FORMAL ANALYSIS OF INVARIANT-BASED DESIGN PATTERNS

By
HANY SHALABY, B.SC.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M.Sc.

Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2004) McMaster University
Hamilton, Ontario

TITLE: Formal Analysis of Invariant-Based Design Patterns

AUTHOR: Hany Shalaby, B.Sc.(McMaster University)

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: viii, 132

Abstract

This study introduces a formal abstract notation to describe precisely structural and behavioral aspects of design patterns. A pattern description is constructed based on the book of *Gamma et al.*, and is refined through analyzing carefully selected example implementations. Example implementations come from existing large real-world programs. Produced pattern descriptions can be used to check the compliance of other instances with intended patterns.

It is also shown that for some patterns, even the combined structural and behavioral descriptions are not enough to capture the essence of a pattern. An invariant has to be introduced to complement the pattern description. Based on components needed to describe patterns, they can be classified into structure-based patterns, behavior-based patterns, and invariant-based patterns. The latest require the most comprehensive description and are the focus of this study.

As a very detail-rich pattern, *Iterator* is selected as an example to apply the complete process on it. Invariants are also introduced for three other patterns to show that the need for invariants is not limited to one pattern.

The study shows that differentiating design patterns in their formal abstract notation is easier and more precise than when done based on class diagrams and natural language descriptions. Also, the process of establishing a pattern description can itself give a better insight about the essence and details of the described pattern.

Acknowledgements

Sincere thanks to:

My supervisor, Dr. Emil Sekerinski, for his support, thoughtful guidance and constructive criticism throughout the entire period of thesis preparation.

Dr. Ned Nedialkov, and Dr. Kamran Sartipi for their valuable comments and suggestions.

Dr. Ridha Khedri for his support and encouragement.

All my dear colleagues and department staff for their support and cooperation.

Contents

List of Figures

Chapter 1

Introduction

The claim of this thesis is that informal approaches for choosing and applying design patterns are not precise enough. The book of *Gamma et al.* [12] presents patterns in a well-organized form that is based on class diagrams and natural language. A natural language description can be ambiguous and may not fully capture the essence of a pattern. A designer may not be able to easily compare similar patterns. For instance, *Builder* and *Strategy* are both intended to provide a group of interchangeable implementations to perform some task. Therefore, a more precise description may be needed to make a proper distinction between them. A formal description may also provide an easier way to communicate those differences between designers. A well-defined formal description may be needed to check the correctness of a program instance. The correctness of an instance is determined by its compliance with the pattern formal description. Compliance with a pattern formal description may also maximize the reusability intended by the pattern.

We argue that describing design patterns in a formal abstract notation can improve the process of dealing with them. Describing patterns using abstract data types such as sets and bags allow more concise descriptions than code segments. Abstract pattern descriptions can be constructed after analyzing selected programs that implement them. Analyzing pattern instances from diverse application domains gives the most general description for the pattern.

Formal descriptions have the flexibility of integrating structural and behavioral aspects of patterns. They allow comparisons, verification, and proofs to be done within the same environment. It is also shown that for some patterns, complementing the description with

an invariant is essential to fully describe the pattern. Besides the benefit of choosing the right pattern, a formal description can also be used to check the compliance of a program instance with the pattern.

Iterator is selected in this study as an example to show the complete process on it. Invariants are also introduced for *Abstract Factory*, *Composite*, and *Singleton*.

1.1 Pattern Description

The language used in this study to describe patterns is an attempt to combine both the structural and behavioral aspects of each pattern.

The structural aspect may be viewed as a direct translation of the class diagram of the pattern. Capturing the structural side of a pattern allows checking if an instance complies with the pattern from the structural point of view. Structural compliance of an instance with the pattern does not mean that methods of the instance will have the same behavior expected from pattern methods.

The behavioral aspect provides the behavior expected from pattern methods [25]. Behavior is given in an abstract notation [30]. An instance complies with the pattern if it satisfies all the statements describing the behavior of the pattern. Statements in the pattern that are not satisfied by the instance usually imply that some behavior or benefits are missing. An instance may still partially comply with the pattern in this case as long as some behavior is supported.

Structural and behavioral compliance with the pattern may not be enough to conclude that a design is an instance of the pattern. In structural and behavioral compliance, we match the requirements of the pattern with statements in the instance. It is still possible to have extra statements in the instance contradicting with the expected pattern behavior. To avoid this kind of problems, we introduce *invariants* to patterns where applicable.

A pattern invariant may be viewed as a condition that needs to hold during the execution of the program. Unlike the check of structural and behavioral compliance, an instance can not partially preserve the invariant. A design that violates the invariant is not an instance of the pattern.

Formal analysis in this study uses common mathematical notation (sets, functions, etc.) to describe patterns. This allows direct comparisons, verification, and proofs as needed.

1.2 The Approach

The motivation behind formalization in this study is to:

- provide a way to examine the suitability of using a specific pattern in a design [23];
- provide means of checking compliance of a design instance with a well-defined pattern [14].

An attempt to achieve these goals is done as follows.

