A Source Code Generation Support System Using Design Pattern Documents Based on SGML

Mika Ohtsuki
Research Center for Higher Education
Kyushu University
Fukuoka 812-8560, JAPAN
mika@rc.kyushu-u.ac.jp

Norihiko Yoshida
Department of Computer and Information Sciences
Nagasaki University
Nagasaki 852-8521, JAPAN
yoshida@cis.nagasaki-u.ac.jp

Akifumi Makinouchi
Graduate School of Information Science
Kyushu University
Fukuoka 812-8581, JAPAN
akifumi@is.kyushu-u.ac.jp

Abstract

Applying design patterns to design of an application makes it robust against issues related to extensibility and maintainability. However, currently, a designer must apply structures and constraints of design patterns to an application by hand, therefore mistakes can occur often. We have proposed a notation using SGML for describing design patterns and a support system for design patterns. We aim at providing a source code generation support system based on the notation and system. There have been many researches on semi-automatic application generators using existing knowledge such as libraries based on algorithms and data structure, and classes packaged based on the object-oriented technology. When using design patterns for source code generation support, the same requirements for such semi-automatic application generators must be considered. The requirements are lessening codes to be written by users, and satisfying constraints without directing users' attention to them.

In this paper, we consider the requirements and describe a design and implementation of the source code generation support system and results.

1. Introduction

Design patterns are design knowledge for solving issues related to extensibility and maintainability which are independent from problems concerned by an application. A design pattern is described in explanation texts and charts expressing its structure and behavior. Applying design patterns to design of an application makes it robust against the issues. However, currently, a designer must apply structures and constraints of design patterns to an application by hand, therefore mistakes can occur often.

We aim at promoting use of design patterns and have proposed a notation using SGML[1] for describing design patterns and a support system for using design patterns[2]. For promoting use of design patterns further, we aim at providing a source code generation support system from design patterns. The system not only supports integrating design patterns as design knowledge, but also aims at cooperating with application components in the distributed components repository which we have developed[3].

There have been many researches on semi-automatic application generators using existing knowledge. Several commercial development environments support integrating algorithm libraries and classes packaged based on the object-oriented technology, so as to lessen codes to be written by users. There are researches such as Draco[4] and GenVoca generators[5] which generates codes from abstract notations.

When using design patterns for source code generation support, the same issues as such semi-automatic application generators must be considered. The issues are, by using existing knowledge, how to lessen codes to be written by users, and how to satisfy constraints without directing users' attention to them.

In this paper, in Section 2, we explain design patterns and **PIML** (Pattern Information Markup Language), which is a language we have proposed for describing a design pattern.

In Section 3, We consider which information can be generated from design patterns automatically, and which constrains in design patterns must be satisfied. Then, in Section 4, we design the source code generation support system. In Section 5, we present implementation of a prototype system and results. In Section 6, we mention related works, and in Section 7, we mention conclusions.

2. Design Patterns and PIML

Design patterns are descriptions about structures appeared in applications frequently for solving issues which are independent from application-dependent problems. A design pattern is expressed as a structure including several classes. Most of design patterns aim at constructing reusable structures in an application by decreasing dependency of parts of the application and increasing extensibility in future[6].

For example, Figure 1 is the *Iterator* pattern expressed in the same notation as [7]'s. The five boxes except the left bottom box express classes. If a class box is separated by a line, a bold word above the line is a class name and words below the line are method names. The left bottom box is a pseudo-code expressing a behavior of the method connected with it by a dotted line with a circle. Lines connecting the class boxes represent relationships. An arrow means reference and a dotted arrow means creation. A word above an arrow is a name of it. A line with a triangle in the middle of it is an inheritance relationship, and the class on the top of the line is a parent class and the class on the bottom of the line is a child class. If a name word is italic, the class or method is abstract.

The *Iterator* pattern generalizes aggregate objects as an abstract class `Aggregate`, and provides another abstract class for iteration as `Iterator` which is independent from `Aggregate`. Therefore, it is unnecessary to show an internal structure of each aggregate object to its client objects, and it becomes easier to update and exchange the aggregate object in future.

