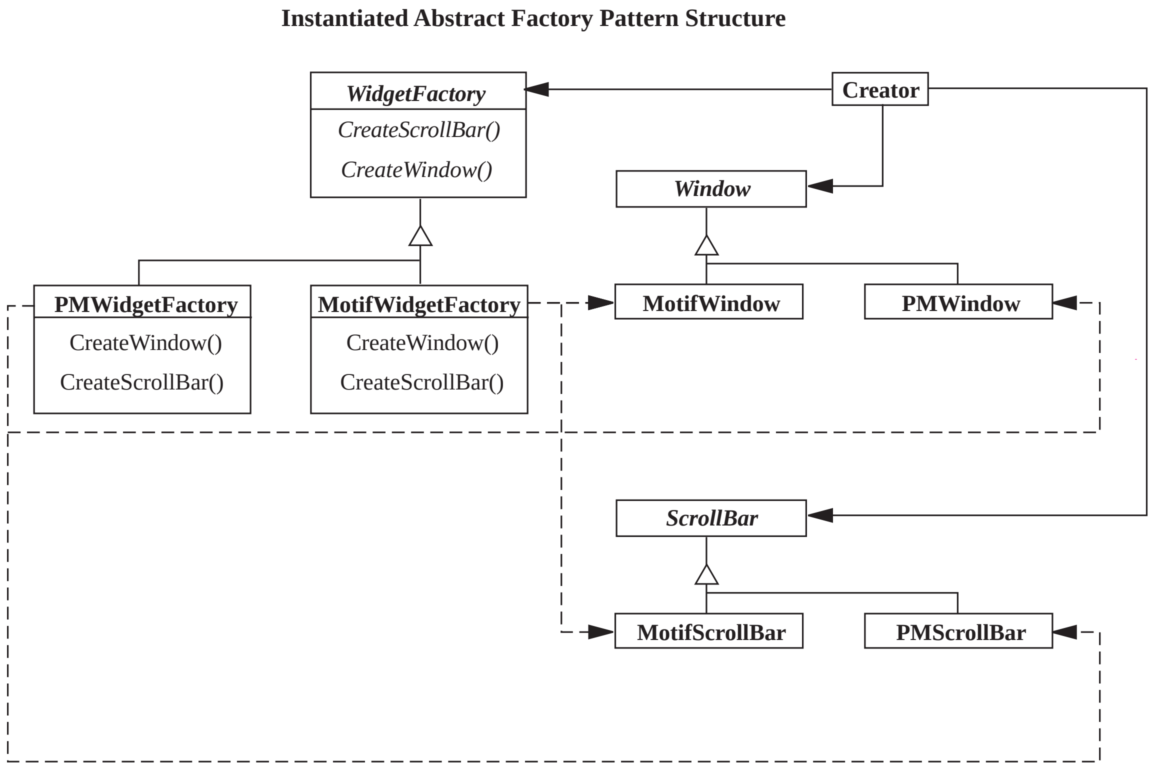


Figure 5.18: Cloning step 3

in this thesis the latter is studied.

There are several code generation systems using design patterns. In addition to IBM T. J. Watson Laboratory's system described in the previous chapter, Utrecht University's system [66] and so on. The difference from Utrecht University's system is that it generates codes by copying classes prepared for a certain design pattern. The problem with this generation method is that constraints to clone classes at the same time are not implemented. In other words, a user must take care of which classes to be copied together.