Description Components

- Pattern description: For each pattern, we analyze a reasonable number of programs that are based on the pattern. This is done to confirm our understanding of the pattern that is based on the book of *Gamma et al.* [12]. We then try to reach an abstract description of that pattern that captures both the structural and behavioral aspects of the pattern. The pattern is introduced as a module. A module is a representation of one or more classes. We give the formal description of classes involved in the pattern including special cases if any.
- Data structures: We define the module data structures that may be needed to express the pattern. A class is expressed as a set of objects. An attribute of type T defined in class C is expressed as a function from type *Object* to type T. More details about the formal language used are given in subsequent chapters.
- Pattern invariant: Invariant-based patterns are those patterns whose essence can not be completely captured without introducing an invariant. If this applies to the pattern we deal with, then we propose an appropriate invariant. We make sure that a proposed invariant is neither too strong nor too specific to a certain application. Invariants are expressed in terms of the defined data structures.

Well-Definedness of a Pattern

A produced pattern description is the basis for checking compliance of instances with the pattern. Therefore, it is necessary to check that the description is well-defined before using it. A significant step in case of invariant-based patterns is to show that methods of the

proposed pattern description preserve the invariant. In practice, these proofs have also led to finding missing statements in pattern descriptions. We also need to give an appropriate initialization for pattern data structures and show that this initialization establishes the invariant.

Checking Compliance of an Instance

A well-defined pattern description is used to check the compliance of instances with that pattern. This step applies whenever we need to check if a design that we produced complies with the pattern, or even to evaluate the compliance of existing programs with the pattern. The process of converting an instance into the introduced abstract notation can also be applied in the absence of source code. This is illustrated in an instance of *Iterator* shown later on. The abstract description of the instance is constructed based on the interface documentation of the design. This extends the applicability of the approach to include development based on specification.

The steps involved in checking compliance are as follows:

- Instance description: We start with giving the instance in the same format in which we have given the pattern itself. That includes the instance invariant if any. An instance invariant is specific to the instance application and may be stronger than the pattern invariant.

- Structural compliance: A check is done to see if the instance complies structurally with the pattern description. This check consists of matching structural statements of the formalized pattern with statements from the formalized instance.

- Behavioral compliance: A check is done to see if the instance complies behaviorally with the pattern description. This check consists of matching statements of formalized pattern methods with statements from formalized instance methods. It is required that the expected functionalities be matched in the instance without side effects that may interfere with them.

- Invariant preservation: A formalized instance has its own invariant. An instance invariant can be stronger than the proposed pattern invariant. A check is done to see

if formalized instance methods preserve the invariant. Doing this in an informal way is the most natural way for such a verification.

- Proof of invariant preservation: Formal proofs for invariant preservation may also help as a double-checking to make sure that formalized instance methods preserve the invariant.

1.3 Invariant-Based Patterns

Design patterns are already classified into three main categories, namely creational, structural and behavioral [12]. This classification follows from the functionality expected from each pattern. Following the formal analysis introduced in this study, we can classify patterns according to components needed to describe and verify them.

As a pattern has no unique abstract description, one pattern may fall in more than one category. The classification given below is associated with the pattern description language introduced in this study.

Structure-Based Patterns

These are patterns that can be completely described without defining class attributes. Direct translation of class diagrams of these patterns is usually enough to describe them. They involve no state and usually require no invariant or behavioral description. These patterns are usually based on call-redirections. An example is *Adapter*, which converts the interface of one class into that of another. Calls to methods in the *Adapter* class are simply redirected to methods in the *Adaptee* class. Structure-based patterns also include *Factory Method* and *Facade*.

Behavior-Based Patterns

These are patterns that cannot be completely described without introducing class attributes. In addition to a structural description, each of these patterns has methods with specific behavior that needs to be behaviorally described. *Observer* is an example of behavior-based patterns.

Invariant-Based Patterns

In some cases, both the structural and behavioral descriptions are not enough to capture the essence of the pattern. Those patterns usually involve implicit assumptions that can not be described structurally or behaviorally. Those cases are best covered by introducing invariants. Many patterns can either belong to behavior-based or invariant-based patterns depending on how general the proposed invariant is. Before concluding that an invariant is indeed a pattern invariant, a reasonable number of pattern instances need to be analyzed. The diversity of instance sources is essential to make sure that an invariant is neither too strong nor too specific to application field. Invariant-based patterns include *Iterator*, *Abstract Factory*, *Composite*, and *Singleton*. We focus in this study on invariant-based patterns, as the process of analyzing them covers all necessary steps to deal with all types of patterns.

1.4 Example Sources

Program examples that are used to illustrate the approach are carefully selected from various sources. Diversity in sources is a key decision to make sure that any derived conclusion is based on independent sources and examined on different application fields.

Most of the examples are implemented in *Java*. However, a few examples are given in other programming languages such as *C#*. This is to demonstrate that the techniques are applicable in any programming language that supports object-oriented features. As detailed below, some of the sources for applications are major, while others are only considered for diversity.

Design patterns are usually described using examples. That is why the first major example source may naturally be a set of learning programs. The *Java Design Patterns* book is one such source [5]. The book illustrates each pattern with a simple *Java* program that uses the pattern to solve a design problem. Example programs are given as a learning tutorial, so they are simple, and the main focus of programs is to illustrate patterns rather than what programs do.

Another major source is *JHotDraw*. It is a *Java GUI* framework that is intended to produce programs involving technical and structured graphics. The reason for considering this framework as a major example source, is that its design relies heavily on many well-

known design patterns [17]. Another reason is that an original author of the framework, *Erich Gamma*, is one of the authors of the book introducing design patterns [12].