[7] is a catalogue book of design patterns. It includes 23 patterns such as the *Iterator* pattern and *Composite* pattern. In this book, a design pattern includes items as follows: "name," "also known as," "intent," "motivation," "applicability," "structure," "participants," "collaborations," "consequences," "known uses," "related pattern."

Using design patterns for supporting design and understanding source codes is effective in software development. They must be described in an electric format providing functions for the use. We have proposed a language for describing design patterns using SGML[1] framework in [2]. We call the language **PIML** (Pattern Information Markup Language).

A PIML description of a design pattern consists of

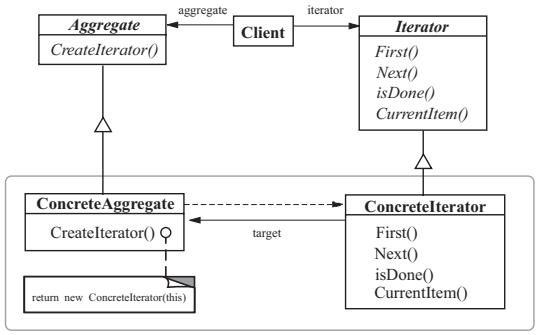

Figure 1. Iterator pattern

three parts. They are explanation texts such as motivation and availability, structure information such as constituent classes and relationships among them, and pseudo-codes expressing behaviors of classes. The explanation texts and structure information are described using SGML elements, and the pseudo-codes are described in a simple programming language designed by us originally. The reason of designing a simple programming language for describing the pseudo-codes is for using them for display as is, and using the pseudo-codes at source code generation.

The SGML elements are based on item names in [7]. Figure 2 indicates the correspondence of PIML description elements to items of a design pattern. The name of a design pattern is contained in the `pattern` element as an attribute. The items "intent" and "motivation" are enclosed by pairs of tags, (`<intent>`, `</intent>`) and (`<motivation>`, `</motivation>`). The class diagram expressing structure is converted into nested description in a pair of tags (`<structure>`, `</structure>`). Pseudo-codes also are contained in the `structure` element, but the details are omitted in the figure.

3. Source Code Generation from Design Patterns

We provide a source code generation support system for supporting design and supporting understanding source codes. It aims at using design patterns as components efficiently by automating to integrate design patterns into an application design as much as possible. It also aims at enabling users understand meanings of structures in source codes by relating them to design patterns which they are based on.

The source code generation support system must generate all source codes that can be generated automatically. Let us consider what is the information that can be generated automatically.

A constituent class and method of a design pattern can

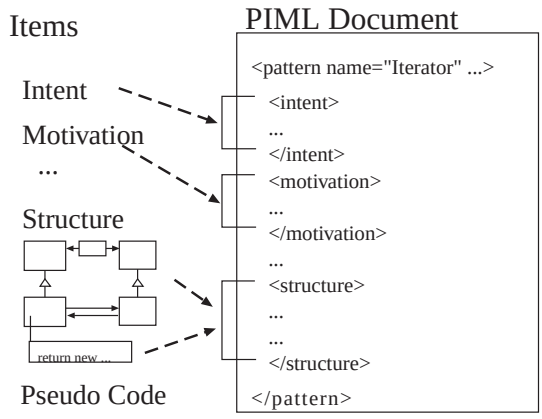

Figure 2. Correspondence of PIML description elements to items of design pattern

correspond to several classes and methods in an application. In this research, we call the class and method respectively a **role** and **operation**, to distinguish them from a class and method in an application. We call generating more than one class or method from one role or one operation **cloning**.

The information which a design pattern provides is as follows.

1. Roles as constituent elements.
2. Operations belonging to each role.
3. Relationships among the roles.
4. Behaviors of the roles for performing functions as a design pattern.

The former three are structural information and are described as nested elements in a PIML document. The last one is behavioral information and is described as pseudo-codes in a PIML document.

A PIML document has the structural information and the behavioral information in a `structure` element. A `structure` element contains a `relations` element consisting of elements expressing relationships and a `roles` element consisting of `role` elements. A `role` element expressing a role contains an `operation` element consisting of `operation` elements. An `operation` element expressing an operation contains an `args` element consisting `arg` elements and a `pseudocode` element expressing a pseudo-code. An `arg` element expresses an operation argument.