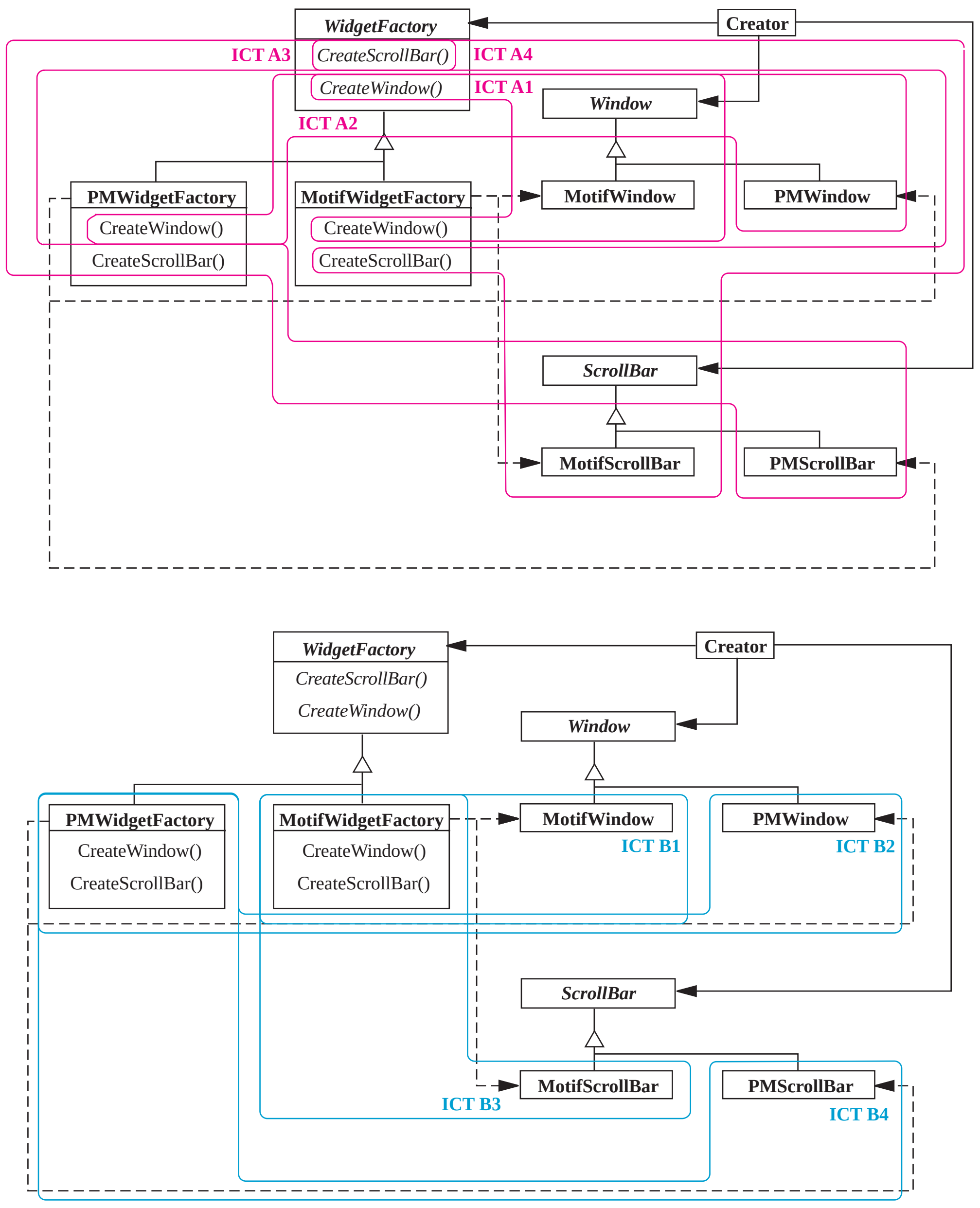


Figure 5.19: ICTs at step 3

5.6 Concluding Remarks

To promote the use of design patterns and to support the comprehension of an application by providing connections among design pattern components and source code components, we proposed a source code generation support system, designed it, and implemented a part of it. In concrete, we implemented Java source code generation support system using Java language. The language dependent part is separated from main generation support mechanism by designing the separated process model, therefore it can be applied to other languages easily.

Support for use of design patterns is achieved by decreasing complicated repetition. However, integration with source codes components, is not achieved yet.

There are future issues as follows:

- Integration with class components

- Support for reverse engineering

  To relate design patterns to source codes, source codes are generated from design patterns in our current system. However, reverse engineering, i. e. abstracting existing source codes to extract patterns corresponding to them is required for practical use. It is difficult to analyze the structure of source codes automatically. Therefore, we plan to do this through the interaction between the system and users.

Chapter 6

Conclusion

This thesis aims to foster the reuse of object-oriented software components shared by software developers in distributed environments. Frameworks to share components and relate them each other are necessary. For the first step to develop the frameworks, we provide two basic frameworks: a distributed component repository for class components and a structured document framework to deal with design patterns as components.

6.1 Distributed Component Repository and Class Components

We aim to manage various object-oriented software components which are connected each other. We proposed a system using the hypertext model and implemented a prototype. This prototype targets class components because they are basics for other object-oriented software components such as modules, libraries, specifications and patterns. The classes are extracted from source codes as fundamental units of object-oriented software components. The purpose of dealing with classes as components is relating them to other object-oriented software components. Class components are necessary to have several views as versions. In this thesis, from the point of view of reuse, we provide public and non-public views of a class component and express them as two hypertext nodes. If users need to modify a class, they can obtain the non-public node. If not, they can obtain only the public node.

Because components must be dealt with in distributed environments we adopt WWW (World Wide Web) as a distributed hypertext. To realize distribution transparency on WWW, we add a server to manage a ID-location table to it, and manage consistency of tables among servers. By putting an identifier as an argument for a CGI (Common Gateway Interface) program in any document, users can obtain a public interface node of the class component from the repository. In other words, users can access a class components regardless of its physical location, by querying the identifier to the ID-location management server through CGI, obtaining its location, and displaying the corresponding node. We call the mechanism a "distributed component repository".

Furthermore, in order to use usual libraries in our prototypes, we provide a mechanism to extract class information from C++ source codes and generate hypertext nodes corresponding classes. Because most libraries are provided with C++, we believed that it was suitable as subjects for the class component extraction mechanism. In practice, several classes in a

C++ library (actually the GNU C++ library) are registered in our distributed component repository, then they can be listed out and obtained through the list. Now, the extraction mechanism targets C++, however, it is independent from the distributed component repository. Therefore, it is possible to provide extraction mechanisms for any other languages.

Our distributed component repository provides a basis for sharing object-oriented components in distributed development environments, by using a framework of a distributed hypertext. A reuser (a user who wants to reuse components) can browse components by following relationships among them and obtain them without concerning their physical location. On the other hand, a provider of components can provide his/her own components regardless of the reuser's location, by registering them into our repository.

Now class components can be registered and reused in our repository. For example, the provider can put a URL consisting of the CGI program in our system and the class component ID in such documents as introduction documents. It is not necessary for the provider to concern effects by arranging structure and location of the source code files. The reuser can obtain the proposed class in hypertext format from the documents through the CGI program and follow classes related to the class by our system. We again point out that it is difficult to comprehend the class with only classes as now. We argue that a mechanism to share such design ideas and concepts as design patterns is necessary, and discuss the mechanism in this thesis.

Our repository mechanism can be used for the case in which data are distributed on the network. The case is difficult for a centralized repository to deal with. Our repository can collaborate with a single or centralized repository in managing distributed data as an assistant. In other words, it is possible to publish and browse object-oriented software components among cooperated organizations in their own repositories, by applying the sharing mechanism to them. However, we should consider how to register and update components shared by an internal repository and our distributed repository, and interoperability of data on practical use. Now a component provider selects class components to be shared and registers them by hand. In more concrete, he/she inputs source code files into the registration client program to register classes into the repository.

Extending distributed management mechanism is also a future issue. It includes adding facilities for cache and update notification. Version management should be considered.

6.2 Design Pattern Components

There are two purposes in dealing with design patterns as components. One is that it makes design support easy by providing electrical forms for them, because they are efficient to design reusable frameworks. The other is that it makes framework users easy to understand their structure, by realizing mutual relationships among design patterns and classes. Characteristics of our research are as follows:

First, we considered design patterns to be documents with semantics structure, and designed the document structure reflecting the semantics structure based on SGML. It enabled us to make a catalogue of, manage, share and publish design patterns like various documents which have been already applied to SGML.

Secondly, the structured document based on SGML contains not only information described in plain texts but also information described by figures and pseudo codes which cannot be described in plain texts. Because these descriptions relate to each other closely when browsing them and code generation, it is difficult to maintain consistency in usual formats in which they are separated.

Thirdly, for searching and browsing the integrated SGML description, we provide a dynamic generation and display system of description items and structure figures on demand for WWW browsers. By the generator, the user does not need to be concerned that design patterns are described structurally in SGML.

By dealing with design patterns as components in the form mentioned above, we can deal with them as software components. Therefore, they can be managed and used integrating other components, for example, source codes through code generation. The author of design patterns can describe design patterns as uniformed documents which can be processed in computers, then can check their consistency. Because SGML is a standard for electric commerce, the described design patterns can be published in wide area. The user can obtain published design patterns, and also can process them in a computer. Moreover, the user can browse design patterns on a WWW browser, and can get support by a computer at implementing them in practice.

Next we go on to describe the source code generation system mentioned in the previous chapter. The current target language in the generation system is Java, however, it is not limited to Java. The generation system can be applied such other languages as C++ and Smalltalk, by exchanging grammar description modules. Because Java is simpler than C++ as a object-oriented language, we considered that it is suitable for source code generation mechanism.