The third major source used is the *Java* libraries. Even the early stages of *Java* libraries were packed with many design patterns [33]. *Erich Gamma* wrote an interesting article on this topic in 1996. The article shows how the original *Java* team integrated many design patterns in the *AWT* (Abstract Window Toolkit) design. He even shows as an example a relationship between five classes (*ComponentPeer*, *Component*, *Toolkit*, *Container* and *LayoutManager*) involving at least five different design patterns (*Composite*, *Strategy*, *Bridge*, *Abstract Factory* and *Singleton*) [11, 16].

All other sources used were considered for having a different implementation language, different application field or different implementation approach.

Chapter 2

Related Work

One related approach uses a specification method called *DisCo* that can be used to define a system using three components: classes, relations and actions [26]. Preconditions and postconditions are expressed in the form of relations and given to specify the actions within the system. Such a system is used to describe design patterns formally. *Observer* is introduced as an example. The approach is quite simple. However, it does not associate actions with classes. Therefore, no direct calls can be made to actions as introduced by the approach. Also, the approach does not consider a module invariant or module initialization.

The authors of [22] argue that many programs using a design patterns can be understood as formal refinements of a specifications not using the pattern. The approach formalizes the correctness of the transformation steps and uses the *Object Calculus*, a temporal logic, as the framework for reasoning. The paper also highlights key requirements needed by a system using a design pattern to be a correct refinement of the original system. The above case is illustrated on many design patterns, including creational patterns, structural patterns and behavioral patterns. The introduced framework allows adding invariants to pattern descriptions. An invariant is attached only to the *Observer* pattern, following the description of [12]. However, the only invariant introduced in the study is a trivial one. The approach does not involve invariants in the formal processing of the pattern. Pattern descriptions given are not justified or refined by comparisons with real examples. There is also no mention of the well-definedness of the given descriptions.

Another interesting study may be considered as a group of guidelines to find a good formalization rather than providing a specific approach [13]. Four interesting points are

given to evaluate a theory for design patterns. These points are:

- The possibility of formalizing both the problem and the solution given by the pattern. This is achieved in our study be giving a consistent representation for both the pattern and its instances. This is illustrated on all considered patterns as shown later on.

- The constraints for applying each design pattern. This was also addressed in our study. Formal statements used to describe patterns in our study represent constraints for applying these pattern in instances.

- The possibility of automating the process of applying a pattern, recognizing a pattern in a program, and discovering new patterns from repeated problems. Tools for design patterns manipulation are based on formal models to represent patterns. We introduce such a model to describe patterns in our study.

- The effective classification for design patterns. Our study even proposed a further classification for design patterns based on the components needed to describe them rather than what they do. This is shown latter on in this study.

A detailed method is given in another study using a sophisticated notation [14]. Patterns are abstract in order to be applicable in various domains and to achieve the desired reusability. Three formalization approaches are briefly introduced (*LePUS*, *DisCo* and *RSL*). A justification for adopting the third one is given. The approach is to generalize the model, formally specify how to match a design with a pattern, and then to include in the model a specification of the behavioral properties in it. It is claimed that the approach was applied on many common patterns. It has identified ambiguities and incompleteness in informal pattern descriptions, and has led to proposed new pattern structures [4]. One such proposed structure is for the *Builder* pattern. It is suggested that *Client* be added to the class diagram as an explicit participant in it. *Client* has one proposed method *Create* that makes the responsibilities of *Client* explicit. Those responsibilities are: instantiating *ConcreteBuilder* and *Director*, then, to invoke *Construct* method in class *Director* and *GetResult* in class *ConcreteBuilder*. The original class diagram does not include these actions and rather has them in an interaction diagram. However, the analysis of the introduced approach suggests making these interactions explicitly part of the pattern structure.

A quite different approach proposes a framework to represent patterns as a structured document based on *SGML* (Structured Generalized Markup Language) [27]. The approach integrates components of a pattern description (text, charts and pseudo-code) in a single document. That document will also include links to related documents and source code. Charts can be generated automatically from that document. A significant motivation for the approach is to allow the pattern to be effectively processed on a computer and to make a design patterns catalog available on the computer. All information are enclosed in markup language tags. For example, the intent of the pattern is enclosed between $<$ *intent* $>$ and $<$ /*intent* $>$ tags. The information needed to generate code and build charts also lie between *structure* tags. The description of methods is given in terms of simple abstract terms that can also be translated to real code (*Java* for example).

A technique is given in a study to formalize design patterns so that they can be identified in the source code [19]. A check to see if a design pattern is used correctly is also possible. The approach is to define roles of the pattern, annotate the code with comments that indicate those roles, and give rules to check the relationship between those roles. Decorator is used as an example in the study.

The significance of having tools for the application of design patterns is highlighted in another study [6]. The idea is to allow less experienced programmers to benefit from using them. The document introduces a method called *ADV* (Abstract Data View) to formalize patterns. *ADV* is given as the basis for creating such tools. A brief description is also given to one prototype tool.