The code in Figure 3 is an extract from a PIML description of the *Iterator* pattern in Figure 1. The letters "`...`" in the code express omission. The *Iterator* pattern contains five roles (`Aggregate`, `ConcreteAggregate`, `Iterator`, `ConcreteIterator` and `Client`), however, `Iterator`, `ConcreteIterator` and `Client` are omitted because of limitation of space. As for the role `Aggregate`, its detail is omitted for the same reason.

```
<structure>
  ...
 <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="target">
  <creation origin="ConcreteAggregate"
            target="ConcreteIterator">
 </relations>
 <roles>
  ...
  <role syslabel="Aggregate" abstract="abstract">
   ...
  </role>
  ...
  <role syslabel="ConcreteAggregate"
        abstract="concrete">
   ...
   <operations>
    <operation constructor="noconstructor" override="do"
               access="public" return="Iterator"
               syslabel="CreateIterator">
     ...
     <pseudocode>
      return new "ConcreteIterator" ( "this" )
     </pseudocode>
    </operation>
   </operations>
  </role>
  ...
 </roles>
 <cloneables>
  ... (Information for Cloning) ...
 </cloneables>
 <layout rows="2" columns="3">
  ... (Layout Information) ...
 </layout>
</structure>
```

Figure 3. Structure description example of Iterator pattern

A `syslabel` attribute of a `role` element is an identifier of the role, and an `abstract` attribute expresses whether the role becomes an abstract class. In the code, the two roles have identifiers "`Aggregate`" and "`ConcreteAggregate`" respectively, the `Aggregate` role is abstract and the `ConcreteAggregate` role is concrete.

The `relations` element in the code contains six relationships among the roles. There are four kinds of relationships in PIML, `aggregate`, `reference`, `inheritance` and `creation`, and they are expressed as elements. An `aggregate` element and a `reference` element contain an identifier as a `syslabel` attribute. All the kinds of relationship has a direction, that is, a start point

and an end point. The start point is described in an `origin` attribute, and the end point is described in a `target` attribute. For example, in the code, a reference relationship named "`target`" has a start point role `ConcreteIterator` and an end point role `ConcreteAggregate`.

The `ConcreteAggregate` role has an operation named `CreateIterator()` containing a pseudo-code. There are five attributes in an `operation` element, in addition to a `syslabel` attribute for an identifier, an `override` attribute for an overriding relationship originated from inheritance, a `constructor` attribute as a flag whether the operation is a constructor, an `access` attribute for indicating an access level of the operation, and a `return` attribute for indicating a type of return value. For example, the `CreateIterator()` operation overrides on the same name operation in the parent role `Aggregate`, is not a constructor, is public and returns `Iterator`. The example pseudo-code in the `pseudocode` element expresses a behavior to generate an object of `ConcreteIterator` and return it as `Iterator`.

In addition to elements mentioned above, each of a `role` and an `operation` element contains a `notes` element for describing explanations of them. A `structure` element also contains a `layout` element for describing the layout information of roles used at displaying them. However, because these elements are not used for code generation, their details are omitted in this paper. The `cloneables` element and constituent elements of it are explained later.

Because functions of a design pattern do not provide solutions for application-dependent problems, the solutions must be provided by users. The structure information and behavioral information which a design pattern provides for adding extendibility and maintainability to an application can be converted into codes. However, the structure and algorithms for solving application-dependent problems must be described by users.

For example, in the *Iterator* pattern, classes can be generated from the five roles by replacing names of them, and codes in operations corresponding to `CreateIterator()` in the `ConcreteAggregate` role can be generated from its pseudo-code. However, other implementation codes must be provided by users, because the *Iterator* pattern does not provides it. For example, internal structure of classes generated form `ConcreteAggregate` and concrete operations, internal structure of classes corresponding to them and generated form `ConcreteIterator`, and implementation of operations such as `first()` in them.

The codes generated from pseudo-codes are not completely suitable to the target application. In other words, the behavior performed by the codes may have to be replaced by other functions or delegated to other objects. However, it is necessary to generate them, because they can be help for implementation by indicating behaviors in a design pattern as examples in the target programming language.