Source code generation support system aims to supply the necessary information in design patterns automatically. The information to be used for source code generation includes two kinds: information described in the "structure" item including the number of member classes, relationships among them and attributes and behaviors of them, and information described in the "implementation" item as trade-offs. In this thesis, we provide a pseudo code framework in addition to the SGML framework to describe the former information in PIML documents, and use the information at source code generation.

Therefore, a programmer using the source code generation support system can obtain not only a set of skeleton codes including classes which satisfy constraints of relationships (inheritance, reference, aggregate, creation and method call), but also method implementation codes if the operation in a design pattern corresponding to the method has pseudo codes. The generation of a whole application automatically is not the research subject. To begin with, design patterns are the knowledge to design classes to obtain flexibility and extendibility. Therefore, our generation system can generate only the implementation for the structure. Furthermore, such a subject can generate codes from an algorithm which requires studies of frameworks to deal with algorithms as components and to handle them like our research.

By providing a framework to describe design patterns as SGML documents and mechanisms for processing them which include a source code generation mechanism, reuse of design patterns as components are fostered. Henceforth, it is necessary to foster reuse of source codes by relating design patterns and source codes. The integration of source codes and design pattern components has not been achieved but a foundation has been built. For the integration, a format for storing in files must be discussed. There are some formats to be considered such as, SGML, source code with comments, a literate programming format such as NoWeb. For example, in Java, a mechanism for putting tags using comments into source codes are provided. In Perl, a document generating mechanism called POD is provided. In C++ development environments, there are a lot of formats to describe comments. CDIF is a standard for defining formats to exchange information among various CASE tools. Which is the best choice for saving source codes as components? We must discuss further.

Furthermore, as for other components, for example, specifications are not dealt with now, because several research teams have provided such a mechanism to realize relationships among them and source codes. More conceptual components as architecture is a future issue after design pattern components are achieved. We would like to integrate the components into our system. We are also considering the applicability of research to reverse engineering for design patterns [50] by relating class information to design pattern information. At the same time, we develop a support system to describe PIML documents.

Bibliography

[1] K. M. Anderson, R. N. Taylor, E. J. Whitehead, Jr. , "Chimera: Hypertext for Heterogeneous Software Environments", Proceedings of ECHT'94, Papers, pp. 94–107, 1994

[2] V. R. Basili and H. D. Rombach, "Towards a comprehensive framework for reuse: A reuse enabling software evolution environment". Tech. Rep. CS-TR-2158, Univ. of Maryland, 1988

[3] K. Beck, "Using a Pattern Language for Programming", (in Workshop on Specification and Design organized by Norman Kerth) Addendum to the Proceedings of OOPSLA'87, ACM SIGPLAN Notices, Vol. 23, No. 5, p. 16, 1988.

[4] J. Bigelow, "Hypertext and CASE", IEEE Software, pp. 23–27, 1988

[5] G. Booch, *OBJECT-ORIENTED ANALYSIS AND DESIGN with Applications 2nd Edition*, The Benjamin/Cummings Publishing Company, Inc., 1994

[6] C. L. Braun, NATO Standard for the development of reusable software components, volume 1 (of 3 documents), 1994

[7] C. L. Braun, NATO Standard for management of a reusable software component library, volume 2 (of 3 documents), 1994

[8] C. L. Braun, NATO Standard for software reuse procedures, volume 3 (of 3 documents), 1994

[9] S. V. Browne and J. W. Moore, "Reuse Library Interoperability and the World Wide Web", Proceedings of ICSE 97, pp. 684–691, 1997

[10] S. V. Browne, J. Dongarra, S. Green, K. Moore, T. Rowan, R. Wade, G. Fox, K. Hawick, K. Kennedy, J. Pool, R. Stevens, R. Olson and T. Disz, "The National HPCC Software Exchange," IEEE Computational Science and Engineering, Vol. 2, No. 2 pp. 62–69, 1955

[11] A. W. Brown, "An Examination of Current State of IPSE Technology", Proceedings of the 15th International Conference on Software Engineering, pp. 338–347, 1993

[12] F. J. Budinsky, M. A. Finnie, J. A. Vlissides and P. S. Yu, "Automatic Code Generation from Design Patterns", IBM Systems J. , Vol. 35, No. 2, 1996

[13] B. Campbell and J. M. Goodman, "HAM: A General Purpose Hypertext Abstract Machine", Communications of the ACM, Vol. 31, No. 7, pp. 856–861, 1988

[14] P. Cederqvist, “Version Management with CVS”, Technical Report, Signum Support AB, 1993

[15] D. Colman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes, *OBJECT-ORIENTED DEVELOPMENT The Fusion Method*, Prentice Hall, 1994

[16] J. O. Coplien and D. C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995

[17] P. Coad, “Object-Oriented Patterns”, Communications of ACM, Vol. 35, No. 9, 1992

[18] B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986

[19] M. A. Ellis and B. Stroutrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

[20] S. I. Feldman, “MAKE – A Program for Maintaining Computer Programs”, Software-Practice and Experience, Vol. 9, pp. 255–265, 1979

[21] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[22] D. Garlan, R. Allen and J. Ockerbloom, “Architectural Mismatch or Why it’s hard to build systems out of existing parts”, Proceeding of the 17th International Conference on Software Engineering, pp. 179–185, 1995

[23] S. Gibbs, D. Tsichritzis, E. Casais, O. Mierstraz and X. Pintado, “Class Management for Software Communities”, Communications of the ACM, pp. 90–103, 1990

[24] J. Gosling, B. Joy and G. L. Steele : *The Java Language Specification*, Addison-Wesley, 1996

[25] F. Halasz, M. Schwartz, “The Dexter Hypertext Reference Model”, Communications of ACM, Vol. 37, No. 2, pp. 30–39, 1994

[26] Y. Hamazaki, M. Tsukamoto, M. Ohnishi, Y. Niibe, “The Object Communication Mechanisms of OZ++: An Object-Oriented Distributed Programming Environment”, Proceedings of the IEEE 9th International Conf. on Information Networking (ICOIN-9), pp. 425–430, 1994

[27] R. Helm, I. M. Holland and D. Gangopadhyay, “Contracts: Specifying Behavioral Compositions in Object-Oriented Systems”, Proceedings of ECOOP/OOPSLA ’90, pp. 169–180, 1990

[28] U. Hölzle, “Integrating Independently-Developed Components in Object-Oriented Languages”, Proceedings of ECOOP ’93, LNCS 707, pp. 36–56, 1993