An interesting study is motivated by the belief that design patterns are often used to create code but then they are usually forgotten [31]. It is also likely that modifications to the code will not conform to design patterns used. The solution proposed here is to allow programmers to work in terms of design patterns and source code simultaneously. A suite of tools is suggested to do this. *PEKOE* is a prototype tool that was developed to support the idea. Using the tool, patterns can be identified, created, verified, and edited in conjunction with source code. It is also possible to check if the pattern is still maintained by code that was modified. To reach these goals, a precise language is used to define design patterns. This language is based on breaking the pattern into elements and constraints. This will assist with creating queries to identify instances of patterns.

A study sees design patterns as abstractions that are used to describe portions of systems

so that designers can learn from them [29]. Some concepts are repeated within the common design patterns. Those concepts are considered as elementary patterns that can be used as building blocks to construct the common design patterns. The study provides those basic units and calls them *EDPs* (Elemental Design Patterns).

Another study claims that the problem with most formal approaches to describe patterns is the focus on either the structural or the behavioral aspect, but not on both [32]. *BPSL* (Balanced Pattern Specification Language) is introduced as a balanced, yet simple approach that integrates both structural and behavioral aspects of a pattern. *BPSL* combines two subsets of logic, *FOL* (First Order Logic) and *TLA* (Temporal Logic of Actions). A typical formalization using *BPSL* includes permanent relations, temporal relations, invariant, initial conditions, and actions. *Observer* is formalized as a case study. However, the invariant introduced is also trivial and is not involved in the formal processing of the pattern. Our study also achieves a balance between the structural and behavioral aspects of a pattern. However, our study represents behavior in a programming-like notation rather than temporal logic of actions. We also make a more precise notion for method calls, in particular which object triggers a call.

Chapter 3

Formal Pattern Analysis

In this chapter, we give more details about the formal language and techniques used in this study. The notation used is based on an object-oriented programing language introduced in a study to describe concurrent systems [28].

We assume that every expression e has a unique type T, written $e : T$. For a function f of type $T \rightarrow U$ the application to argument e of type T is written as $f(e)$. Predicates are expressions of type *boolean*, with values *true* and *false*. On predicates, we use the operators $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), and $\Leftarrow$ (consequence).

3.1 Syntax for Classes

We give first the formal syntax of the language in extended *BNF*. The construct $a \mid b$ stands for either a or b, $[a]$ means that a is optional, and $\{a\}$ means that a can be repeated zero or more times:

class	::=	**class** *identifier* [**implements** *identifier*]
		{ *attribute* | *initialization* | *method* }
		end
attribute	::=	**attr** *variableList*
initialization	::=	**initialization** (*variableList*) *statement*
method	::=	**method** *identifier* (*variableList*) [: *type*] *statement*
statement	::=	**assert** *expression* |

$$
\begin{array}{lll}
 & & \textit{identiferList} := \textit{expressionList} \mid \\
 & & \textit{identiferList} :\in \textit{expressionList} \mid \\
 & & \textit{identifier} := \textit{identifier}.\textit{identifier}\,(\,\textit{expressionList}\,) \mid \\
 & & \textit{identifier} := \textbf{new}\ \textit{identifier}\,(\,\textit{expressionList}\,) \mid \\
 & & \textbf{return}\ \textit{expression} \mid \\
 & & \textbf{begin}\ \textit{statement}\ \{\ ;\ \textit{statement}\ \}\ \textbf{end} \mid \\
 & & \textbf{if}\ \textit{expression}\ \textbf{then}\ \textit{statement}\ [\ \textbf{else}\ \textit{statement}\ ] \mid \\
 & & \textbf{while}\ \textit{expression}\ \textbf{do}\ \textit{statement} \\
 & & \textbf{for}\ \textit{expression}\ \textbf{do}\ \textit{statement} \\
 & & \textbf{var}\ \textit{variableList} \bullet \textit{statement} \\
\textit{variableList} & ::= & \textit{identifierList}\ :\ \textit{type}\ \{\ ,\ \textit{identifierList}\ :\ \textit{type}\ \} \\
\textit{identifierList} & ::= & \textit{identifier}\ \{\ ,\ \textit{identifier}\ \} \\
\textit{expressionList} & ::= & \textit{expression}\ \{\ ,\ \textit{expression}\ \}
\end{array}
$$

A class is declared by giving it a name, optionally stating that the class implements another class, and then listing all the attributes, initializations, and methods. Initializations have only value parameters, methods may have both value and result parameters. The assertion statement **assert** b checks whether boolean expression b holds. If it holds, it continues, otherwise it aborts. The object creation $x := \textbf{new}\ C(e)$ creates a new object of class C and calls its initialization with value parameter e. We do not further define *identifier* and *expression*.

Bag, Sequence and Set Notation

We use the notation $[\,]$ to indicate an empty bag (also known as multi-set) , $[x, y, ..., z]$ for the bag with elements $x, y, ..., z$, $A \cup B$ for the union of bags A and B, and $A - B$ for the subtraction of bag B from A. We use the notation $\langle\rangle$ to indicate an empty sequence, $A \,\&\, B$ for the concatenation of two sequences A and B, and $A - B$ for the subtraction of sequence B from sequence A. The predicate $x \in S$ evaluates to *true* if the element x is in sequence S at least once, and evaluates to *false* otherwise. Sequence subtraction $A - B$ removes all the occurrences of any element in B from A. We use the expression $\mathit{length}(s)$ to indicate the length of sequence s. We use the notation $\{\,\}$ to indicate an empty set, and

$| A |$ to indicate the cardinality of set A.