4. Design of the System

The structural and behavioral information described in a PIML document are parsed into a tree data structure called a **Pattern Structure** (PS). The tree data structure is used in a system which we have developed for authorizing and browsing design patterns[2]. The structure is designed using the *Visitor* pattern[7] to be traced by a converter separated from it. The separated converter is called a `ConcreteVisitor` in the *Visitor* pattern. A converter for instantiating a design pattern in the source code generation system is also designed as a `ConcreteVisitor` which can be replaced by the converter in the authoring and browsing system. It prompts users to input necessary information for conversion by tracing a tree of the target PS all along.

We design the generation process as follows. The details of data structures **IPS** (Instantiated Pattern Structure) and **AC** (Application Component) in the following process are explained in the following subsections.

1. Instantiating design patterns

   By giving names which are language-independent information, an IPS is generated from a PS.

2. Relating design patterns to an application

   An AC is generated from an IPS automatically. If necessary, concrete types and concrete implementation codes are added and changed.

3. Generating source codes

   Source codes are generated in the target programming language from an AC.

Figure 4 expresses the input and transformation of structures.

The design is based on the requirements described in the following subsections.

4.1. Satisfying Constraints

The followings are the constraints which should be satisfied by classes generated from a design pattern.

1. Relationships described in a design pattern

   The relationships among generated classes must correspond to the relationships among roles. Furthermore, if an operation is overridden because of an inheritance relationship, the corresponding methods must be overridden also.

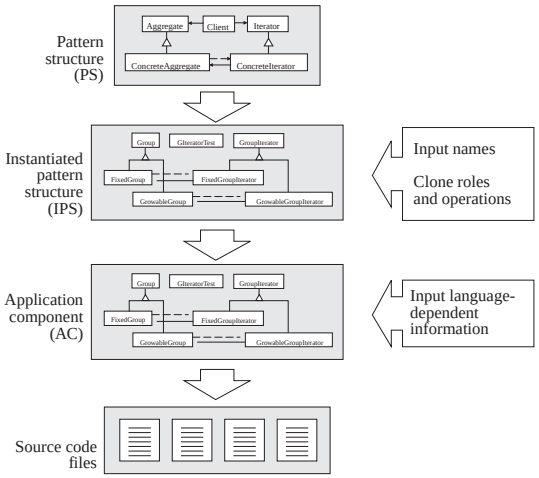

Figure 4. Code generation process

The relationship constraint can be satisfied, by converting relationship information held in a PS into the generated structure correctly. A PS has overriding relationships for overridden operations, and the generated methods to be overridden is attached the same overriding relationships as operations.

2. Correspondence of identifiers

   Identifiers of roles, operations and relationships in pseudo-codes and operation arguments must be replaced by names of classes, methods and variables corresponding to them. At the time, it is necessary to prevent confusing names of classes generated from a role.

   For example, the confusion can occur when pseudo-codes have a method call to a role which is not related directly to the role containing the codes. When there are relationships among roles for identifiers included in pseudo-codes and operation arguments, they can be distinguished by referring to the generated relationships. However, when there are not relationships for identifiers, the confusion among identifiers generated them can occur.

   To prevent the confusion of identifiers provided by unrelated roles, we provide a **name space**. A name space is a table to relate the unique names in a design pattern to the names of generated classes and methods. A name space is generated at cloning, and is added pairs of a name in a PS and a name in an IPS. A converter for code generation can obtain appropriate identifiers from the name space which the target belongs to.

   For example, when cloning based on the *Iterator* pattern as Figure 5, two name spaces are generated as follows. (`Iterator` → `Iterator`, `Aggregate` → `Group`, `ConcreteIterator` → `FixedGroupIterator`, `ConcreteAggregate` → `FixedGroup`) and (`Iterator` → `Iterator`, `Aggregate` → `Group`, `ConcreteIterator` → `GrowableGroupIterator`, `ConcreteAggregate` → `GrowableGroup`).