[29] A. L. Johnson and B. C. Johnson “Literate Programming Using Noweb”, Linux Journal, Vol. 42, pp. 64–69, 1997

[30] R. E. Johnson, "Frameworks = (Components + Patterns)", Communications of ACM, Vol. 40, No. 10, pp. 39–51, 1997

[31] G. Kiczales, J. de Rivieres and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991

[32] D. E. Knuth, *The Art of Computer Programming, Vol I: Foundation Algorithms*, Addison-Wesley, 1973

[33] D. E. Knuth, *The Art of Computer Programming, Vol II: Seminumerical Algorithms*, Addison-Wesley, 1973

[34] D. E. Knuth, *The Art of Computer Programming, Vol III: Sorting and Searching*, Addison-Wesley, 1973

[35] D. E. Knuth, *Literate Programming*, Leland Stanford Junior University, 1990

[36] R. E. Kraul, L. A. Streeler, "Coordination in Software Development", Communications of ACM, Vol. 88, No. 1, pp. 69–81, 1995

[37] K. J. Lieberherr, *Adaptive Object-Oriented Software*, PWS Publishing, 1995

[38] D. Marca, G. Bock, ed, *GROUPWARE: Software for Computer-Supported Cooperative Work*, IEEE Computer Society Press, 1992

[39] B. Meyer, *Object-Oriented Software Construction*, Interactive Software Engineering, Santa Barbara, California

[40] T.J. Mowbray and R. Zahavi, *The Essential CORBA: Systems Integration Using Distributed Objects*, Wiley/OMG, 1995

[41] P. A. Oberndorf, "The Common Ada Programming Support Environment (APSE) Interface Set (CAIS)", IEEE Transactions on Software Engineering, Vol. 14, No. 6, pp. 742–748, 1988

[42] M. Ohtsuki, N. Yoshida and A. Makinouchi, "A Distributed Repository for Object-Oriented Software Components", Proceedings APSEC '96, Vol. 2, pp. 47–54, 1996

[43] W. Pree, *Design Patterns for Object-Oriented Software Development*, the ACM Press, 1995

[44] T. Quatrani and G. Booch, *Visual Modeling with Rational Rose and UML*, Addison-Wesley, 1998

[45] M. F. Rochind, "The Source Code Control System", IEEE Transactions on Software Engineering, Vol. SE-1, No. 4 pp. 364–370, 1975

[46] J. Rumbaugh: *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[47] R. Sacks-Davis, T. Arnold-Moore and J. Zobel, "Database Systems for Structured Documents", IEICE Trans. on Information and Systems, Vol. E78-D, No. 11, pp. 1335–1342, 1995

[48] J. Sametinger, *Software Engineering with Reusable Components*, Springer-Veerlag Berlin Heidelberg, 1997

[49] S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis*, Prentice-Hall, 1988

[50] F. Shull, W. L. Melo, and V. R. Basili, "An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems", Tech. Rep. CS-TR-3597, Univ. of Maryland, 1996

[51] F. Shull, F. Lanubile, and V. R. Basili, "Investigating Reading Techniques for Framework Learning", Tech. Rep. CS-TR-3896, Univ. of Maryland, 1998

[52] J. B. Smith, S. F. WEISS "Hypertext", Communications of ACM, Vol. 31, No. 7, pp. 816–819, 1988

[53] Y. Sugiyama, "Configuration Management with Object Make", Information Processing Society Japan SIG Notes, pp. 9–15, 1994

[54] T. Suzuki, K. Kubota, M. Nakamura, E. Okubo and Y. Ohno, "A Structure of Program Development Support System C++base", Proceedings of the 49th National Convention IPSJ, 5-255, 1994

[55] Y. Tanaka, "Meme Media and a World-Wide Meme Pool", Proceedings of the Fourth ACM Multimedia Conference (MULTIMEDIA'96), pp. 175–186, ACM Press, 1996

[56] The Xerox Learning Research Group, "The Smalltalk-80 System", BYTE, Vol. 6, No. 8, pp. 57–69, 1981

[57] W. F. Tichy, "RCS—A System for Version Control", Software—Practice and Experience, Vol. 15, No. 7, pp. 637–654, 1985

[58] W. Tracz, *Confession of a Used Program Salesman: Institutionalizing Software Reuse*, Addison-Wesley, 1995

[59] J. Udell, "Componentware", Byte, Vol. 19, No. 5, pp. 46–56, 1994

[60] D. Unger and R. B. Smith, "Self: The Power of Simplicity", Proceedings of OOPSLA '87, pp. 227–242, 1987

[61] I. Vessey, A. P. Sravanapudi, "CASE Tools as Collaborative Support Technologies", Communications of ACM, Vol. 88, No. 1, pp. 83–95, 1995

[62] J. M. Vlissides, J. O. Coplien and N. L. Kerth (eds.), *Pattern Languages of Program Design 2*, Addison-Wesley, 1996

[63] K. Yasuda, at el, "Constructing a C++ Program Database", Information Processing Society Japan SIG Notes, 94-SE-96, pp. 145–152, 1994

[64] *Portland Pattern Repository*, http://c2.com/ppr/index.html

[65] http://ganesh.mit.edu/sanjeev/dp.html

[66] http://www.cs.ruu.nl/˜florijn/research/patterns.html,

[67] ISO 8879 Standard Generalized Markup Language (SGML), 1986

[68] ECMA : Portable Common Tool Environment (PCTE) Abstract Specification, Standard ECMA-149, 1990

[69] "Special Issue on Software Architecture", IEEE Trans. on Soft. Eng. Vol. 21, No. 4, 1995

[70] "Special Issue on Software Architecture", IEEE Software, Vol. 12, No. 6, 1995