Abstract Statements
The nondeterministic assignment operator $:\in$ is introduced above as a part of the language used in this study. The assignment $x :\in s$ assigns to x any element from set, bag or sequence s as long as at least one element exists, otherwise, x is assigned the value *nil*. The assignment $x :\notin s$ assigns to x any element such that this element is not in s.

Transitive Closure of a Relation
The relation R^+ is the transitive closure of a binary relation R. It is defined to be the set of pairs (u, v) such that there is a path of length one or more from u to v. The pair (u, u) is in R^+ if and only if there is a cycle of length one or more from u to u.

Controlling Access to Members of a Class
The language introduced is intended as a module description language. Therefore, all attributes and methods are publicly accessible. We define the keyword *private* to give access privilege to a member only within its own class.

3.2 Classes and Modules

A pattern is represented by a module. A module is equivalent to a package containing one or more classes. Classes are declared by a class declaration, with every class we associate a set of objects of that class. A subclass is associated with a subset of its superclass set of objects. A module declares variables with initial values as well as procedures. Procedures operate on local variables and possibly variables declared in other modules. We use the following syntax for defining a module with two variables p, q and a single procedure m:

module K
  var $p : P := p_0$
  var $q : Q := q_0$
  procedure $m(u : U) : T$
    M
end

The class defines the attributes and the methods of objects. A class is defined in terms of a module with one variable for each attribute, one procedure for each method, and an extra variable for the objects populating that class. Variables map each object of the class to the corresponding attribute value. We allow variables to be of abstract types such as sets, bags, etc. Each procedure takes an additional value parameter, *this*, for the object to which the procedure is applied. We assume the type *Object* has infinitely many elements including the distinguished element *nil*. All objects are of type *Object*.

package R $\widehat{=}$ **module** R
  class C
    attr $a : A$
    static attr $b : B$
    initialization $(g : G)$
      I
    method $m(u : U) : V$
      M
    static method $n(s : S) : T$
      N
  end
  class D **inherits** C
    attr $e : E$
    initialization $(g : G)$
      J
    method $l(q : Q) : R$
      L
  end
end

  var $C, D :$ **set of** *Object* $:= \{\}, \{\}$
  var $C.a : Object \rightarrow A$
  var $D.e : Object \rightarrow E$
  var $C.b : B$
  procedure $C.new(g : G) : Object$
    var $this : Object \bullet$
    $this :\notin C \cup \{nil\}$; $C := C \cup \{this\}$;
    I ; **return** *this*
  procedure $D.new(h : H) : Object$
    var $this : Object \bullet$
    $this :\notin D \cup \{nil\}$; $C := C \cup \{this\}$;
    $D := D \cup \{this\}$; J ; **return** *this*
  procedure $C.m(this : Object, u : U) : V$
    assert $this \in C$; M
  procedure $C.n(s : S) : T$
    N
  procedure $D.l(this : Object, q : Q) : R$
    assert $this \in D$; L
  procedure $D.m(this : Object, u : U) : V$
    assert $this \in D$; $M[C.m \backslash D.m]$
  procedure $D.n(s : S) : T$
    N
end

Attribute e is introduced as a mapping between objects and elements of type E. Each object of class C maintains its own value for attribute e. Therefore, e is referred to by *this*.e within a method body. The notation $S[m\backslash n]$ stands for statement S with every occurrence of m replaced by n; it is used above to capture the redirection of calls in methods of C to methods overwritten in D. A *static* attribute b is introduced as a single element of type B. Therefore, all objects of class C share the same value for attribute b. Similarly, *static* method l is not associated with any object, while method m is associated with the caller object, referred to as *this*. A class name prefix as in the case of $C.new$ is dropped whenever there is no ambiguity. In general, referencing $x.e$ amounts to applying the mapping e to x. Creating a new object x of class C with initialization parameter e amounts to calling the *new* procedure of class C. Calling the method m of an object x of class C amounts to calling the procedure m of class C with x as the additional parameter that is bound to *this* in m:

$$
\begin{array}{lcl}
x.p & \widehat{=} & p(x) \\
x := \textbf{new}\ C(e) & \widehat{=} & x := C.new(e) \\
z := x.m(f) & \widehat{=} & z := C.m(x, f) \\
z := p(e) & \widehat{=} & \textbf{var}\ v, result \bullet \\
 & & v := e; S; z := result
\end{array}
$$

where p is declared by :

procedure $p(v)$

S

3.3 Invariant Notation and Proofs

We express invariants in terms of *typed logic*. Predicates that we produce are similar in context to the terminology introduced by common modeling languages. For instance, *OCL* (Object Constraint Language) is a formal language used to express constraints. It is intended to complement *UML* descriptions [30]. Even thought it uses a very similar notation to the one we use, we still prefer our more concise notation with commonly understood meaning. An example would be that *OCL* uses *forAll* and *exists* rather than $\forall$ and $\exists$ respectively. We also avoid the heavy baggage of complexity that may result from using more comprehensive notations like the *B−Method* [1] and *Z notation* [7].

Weakest Precondition *wp*

For a system denoted by S and having a desired post-condition denoted by R, we denote the corresponding weakest pre-condition by $wp(S,R)$ [10]. If the initial state satisfies $wp(S,R)$, the system is certain to establish eventually the truth of R. Because $wp(S,R)$ is the weakest pre-condition, we also know that if the initial state does not satisfy $wp(S,R)$, this guarantee can not be given, i.e. R may not hold or the system may even not terminate.