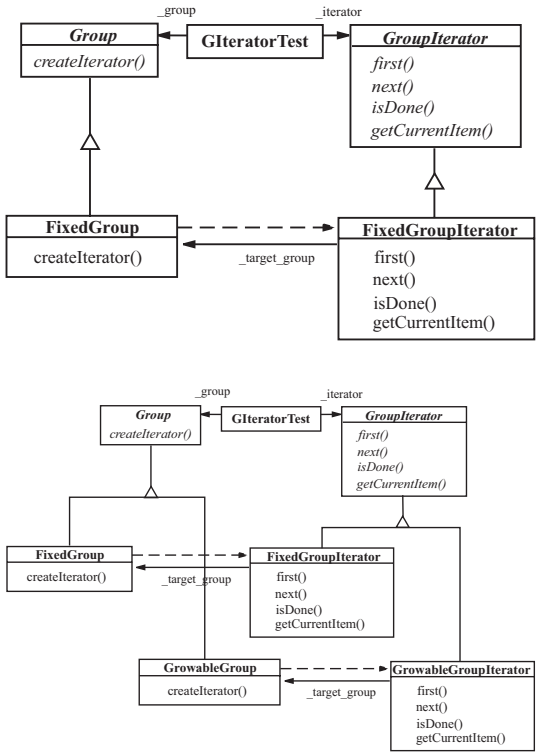

Figure 5. Instantiation and cloning of Iterator pattern

3. Cloning constraints

   When instantiating a role or an operation as a class or a method, sometimes other roles and operations must be instantiated at the same time. For example, in the *Iterator* pattern, because there is a constraint that `CreateIterator()` operation in `ConcreteAggregate` must generate `ConcreteIterator` (Figure 5), when instantiating one of `ConcreteAggregate` and `ConcreteIterator`, it is necessary to instantiate the other.

   This problem does not occur when a role or operation corresponds to only one class or method, but occur when a role or operation corresponds to several classes or methods. Therefore, in this research, the constraint is called a **cloning constraint**.

   Because the *Iterator* pattern has one cloning constraint

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

Figure 6. Cloneable element

and the cloning process itself is simple, effort and caution for maintaining consistency is not necessary so much. However, when there are several cloning constraints and cloning results based on a constraint affect the later cloning (for example, the *AbstractFactory* pattern[7]), much effort and caution for maintaining consistency among them is necessary. Therefore, it is important for the system to guarantee the consistency.

For solving this problem, first, we extend PIML to describe a cloning constraint. A cloning constraint can be inferred from relationships and extra constraints(they must be defined by a design pattern author). In this research, we introduce an element `cloneable` to describe a set of roles and operations which must be cloned at the same time. For example, the example constraint of the *Iterator* pattern mentioned above is described as Figure 6.

A `cloneable` element contains roles and operations to be cloned at the same time. Each role or operation is described in a `celem` element with its kind and its name (for example, "`role`" and "`ConcreteAggregate`"). A `cloneables` element can contain several `cloneable` elements, because there can be several constraints in a design pattern.

The system has several tables containing references to classes and operations, for keeping results of cloning as the candidates for the next cloning. By using the tables, the system can manage the consistency among results of cloning, when there are several cloning constraints and they affect one another.

4.2. Preventing Confusion among Design Patterns in an Application

An application can include several design patterns. Then, a class can include several roles for different functions as sets of methods. For modification later, it is necessary to distinguish them to prevent confusion.

To distinguish them, we provide an intermediate structure called an **Instantiated Design Pattern** (IPS). We call a structure containing an application information an **Application Component** (AC).

Strictly speaking, an AC is an abstract object and an interface of a concrete data structure for each programming language. The concrete data structure (for example, JavaAC) is a tree data structure and contains information such as environment information for constructing an application (the compiler, the library path, etc.), constituent classes and concrete detailed implementation codes. An AC is a tree data structure, of which each node is an interface of each node of the language-dependent data structures. An IPS can handle any of such language-dependent data structures through an AC as an interface. In this paper, "an AC" is used to express an object of any of the language-dependent data structures.

An IPS is also a tree data structure. One node of a PS corresponds to several nodes of an IPS. One node of an AC corresponds to several nodes of an IPS. One node of an IPS has one reference to a PS and one reference to an AC.

The root node of an IPS has an identifier and is independent from other IPSs. Therefore, it is possible to distinguish ones which are based on the same design pattern.

4.3. Converting an Abstract Description into a Concrete Language

Because a PIML description does not depend on a particular programming language, it is necessary to convert it into codes in a target programming language. To generate concrete source codes, it is necessary to convert basic types in PIML into types in the target programming language. And it is necessary to convert one-to-many relationships into aggregate classes or array types.