List of Publications

Papers

i) Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
Distributed Hypertext for Component Management in Software Development
Proceedings of Data Engineering Workshop 95, pp. 119–125 (1995)

ii) Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
Distributed Hypertext-Based Repository for Object-Oriented Software Components
Proceedings of Object-Oriented Software Technology '95 Symposium, pp. 157–164 (1995)

iii) Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
Distributed Component Repository for Cooperative Object Oriented Software Development
Proceedings of Foundation of Software Engineering '95, pp. 207–212 (1995)

iv) Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
A Distributed Repository for Object-Oriented Software Components
Proceedings of the Asia Pacific Software Engineering Conference '96, pp. 439–446 (1996)

v) Mika Ohtsuki, Jun'ichi Segawa, Norihiko Yoshida and Akifumi Makinouchi
Structured Document Framework for Design Patterns Using SGML
Proceedings of Object-Oriented Software Technology '97 Symposium, pp. 31–38 (1997)

vi) Mika Ohtsuki, Jun-ichi Segawa, Norihiko Yoshida and Akifumi Makinouchi
Structured Document Framework for Design Patterns Based on SGML
Proceedings of COMPSAC 1997, pp. 320–323 (1997)

vii) Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
Distributed Management System for Object-Oriented Software Components
Research Reports on Information Science and Electrical Engineering of Kyushu University, Vol. 2, No. 2, pp. 259–364 (1997)

viii) Mika Ohtsuki, Jun'ichi Segawa, Norihiko Yoshida and Akifumi Makinouchi
SGML-based Structured Document Framework for Design Patterns and Its Browsing
Transactions of Information Processing Society of Japan, Vol. 39, No. 3, pp. 636–645 (1998)

ix) Mika Ohtsuki, Jun'ichi Segawa, Norihiko Yoshida and Akifumi Makinouchi
Visual Aids for Cataloging and Code Generation for SGML-based Documents of Design Patterns
Proceedings of IDPT 98, pp. 829–834 (1998)

Reports

i) Mika Ohtsuki and Norihiko Yoshida
An Object-Oriented Computation Model Based on Prototypical Objects
Proceedings of the 46th National Convention IPSJ [1], 4-141 (1993)

ii) Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
Component Management and Cooperative Development Support for Object-Oriented Software Using Distributed Hypertexts
IPSJ SIG Notes, 94-SE-101, pp. 49–56 (1994)

iii) Shingo Ishimatsu, Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
Distributed Makefile Construction in Distributed Software Component Repository
Record of 1994 Joint Conference of Electrical and Electronics Engineers in Kyushu, p. 733 (1994)

iv) Mika Ohtsuki, Shingo Ishimatsu, Norihiko Yoshida and Akifumi Makinouchi
Distributed Repository for Object-Oriented Software Components
Record of 1994 Joint Conference of Electrical and Electronics Engineers in Kyushu, p. 734 (1994)

v) Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
Integrating Design Patterns into Software Component Repository
Proceedings of the 53rd National Convention IPSJ, 1-255 (1996)
(Best Paper Award for Young Researchers of the 53rd National Convention of IPSJ)

vi) Jun'ichi Segawa, Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
User Interface of Aiding System for Design Patterns Based on SGML
Proceedings of the 55th National Convention IPSJ, 1-432 (1997)

[1] IPSJ stands for Information Processing Society of Japan

vii) Jun'ichi Segawa, Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
Automatic Generation of OMT Charts in Aiding System for Design Patterns
Record of 1997 Joint Conference of Electrical and Electronics Engineers in Kyushu, p. 221 (1997)

viii) Jun'ichi Segawa, Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi
Source Code Generation in Aiding System for Design Patterns Based on SGML
Proceedings of the 57th National Convention IPSJ, (1998)

ix) Shinji Hamada, Mika Ohtsuki, Jun'ichi Segawa, Norihiko Yoshida and Akifumi Makinouchi
Design and Implementation of Input Support in Aiding System for Design Patterns
Record of 1998 Joint Conference of Electrical and Electronics Engineers in Kyushu, (1998)

Appendix A

DTD for PIML

```
<!SGML "ISO 8879:1986"
   CHARSET BASESET
     "ISO 646-1983//CHARSET International Reference Version (IRV)//ESC 2/5 4/0"
   DESCSET   0   9 UNUSED
             9   2   9
            11   2 UNUSED
            13   1  13
            14  18 UNUSED
            32  95  32
           127   1 UNUSED
           128 127 128
           255   1 UNUSED
   CAPACITY PUBLIC "ISO 8879-1986//CAPACITY Reference//EN"
   SCOPE DOCUMENT
   SYNTAX -- PUBLIC "ISO 8879-1986//SYNTAX Reference//EN" --
      SHUNCHAR  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
       16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 127 255
      BASESET
     "ISO 646-1983//CHARSET International Reference Version (IRV)//ESC 2/5 4/0"
         DESCSET  0 256 0
      FUNCTION
         RE           13
         RS           10
         SPACE        32
         TAB SEPCHAR   9
      NAMING
         LCNMSTRT ""
         UCNMSTRT ""
         LCNMCHAR "_-."
         UCNMCHAR "_-."
         NAMECASE
            GENERAL YES
            ENTITY NO
      DELIM
         GENERAL SGMLREF
         SHORTREF SGMLREF
      NAMES SGMLREF
      QUANTITY SGMLREF
            NAMELEN   32
            LITLEN    1000
            ATTCNT    100
   FEATURES
   MINIMIZE DATATAG NO  OMITTAG  YES   RANK     YES SHORTTAG YES
   LINK     SIMPLE  NO  IMPLICIT NO    EXPLICIT NO
   OTHER    CONCUR  NO  SUBDOC   NO    FORMAL   NO
   APPINFO "Pattern Information Markup Language"
   >

<!DOCTYPE piml [

<!ELEMENT piml  - - (pattern) >
<!ELEMENT pattern  - - (intent? & motivation? &
 applicability? & consequences? &
 implementation? &
 sample_code? & known_uses? &
 related_patterns? &
 structure?) >
<!ELEMENT intent - - (RCDATA) >
<!ELEMENT motivation     - -  (RCDATA) >
<!ELEMENT applicability     - -  (RCDATA) >
<!ELEMENT consequences      - -  (RCDATA) >
<!ELEMENT implementation    - -  (RCDATA) >
<!ELEMENT sample_code       - -  (RCDATA) >
<!ELEMENT known_uses        - -  (RCDATA) >
<!ELEMENT related_patterns  - -  (RCDATA) >

<!ELEMENT structure - -  (notes? & relations? &
 cloneables? & roles & layout) >
<!ELEMENT notes - - (#PCDATA) >
<!ELEMENT relations - - (inheritance* & reference* &
 aggregate* & creation* ) >
<!ELEMENT cloneables - - (cloneable+) >
<!ELEMENT roles - -  (role+) >
<!ELEMENT layout - - (box+) >

<!ELEMENT inheritance - O  (EMPTY) >
<!ELEMENT reference - O  (EMPTY) >
<!ELEMENT aggregate - O  (EMPTY) >
<!ELEMENT creation - O  (EMPTY) >

<!ELEMENT cloneable - - (celem+) >
<!ELEMENT celem - O (EMPTY) >

<!ELEMENT role - -  (notes? & operations?) >
<!ELEMENT operations - -  (operation+) >
<!ELEMENT operation - O (EMPTY) >

<!ELEMENT box - O (EMPTY) >


<!ATTLIST pattern name NAME #REQUIRED
alias NAMES #IMPLIED>

<!ATTLIST inheritance to NAME #REQUIRED
from NAME #REQUIRED>
<!ATTLIST aggregate to NAME #REQUIRED
from NAME #REQUIRED
number (many|single) single>
<!ATTLIST reference to NAME #REQUIRED
from NAME #REQUIRED
number (many|single) single>
<!ATTLIST creation to NAME #REQUIRED
from NAME #REQUIRED>

<!ATTLIST celem type (role|op) #REQUIRED
id NAME #REQUIRED>

<!ATTLIST role syslabel NAME #REQUIRED
abstract (abstract|concrete) concrete>

<!ATTLIST operation syslabel NAME #REQUIRED
override  (done|do|not) not
return NAME #REQUIRED
access (public|protected|
 privateprotected|private) public>

<!ATTLIST layout rows NUMBER #REQUIRED
columns NUMBER #REQUIRED>
<!ATTLIST box name NAME #REQUIRED
row NUMBER #REQUIRED
column NUMBER #REQUIRED>
]>
```