Weakest Liberal Precondition *wlp*

The weakest liberal precondition $wlp(S,R)$ is weaker than $wp(S,R)$ defined above. The precondition $wlp(S,R)$ only guarantiee that the system will not produce the wrong result, i.e. will not reach a final state not satisfying R, but nontermination is left as an alternative. The notion of $wp(S,R)$ and $wlp(S,R)$ also apply when we replace the post-condition R with an invariant P. We relate *wp* to *wlp* as follows: $wlp(S,R) \equiv wp(S,true) \Rightarrow wp(S,R)$.

Verifying Invariants

For a module M with variables $v1,v2,..$ represented as V having initial values $v1_0,v2_0,..$ represented as V_0 and public procedures represented by $S1,..,Sn$, predicate P is an *invariant* of M if:

1. Initial values establish P

   $$V = V_0 \Rightarrow p$$

2. Public procedures preserve P

   $$p \Rightarrow wlp(m1,p)$$
   $$\ldots$$
   $$p \Rightarrow wlp(mn,p)$$

The following rule is used in checking of invariant preservation:

$$P1 \textit{ is an invariant of } \mathrm{M} \wedge P2 \textit{ is an invariant of } \mathrm{M} \Rightarrow P1 \wedge P2 \textit{ is an invariant of } \mathrm{M} \quad (3.1)$$

Weakest Liberal Precondition Rules

The expression $p\ [x\backslash e]$ denotes P, where every occurrence of x is replaced by e. The expression $(a; x : e)$ denotes container a where the element at position x is replaced by the value e. The expression $p\ [a\backslash(a\ ; x : e)]$ denotes P, where every occurrence of a is replaced by $(a\ ; x : e)$ [8].
notice that:

$$(a\ ; x : e)(x) = e \tag{3.2}$$

$$(a\ ; x : e)(y) = a(y)\ \textit{if}\ x \neq y \tag{3.3}$$

Rules for finding the weakest liberal preconditions are given below [10].

$$wlp(x := e,\ p) = p\ [x\backslash e] \tag{3.4}$$

$$wlp(x :\in s,\ p) = \forall x \in s \bullet p \tag{3.5}$$

$$wlp(x :\notin s,\ p) = \forall x \in \overline{s} \bullet p \tag{3.6}$$

$$wlp(x.a := e,\ p) = p\ [a\backslash(a\ ; x : e)] \tag{3.7}$$

$$wlp(x.a :\in s,\ p) = \forall h \in s \bullet p\ [a\backslash(a\ ; x : h)] \tag{3.8}$$

$$wlp(S\ ;\ T,\ p) \Leftarrow wlp(S, wlp(T, p)) \tag{3.9}$$

$$wlp(\textbf{assert}\ b,\ p) = b \Rightarrow p \tag{3.10}$$

$$wlp(\textbf{return}\ e,\ p) = p\ [\mathit{result}\backslash e] \tag{3.11}$$

Rule (3.4) states that the weakest liberal precondition for an assignment statement $x := e$ with respect to an invariant p is equal to the invariant expression p where e is substituted into x.

Rule (3.5) states that the weakest liberal precondition for a nondeterministic assignment statement $x :\in s$ with respect to an invariant p is a universal quantification that reads: for every element in container s, the predicate p holds.

Rule (3.6) states that the weakest liberal precondition for a nondeterministic assignment statement $x :\notin s$ with respect to an invariant p is a universal quantification that reads: for every element in the complement of s, the predicate p holds.

Rule (3.7) states that the weakest liberal precondition for an assignment statement $x.a := e$ with respect to an invariant p is equal to the invariant expression p where element x at container a is replaced by the value e.

Rule (3.8) states that the weakest liberal precondition for a nondeterministic assignment statement $x.a :\in s$ with respect to an invariant p is a universal quantification that reads: when every element of s is substituted into element x at container a in predicate p, the predicate holds.

Rule (3.9) states that to find the weakest liberal precondition for a sequence of statements $S\ ;\ T$ with respect to an invariant p, we need to find the weakest liberal precondition $wlp(T, p)$ for the latest statement first. $wlp(S, wlp(T, p))$ is stronger than the weakest liberal precondition for the sequence of statements $S\ ;\ T$.

Rule (3.10) states that the weakest liberal precondition for the statement **assert** b with respect to an invariant p is simply equal to $b \Rightarrow p$.

Rule (3.11) states that the weakest liberal precondition for the statement **return** e with respect to an invariant p amounts to the invariant expression p itself.

3.4 Example

To illustrate the process introduced in this study, we apply it on a selected portion of the *Iterator* pattern where iteration is over a sequence. Both the iterator and the aggregate are represented by class *Iterator* given below. We give the translation from a class into a module and show how the module is used to prove compliance with the invariant [9].