Furthermore, it is necessary to generate concrete codes for behaviors. Pseudo-codes are descriptions of behaviors in a design pattern, and has a simple grammar without arithmetic operations and comparison operation. It is necessary to convert the simple programming language to the grammar of the target programming language.

The gap is solved when converting an IPS to an AC.

The basic types which can be described in a PIML document are `int`, `void` and `boolean`. The system uses a table to convert them for each programming language. In addition to them, `anytype` can be described. `Anytype` is to indicate a type which is not defined in a design pattern, and is not a basic type either. Any classes in an application can be set in the place of `anytype`. For one-to-many relationships, the system provides a candidate list of array types and aggregate classes to enable users select.

Because the grammar of pseudo-codes consists of basic elements, there are scarcely problems at converting them into the target programming language. A special grammar element is a `foralll` statement to express a behavior to repeat certain operations for all child objects of the class the generated codes belong to. For converting a `foralll` state-

ment, the system provides a mechanism to convert each step for iteration (initializing iteration, proceeding count, getting the current element, checking statement and finalizing iteration) into concrete codes to operate array types or aggregate classes in the target language. When array types and aggregate classes are registered in the system, concrete codes for them corresponding to the iteration steps respectively are also registered. The system replaces `foralll` statements with the codes at generation.

The generated codes from pseudo-codes can be overwritten by users freely.

4.4. Separating Language-independent Parts from Language-dependent Parts

Because design patterns hardly depend on a programming language, it is useful to distinguish language-independent information and language-dependent information for adapting the system to multiple programming languages or converting applications into another language. Here, concretely, the language-independent information means mainly structure information such as class names and method names, and the language-independent information means concrete types for one-to-many relationships and concrete implementation codes necessary at execution.

The system provides two data structures for language-independent information and language-dependent information respectively, and two input processes for them. An IPS contains language-independent information and an AC contains language-dependent information. The converter from an IPS to an AC is based on *Visitor* pattern as the converter from a PS to an IPS, and converts an IPS recursively with tracing the IPS tree. Therefore, the system can be adapted to another programming language by replacing the converter with one for the language.

5. A Prototype and an Example of Generated Codes

We implement a prototype of the source code generation support system, based on a system which we have developed for authorizing and browsing design patterns[2]. The system reuses the PIML parser and data access part of the existing system.

All the required data structures described in the previous section, IPS, AC, tables for managing the cloning process and tables for relating pseudo-codes and a concrete language, are implemented in the prototype system. The system has converters for processing the data structures appropriately. Java is selected as the target language for the first prototype system, because it is a simpler object-oriented language than other widely-used languages.

FixedGroup.java

```
public class FixedGroup implements Group {
  protected Object _elements[];
  protected int _max_length;
  protected int _cur_length;

  public FixedPersonGroup(int max_length) {
    persons = new Object[_max_length = max_length];
    _cur_length = 0;
  }

  public int getMaxLength() {
    return _max_length;
  }

  public void add(Object o) {
    if (_cur_length>=_max_length)
      errorHandle();
    else
      _elements[_cur_length++] = o;
  }

  public void removeLast() {
    if (_cur_length<=0)
      errorHandle();
    else
      _elements[_cur_length--] = null;
  }

  public Object getAt(int idx) {
    return elements[idx];
  }

  /***
   *  auto generated
   */
  public Iterator createIterator () {
    return new FixedGroupIterator(this);
  }
```

FixedGroupIterator.java

```
public class FixedGroupIterator implements Iterator {
  FixedGroup _group;
  int _cur_idx;

  public FixedGroupIterator (FixedGroup group) {
    _group = group;
  }

  public boolean isDone () {
    return _cur_idx >= _group.getMaxLength();
  }

  public void next () {
    _cur_idx++;
  }

  public void first () {
    _cur_idx = 0;
  }

  public Object getCurrentItem () {
    return _group.getAt(_cur_idx);
  }
}
```

Makefile

```
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 = FixedGroupIterator.class FixedGroup.class ¥
          GrowableGroupIterator.class GrowableGroup.class ¥
          Group.class Iterator.class IteratorTest.class
%.class: %.java
        $(JAVAC) $(JAVAOPT) $<
all: myprog
myprog: $(CLASSES)
clean:
        rm -f *.class *~
```

Figure 7. Generated files

In addition to the data structures and converters, we implement interactive data inputting part as a Java Applet. A user can generate Java source code files and a simple `Makefile` by inputting necessary information interactively.