Appendix B

Pseudo Code Syntax

```
statements:
      statement
    | statements statement
    ;
statement:
      method_call
    | if_statement
    | forall_statement
    | return_statement
    | assign_statement
    | dummy_statement
    ;
method_call:
      method
    | Identifier . method
    | Identifier :: method
    | constructor_call
    ;
constructor_call:
      new method
    | new Identifier :: method
    ;
method:
      Identifier ( )
    | Identifier ( args )
    ;
args:
      Identifier
    | Identifier , args
    ;
if_statement:
      if ( condition ) { statements }
    | if ( condition ) { statements } else { statements }
    ;
forall_statement:
      forall Identifier in Identifier { statements }
    ;
condition:
      Dummy
    ;
```

```
dummy_statement:
      Dummy
    ;
return_statement:
      return method_call
    | return Identifier
    ;
assign_statement:
      Identifier = Identifier
    | Identifier = method_call
    ;
```

Appendix C

PIML Example (Iterator Pattern)

In this example, the '`&endtago;`' entity in place of '`</`' exists in several elements. It is necessary because the motivation and the other elements are RCDATA and cannot include '`</`'.

```
<pattern name="Iterator" alias="Cursor">
 <intent>
Provide a way to access the elements of an aggregate object sequentially
without exposing its underlying representation.
 </intent>

 <motivation>
An aggregate object such as a list should give you a way to access its
elements without exposing its internal structure. Moreover, you might
want to traverse the list in different ways, depending on what you want
to accomplish. But you probably don't want to bloat the List interface
with operations for different traversals, even if you could anticipate
the ones we'll need. You might also need to have more than one traversal
pending on the same list.
<p>
The Iterator pattern lets you do all this. The key idea in this pattern
is to take the responsibility for access and traversal out of the list
object and put it into an <B>iterator&endtago;B>object. The Iterator class
defines an interface for accessing the list's elements. An iterator
object is responsible for keeping track of the current element; that is,
it knows which elements have been traversed already.
<p>
For example, a List class would call for a ListIterator with the
following relationship between them:
<IMG SRC="./ListIterator.gif">
<P>
Before you can instantiate ListIterator, you must supply, the List to
traverse. Once you have the ListIterator instance, you can access the
list's elements sequentially. The CurrentItem operation return the
current element in the list, First initializes the current element to
the first element, Next advances the current element to the next
element, and IsDone test whether we've advanced beyond the last
element-that is, we're finished with the traversal.
<P>
Separating the traversal mechanism from the List object lets us define
iterators for different traversal policies without enumerating them
in the List interface. For example, FilteringListIterator might provide
access only to those elements that satisfy specific filtering
constraints.
<p>
Notice that the iterator and the list are coupled, and the client must
know that is is a list that's traversed as opposed to some other
aggregate structure. It would be better if we could change the aggregate
class without changing client code. We can do this by generalizing the
iterator concept to support <B>polymorphic iteration&endtago;B>.
<P>
As an example, let's assume that we also have a SkipList implementation
```

```
of a list. A skiplist is a probabilistic data structure with
characteristics similar to balanced trees. We want to be able to write
code that works for both List and SkipList objects.
<P>
We define an AbstractList class that provides a common interface for
manipulating lists. Similarly, we need an abstract Iterator class that
defines a common iteration interface. Then we can define concrete
Iterator subclasses for the different list implementations. As a result,
the iteration mechanism becomes independent of concrete aggregate
classes.
<P>
The remaining problem is how to create the iterator. Since we want to
write code that's independent of the concrete List subclasses, we cannot
simply instantiate a specific class. Instead, we make the list objects
responsible for creating their corresponding iterator. This requires an
operation like CreateIterator through which clients request an iterator
object.
<P>
CreateIterator is an example of a factory method (see Factory
Method). We use it here to let a client ask a list object for the
appropriate iterator. The Factory Method approach give rise to two class
hierarchies, one for lists and another for iterators. The CreateIterator
factory method "connects" the two hierarchies.
 </motivation>

 <applicability>
Use the iterator pattern
<UL>
  <LI>to access an aggregate object's contents without exposing its
      internal representation.
  <LI>to support multiple traversals of aggregate objects.
  <LI>to provide a uniform interface for traversing different aggregate
      structures (that is, to support polymorphic iteration).
&endtago;UL>
 </applicability>


<consequences>
The Iterator pattern has three important consequences:
<P>
<OL>
  <LI>It supports variations in the traversal of an aggregate. Complex
      aggregates may be traversed in many ways. For example, code
      generation and semantic checking involve traversing parse
      trees. Code generation may traverse the parse tree inorder or
      preorder. Iterators make it easy to change the traversal
      algorithm: Just replace the iterator instance withe a different
      one. You can also define Iterator subclasses to support new
      traversals.
  <LI>Iterators simplify the Aggregate interface. Iterator's traversal
      interface obviates the need for a similar interface in Aggregate,
      thereby simplifying the aggregate's interface.
  <LI>More than one traversal can be pending on an aggregate. An
      iterator keeps track of its own traversal state. Therefore  you
      can have more than in progress at once.
&endtago;OL>
 </consequences>

<implementation>
Iterator has many implementation variants and alternatives. Some
important ones follow. The trade-offs often depend on the control
structures your language provides. Some language even support this
pattern directly.
<OL>
  <LI>Who controls the iteration? A fundamental issue is deciding which
      party controls the iteration, the iterator or the client that uses
      the iterator. When the client controls the iteration, the iterator
      is called an external iterator, and when the iterator controls it,
      the iterator is an internal iterator. Clients that use an external
```

```
    iterator must advance the traversal and request the next element
    explicitly from the iterator. In contrast, the client hands an
    internal iterator an operation to perform, and the iterator
    applies that operation to every element in the aggregate.
    <P>
    External iterators are more flexible than internal iterators. It's
    easy to compare two collections for equality with and external
    iterator, for example, but it's practically impossible with
    internal iterators. Internal iterators are especially weak in a