Class Description

We start by giving the class description of that portion of the pattern.

class *Iterator*
  attr *container* : **seq of** *Object*
  attr *i* : *integer*
  initialization
    begin
      this.container := $\langle\rangle$;

```
        this.i := 0
      end
    method Next
      begin
        assert this.i < length(this.container);
        this.i := this.i + 1
      end
  end
```

Module Description

We then give the corresponding module representation.

```
  module Iterator
    var Iterator : set of Object := {}
    var container : Object → seq of Object
    var i : Object → integer
      procedure new : Object
        var this : Object •
        this :∉ Iterator ∪ {nil};
        Iterator := Iterator ∪ {this};
        this.container := ⟨⟩;
        this.i := 0
      end
    procedure Next(this : Object)
      begin
        assert this ∈ Iterator;
        assert this.i < length(this.container);
        this.i := this.i + 1
      end
  end
```

Invariant

$$(\forall k \in Iterator \bullet 0 \leq k.i \leq length(k.container))$$

Well-Definedness

We finally show how the description of *Next* preserves the invariant.

$$
\begin{array}{ll}
 & \mathit{wlp}(\mathit{Iterator.Next}, P) \\
 & \\
\equiv & \ll \textbf{definition } \mathit{of}\ \mathit{Iterator.Next} \gg \\
 & \mathit{wlp}(\textbf{assert } \mathit{this} \in \mathit{Iterator}; \textbf{assert } \mathit{this.i} < \mathit{length}(\mathit{this.container}); \mathit{this.i} := \mathit{this.i} + 1 \\
 & , (\forall k \in \mathit{Iterator} \bullet 0 \leq k.i \leq \mathit{length}(k.\mathit{container}))) \\
 & \\
\Leftarrow & \ll \textbf{wlp } \mathit{of}\ \mathit{this.i} := \mathit{this.i} + 1, \mathit{rules}\ (3.9)\ \mathit{and}\ (3.4) \gg \\
 & \mathit{wlp}(\textbf{assert } \mathit{this} \in \mathit{Iterator}; \textbf{assert } \mathit{this.i} < \mathit{length}(\mathit{this.container}) \\
 & , (\forall k \in \mathit{Iterator} \bullet 0 \leq k.i \leq \mathit{length}(k.\mathit{container}))[\mathbf{\mathit{this.i} \setminus \mathit{this.i} + 1}]) \\
 & \\
\equiv & \ll \textbf{case analysis } \mathit{with}\ k = \mathit{this}\ \mathit{and}\ k \neq \mathit{this} \gg \\
 & \mathit{wlp}(\textbf{assert } \mathit{this} \in \mathit{Iterator}; \textbf{assert } \mathit{this.i} < \mathit{length}(\mathit{this.container}) \\
 & , \quad (\forall k \in \mathbf{\mathit{Iterator} - \{\mathit{this}\}} \bullet 0 \leq k.i \leq \mathit{length}(k.\mathit{container}) \wedge \\
 & \qquad \mathbf{0 \leq \mathit{this.i} \leq \mathit{length}(\mathit{this.container})} \\
 &)[\mathbf{\mathit{this.i} \setminus \mathit{this.i} + 1}]) \\
 & \\
\equiv & \ll \textbf{substitution} \gg \\
 & \mathit{wlp}(\textbf{assert } \mathit{this} \in \mathit{Iterator}; \textbf{assert } \mathit{this.i} < \mathit{length}(\mathit{this.container}) \\
 & , \quad (\forall k \in \mathit{Iterator} - \{\mathit{this}\} \bullet 0 \leq k.i \leq \mathit{length}(k.\mathit{container}) \wedge \\
 & \qquad 0 \leq \mathbf{\mathit{this.i} + 1} \leq \mathit{length}(\mathit{this.container}))) \\
 & \\
\Leftarrow & \ll \textbf{wlp } \mathit{of}\ \mathit{the}\ \mathit{two}\ \textbf{assert}\ \mathit{statements}, \mathit{rules}\ (3.9)\ \mathit{and}\ (3.10) \gg \\
 & \qquad \mathit{this} \in \mathit{Iterator} \wedge \mathit{this.i} < \mathit{length}(\mathit{this.container}) \\
 & \Rightarrow \\
 & \qquad (\forall k \in \mathit{Iterator} - \{\mathit{this}\} \bullet 0 \leq k.i \leq \mathit{length}(k.\mathit{container}) \wedge \\
 & \qquad 0 \leq \mathbf{\mathit{this.i} + 1} \leq \mathit{length}(\mathit{this.container})) \\
 & \\
\equiv & \ll p \wedge q \Rightarrow r \wedge q \equiv p \wedge q \Rightarrow r \gg \\
 & \qquad \mathit{this} \in \mathit{Iterator} \wedge \mathit{this.i} < \mathit{length}(\mathit{this.container}) \\
 & \Rightarrow
\end{array}
$$

$(\forall k \in \mathit{Iterator} - \{\mathit{this}\} \bullet 0 \leq k.i \leq \mathit{length}(k.\mathit{container}) \wedge$
$0 \leq \mathit{this}.i + 1)$

$\Leftarrow$ $\ll$ **definition** *of* *implication*, **strengthening** $\gg$
$(\forall k \in \mathit{Iterator} - \{\mathit{this}\} \bullet 0 \leq k.i \leq \mathit{length}(k.\mathit{container}) \wedge$
$0 \leq \mathit{this}.i + 1)$

$\Leftarrow$ $\ll$ **strengthening** $\gg$
$(\forall k \in \mathit{Iterator} - \{\mathit{this}\} \bullet 0 \leq k.i \leq \mathit{length}(k.\mathit{container}) \wedge$
$0 \leq \boldsymbol{\mathit{this}.i \leq \mathit{length}(\mathit{this}.\mathit{container})})$

$\Leftarrow$

$\boldsymbol{P}$

Chapter 4

Iterator

As described in the book of *Gamma et al*. [12], the *Iterator* pattern is intended to provide a way to access elements of an aggregate sequentially without exposing its underlying representation. The pattern suggests separating the aggregate definition from traversal methods. The responsibilities to access, traverse, and keep track of the current element are placed in the iterator interface. This allows traversing elements of the aggregate in different ways without complicating the aggregate interface. The design is flexible and allows imposing constraints on which elements of the aggregate are to be visited.