Figure 7 illustrates an example of source code files generated from the *Iterator* pattern. Here, the codes generated automatically are grayed and the others are made up by a user.

`FixedGroup.java` is generated from `ConcreteAggregate`. `CreateIterator()` operation defined in `ConcreteAggregate` is converted to `createIterator()` method, and concrete codes in it are generated from pseudo-codes in the operation. `FixedGroupIterator.java` is generated from `ConcreteIterator`. The methods `first()`, `next()`, `isDone()` and `getCurrentItem()` are generated from the corresponding operations in the role. Implementation of the methods is application-dependent and is supplemented by hand.

`Makefile` is all generated from environment information and information of constituent classes stored in an AC (strictly speaking, a JavaAC) automatically. We confirm correct codes generated for cloned classes.

6. Related Works

There are several researches on generating source codes from design patterns: Utrecht University[8], IBM T. J. Watson Laboratory[9], and Nihon University[10].

[8] provides design pattern templates consisting of objects in Smalltalk, and generates an application by cloning

them. The constraints to be satisfied by generated classes are checked after generation, and it does not support satisfying them automatically. Smalltalk is the only target language.

[9] describes information for generation using macros in its target language(C++). A converting program replaces names using the macros with given user's inputs automatically. It does not concern cloning constraints. For adapting the system to another language, macros for the target language must be prepared.

[10] provides models of design patterns in a CASE tool ROSE[11], and uses C++ skeleton codes generated by ROSE. Several constraints such as inheritance, reference and aggregate relationships are checked by a tool developed by [10] originally. The tool targets C++ only.

Cloning constraints are caused by not only relationships such as inheritance and aggregate, but also by other behavioral characteristics such as necessity of calling a particular method. Describing a design pattern using general modeling method does not express all cloning constraints. Therefore, the [10] system depending on the modeling method in ROSE cannot support the cloning process.

No other system than ours has the notation for expressing cloning constraints and supports the cloning process. Our system is easier to be extended than other researches, because it can be adapted to other languages by replacing a converting module.

7. Conclusions

We aim at promoting use of design patterns by supporting application creation from design patterns. We also aim at supporting understanding structures in a given application by relating it to design patterns through the creation process. We design a source code generation support system from design patterns, and implement a prototype system on the authorizing and browsing system of design patterns developed by us[2]. We design and implement data structure for keeping many-to-many relationships between design patterns and an application and for separating language-independent information and language-dependent information, and converters among them. We also design and implement the mechanism for supporting the cloning process. We confirm correct results by inputting test data.

We have to relate design patterns to application components in the distributed components repository developed by us[3], by extending design of the components. We have to examine integration of design patterns when adding another design pattern, and have to adapt the system for addition.

In future, we would like to examine reverse engineering such as relating structures in an application to design patterns interactively based on our system. And we would like to provide a more user-friendly integrated support environment.

References

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

[2] M. Ohtsuki, J. Segawa, N. Yoshida and A. Makinouchi, "SGML-based Structured Document Framework for Design Patterns and Its Browsing", Trans. of Information Processing Society of Japan, Vol. 39, No. 3, pp. 636–645, 1998, in Japanese

[3] M. Ohtsuki, N. Yoshida and A. Makinouchi, "A Distributed Repository for Object-Oriented Software Components", Proc. of the APSEC 96, pp. 439–446, 1996

[4] J. M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components", IEEE Trans. on Software Engineering, Vol. SE-10, No. 5, 1984

[5] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems Using Reusable Components", ACM Trans. on Software Engineering and Methodology, 1992

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

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

[8] G. Florin, M. Meijers, P. V. Winsen, "Tool Support for Object-Oriented Patterns", Proc. of ECOOP'97 pp. 472–495 (1997)

[9] 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

[10] J. Ichihara and Y. Sugiyama, "Supporting Design Patterns in ClassFactory", Proc. of the 57th National Convention IPSJ, pp. 223–224, 1998, in Japanese

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