    language like C++ that does not provide anonymous functions,
    closures, or continuations like Smalltalk and CLOS. But on the
    other hand, internal iterators are easier to use, because they
    define the iteration logic for you.
 <LI>Who defines the traversal algorithm? the iterator is not the only
    place where the traversal algorithm can be defined. the aggregate
    might define the traversal algorithm and use the iterator to store
    just the state of the iteration. We call this kind of iterator a
    cursor, since it merely points to the current position in the
    aggregate. A client will invoke the Next operation on the
    aggregate with the cursor as an argument, and the Next operation
    will change the state of the cursor.
    <BR>
    If the iterator is responsible for the traversal algorithm, then
    it's easy to use different iteration algorithms on the same
    aggregate, and it can also be easier to reuse the same algorithm
    on difference aggregates. On the other hand, the traversal
    algorithm might need to access the private variables of the
    aggregate. If so, putting the traversal algorithm in the iterator
    violates the encapsulation of the aggregate.
 <LI>How robust is the iterator? It can be dangerous to modify an
    aggregate while you're traversing it. If elements are added or
    deleted from the aggregate, you might end up accessing an element
    twice or missing it completely. A simple solution is to copy the
    aggregate and traverse the copy, but that's too expensive to do in
    general.
    <BR>
    A robust iterator ensures that insertions and removals won't
    interfere with traversal, and it does it without copying the
    aggregate. There are many ways to implement robust iterators. Most
    rely on registering the iterator with the aggregate. On insertion
    or removal, the aggregate either adjusts the internal state of
    iterators it has produced, or it maintains information internally
    to ensure proper traversal.
    <BR>
     Kofler provides a good discussion of how robust iterators are
    implemented in ET++[Kof93]. Murray discusses the implementation of
    robust iterators for the USL StandardComponents' List class
    [Mur93].
<LI> Additional Iterator operations. The minimal interface to Iterator
    consists of the operations First, Next IsDone, and
    CurrentItem. Some additional operations might prove useful. For
    example, ordered aggregate can have a Previous operation that
    positions the iterator to the previous element. A SkipTo operation
    is useful for sorted or indexed collections. SkipTo positions the
    iterator to an object matching specific criteria.
<LI> Using polymorphic iterators in C++. Polymorphic iterators have
    their cost. They require the iterator object to be allocated
    dynamically by a factory method. Hence they should be used only
    when there's a need for polymorphism. Otherwise use concrete
    iterators' which can be allocated on the stack.
    Polymorphic iterators have another drawback: the client is
    responsible for deleting them. This is error-prone, because it's
    easy to forget to free a heapallocated iterator object when you're
    finished with it. That's especially likely when there are multiple
    exit points in an operation. And if an exception is triggered, the
    iterator object will never be freed.
    <BR>
    Th ##p:Proxy## pattern provides a remedy. We can use a
    stack-allocated proxy as a stand-in for the real iterator. The
```

```
      proxy deletes the iterator in its destructor. Thus when the proxy
      goes out of scope, the real iterator will get deallocated along
      with it. The proxy ensures proper cleanup, even in the face of
      exceptions. This is an application of the well-known C++ technique
      "resource allocation is initialization" [ES90]. The Sample Code
      gives an example.
 <LI> Iterators may have privileged access. An iterator can be viewed as
      an extension of the aggregate that created it. The iterator and
      the aggregate are tightly coupled. We can express this close
      relationship in C++ by making the iterator a friend of its
      aggregate. Then you don't need to define aggregate operations
      whose sole purpose is to let iterators implement traversal
      efficiently.
      <BR>
      However, such privileged access can make defining new traversals
      difficult, since it'll require changing the aggregate interface to
      add another friend. To avoid this problem, the Iterator class can
      include protected operations for accessing important but publicly
      unavailable members of the aggregate. Iterator subclasses(and only
      Iterator subclasses) may use these protected operations to gain
      privileged access to the aggregate.
 <LI> Iterators for composites. External iterators can be difficult to
      implement over recursive aggregate structures like those in the
      ##p:Composite## pattern, because a position in the structure may
      span many levels of nested aggregates. Therefore an external
      iterator has to store a path through the Composite to keep track
      of the current object. Sometimes it's easier just to use an
      internal iterator. It can record the current position simply by
      calling itself recursively, thereby storing the path implicitly in
      the call stack.
      <BR>
      If the nodes in a Composite have an interface for moving from a
      node to its siblings, parents, and children, then a cursor-based
      iterator may offer a better alternative. The cursor only needs to
      keep track of the current node; it can rely on the node interface
      to traverse the Composite.
      <BR>
      Composites often need to be traversed in more than one
      way. Preorder, postorder, inorder, and breadth-first traversals
      are common. You can support each kind of traversal with a
      different class of iterator.
 <LI> Null iterators. A NullIterator is a degenerate iterator that's
      helpful for handling boundary conditions. By definition, a
      NullIterator is always done with traversal; that is, its IsDone
      operation always evaluated to true.
      <BR>
      NullIterator can make traversing tree-structured aggregates (like
      Composites) easier. At each point in the traversal, we ask the
      current element for an iterator for its children. Aggregate
      elements return a concrete iterator as usual. But leaf elements
      return an instance of NullIterator. That let us implement
      traversal over the entire structure in a uniform way.
&endtago;OL>
 </implementation>


 <sample_code>
We'll look at the implementation of a simple List class, which is part of
our foundation library. We'll show two Iterator
implementations, one for traversing the List in front-to-back order,
and another for traversing back-to-front (the foundation library
supports only the first one). Then we show how to use these iterators
and how to avoid committing to a particular implementation. After
that, we change the design to make sure iterators get deleted
properly. The last example illustrates an internal iterator and
compares it to its external counterpart.
 </sample_code>

 <known_uses>
```

```
Iterators are common in object-oriented systems. Most collection
class libraries offer iterators in one form or another.
<P>
Here's an example from the Booch components, a popular collection
class library. It provides both a fixed size (bounded) and dynamically
growing (unbounded) implementation of a queue. The queue interface is
defined by an abstract Queue class. To support polymorphic iteration
over the different queue implementations the queue iterator is
implemented in the term of the abstract Queue class interface. This
variation has the advantage that you don't need a factory method to
ask the queue implementations for their appropriate iterator. However,
it requires the interface of the abstract Queue class to be powerful
enough to implement the iterator efficiently.
<BR>
Iterators don't have to be defined as explicitly in Smalltalk. The
standard collection classes (Bag, Set, Dictionary, OrderedCollection,
String, etc)define an internal iterator method do:, which takes a
block(i.e., closure) as an argument. Each element in the collection is
bound to the local variable in the block; then the block is
executed. Smalltalk also includes a set of Stream classes that
support an iteratorlike interface. ReadStream is essentially an
Iterator, and it can act as an external iterator for all the
sequential collections. There are no standard external iterators for
nonsequential collections such as Set and Dictionary.
<P>
Polymorphic iterators and the cleanup Proxy described earlier are
provided by the ET++ container classes. The Unidraw graphical editing
framework classes use cursor-based iterators.
<P>
Object Windows2.0 provides a class hierarchy of iterators for
containers. You can iterate over different container types in the same
way. The ObjectWindow iteration syntax relies on overloading the
postincrement operatior ++ to advance the iteration.
 </known_uses>

<related_patterns>
##p:Composite##: Iterator are often applied to recursive structures such as
Composites.
<P>
##p:FactoryMethod##: Polymorphic iterators rely on factory methods to
instantiate the appropriate Iterator subclass.
<p>
##p:Memento##: is often used in conjunction with the Iterator pattern.
An iterator can use a memento to capture the state of an iteration.
The iterator stores the memento internally.
 </related_patterns>

 <structure>
  <!--
 Structure of Iterator Pattern -->
  <notes>
    A ConcreteIterator keeps track of the current object
    in the aggregate and can compute the succeeding object
    in the object in the traversal.
  </notes>

  <relations>
    <inheritance origin="ConcreteIterator" target="Iterator">
    <inheritance origin="ConcreteAggregate" target="Aggregate">
    <reference origin="Client" target="Iterator"
 syslabel="iterator">
    <reference origin="Client" target="Aggregate"
        syslabel="aggregate">
    <reference origin="ConcreteIterator" target="ConcreteAggregate"
syslabel="concreteaggregate">
    <creation origin="ConcreteAggregate" target="ConcreteIterator">
  </relations>

  <cloneables>
 <cloneable>
  <celem type="class" id="Composite">
 </cloneable>

 <cloneable>
  <celem type="class" id="Leaf">
 </cloneable>

 <cloneable>
  <celem type="op" id="Component::Operation">
  <celem type="op" id="Composite::Operation">
  <celem type="op" id="Leaf::Operation">
 </cloneable>
</cloneables>

<roles>
<!--
                          Iterator Role
 -->
<role syslabel="Iterator" abstract="abstract">
 <notes>
  <ul>
   <li> define an interface for accessing and traversing elements.
  &endtago;ul>
 </notes>

 <operations>
  <!-- Operations included in Iterator Role -->
  <operation override="done"
             access="public" return="void"
             syslabel="First">
   <notes>
   </notes>
  </operation>

  <operation override="done"
             access="public" return="void"
             syslabel="Next">
   <notes>
    <!-- Information of Operation -->
   </notes>
  </operation>

  <operation override="done"
             access="public" return="boolean"
             syslabel="isDone">
   <notes>
   </notes>
  </operation>

  <operation override="done"
             access="public" return="anytype"
             syslabel="CurrentItem">
   <notes>
   </notes>
  </operation>
 </operations>
</role>

<!--
                      ConcreteIterator Role
 -->
<role syslabel="ConcreteIterator" abstract="concrete">
 <notes>
  <ul>
   <li> Implements the Iterator interface
   <li> keep track of the current position in the traversal of the aggregate.
  &endtago;ul>
 </notes>
```

```
   <operations>
    <!--
Operations included in ConcreteIterator Role
     -->
    <operation override="do" constructor="constructor"
               access="public" return="ConcreteIterator"
               syslabel="ConcreteIterator">
     <args>
       <arg syslabel="target" type="ConcreteAggregate">
     </args>

     <notes>
     </notes>

     <pseudocode>
       "aggregate" = "target"
     </pseudocode>
    </operation>

    <operation override="do"
               access="public" return="void"
               syslabel="First">
     <notes>
     </notes>

     <pseudocode>
     </pseudocode>
    </operation>

    <operation override="do"
               access="public" return="void"
               syslabel="Next">
     <notes>
     </notes>
    </operation>

    <operation override="do"
               access="public" return="boolean"
               syslabel="isDone">
     <notes>
     </notes>
    </operation>

    <operation override="do"
               access="public" return="anytype"
               syslabel="CurrentItem">
     <notes>
     </notes>
    </operation>
   </operations>
  </role>

  <!--
                            Aggregate Role
    -->
  <role syslabel="Aggregate" abstract="abstract">
   <notes>
    <ul>
      <li> defines an interface for creating an Iterator object.
    &endtago;ul>
   </notes>

   <operations>
    <operation constructor="noconstructor" override="done"
               access="public" return="Iterator"
               syslabel="CreateIterator">
     <notes>
     </notes>
    </operation>
   </operations>
  </role>

  <!--
                       ConcreteAggregate Role
   -->
  <role syslabel="ConcreteAggregate" abstract="concrete">
   <notes>
    <ul>
      <li> implements the Iterator creation interface
           to return an interface of the proper ConcreteIterator.
    $endtago;ul>
   </notes>

   <operations>
    <!-- Operations included in Unknown Role -->
    <operation constructor="noconstructor" override="do"
               access="public" return="Iterator"
               syslabel="CreateIterator">
     <notes>
      <!-- Information of Operation -->
     </notes>

     <pseudocode>
      return new "ConcreteIterator" ( "this" )
     </pseudocode>
    </operation>
   </operations>
  </role>

  <!--
                            Client Role
   -->
  <role syslabel="Client" abstract="concrete">
   <notes>
    <!-- Information of Client Role -->
   </notes>

  </role>
  </roles>

  <cloneables>
   <cloneable>
     <celem type="role" id="ConcreteAggregate">
     <celem type="role" id="ConcreteIterator">
   </cloneable>
  </cloneables>

  <layout rows="2" columns="3">
    <box name="Aggregate" row="1" column="1">
    <box name="ConcreteAggregate" row="2" column="1">
    <box name="Iterator" row="1" column="3">
    <box name="ConcreteIterator" row="2" column="3">
    <box name="Client" row="1" column="2">
  </layout>

 </structure>
</pattern>
```