The pattern involves four participants. *Iterator* defines an interface to access and traverse elements of an aggregate. *ConcreteIterator* implements the *Iterator* interface, and keeps track of the current element of the traversed aggregate. *Aggregate* defines an interface for creating an *Iterator* object. *ConcreteAggregate* implements the *Iterator* creation interface to return an instance of the proper *ConcreteIterator*.

Four methods are introduced in the interface for *Iterator*: *First* initializes the current element to the first element, *Next* advances the current element to the next element, *IsDone* checks whether we have advanced beyond the last element, and *CurrentItem* returns the current element.

The pattern is applicable whenever we need to access elements of an aggregate without exposing its internal representation. It can also provide a way to traverse objects of the same aggregate multiple times. Another application for the pattern is when we need a uniform interface for traversing different aggregate structures.

An iterator object is constructed around a specific aggregate object, and the two objects

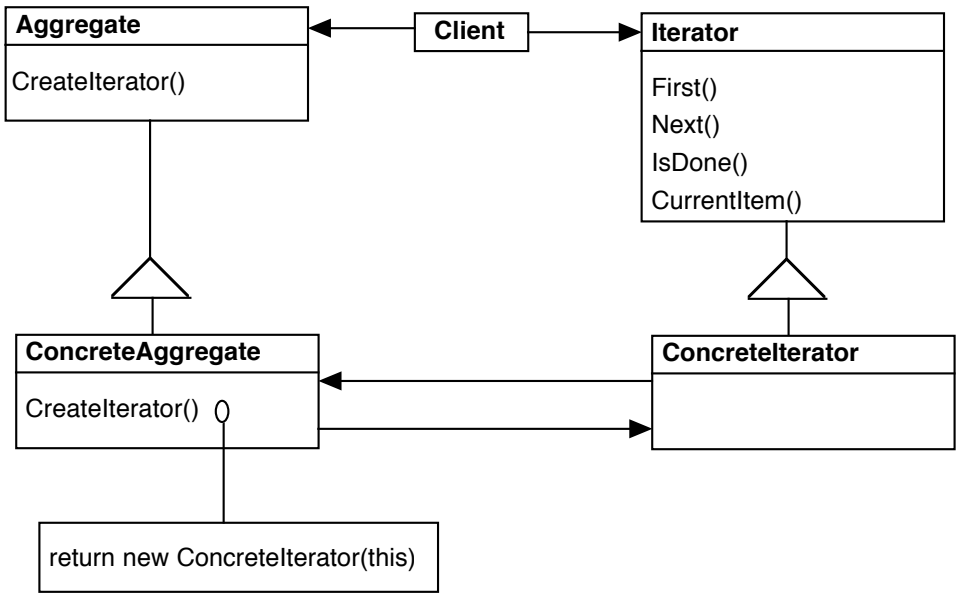

Figure 4.1: Iterator Class Diagram

are coupled. An iterator object is returned by method *CreateIterator* in the aggregate object. Apart from *CreateIterator*, other methods of an aggregate are not specified by the pattern. However, those methods are required to invalidate all associated iterators in case of adding or removing elements. Invalidating an iterator is represented by setting a boolean flag to false.

In our study, we relax the sequential access requirement to include unordered aggregates. An unordered aggregate is the basis to generate the most basic form of the pattern. An ordered aggregate is used to generate a special case of the pattern and is considered to be a refinement of the basic form. Another refinement of the basic form is the case when there is a criteria to select elements visited rather than visiting all elements.

Iterator is one of the richest invariant-based patterns. Unlike most of design patterns, it requires a considerable amount of behavioral description for its methods. Therefore, *Iterator* is selected to illustrate the complete analysis process introduced in this study.

As shown earlier, a pattern is introduced as a module. A module is a representation of one or more classes. In this chapter, we give the formal description of classes involved in the pattern including special cases. We also give the module variables and invariant. Methods of a class are translated to module procedures only within the formal proofs. In the following chapters, we give the formal proofs of the pattern description well-definedness and show the compliance of a selected instance with the pattern.

4.1 The Pattern

Below we give the formal description for the *Iterator* design pattern. As mentioned earlier, it is based on the description in the book of *Gamma et al*. [12]. The description is refined by analyzing instances that implement the pattern.

```
class Aggregate
  method CreateIterator : Iterator
end
```

```
class ConcreteAggregate implements Aggregate
  attr cont : bag of Object
  initialization
    this.cont := []
  method CreateIterator : Iterator
    var c : Object •
    begin
      c := new ConcreteIterator;
      return c
    end
end
```

```
class Iterator
  method First
  method Next
  method IsDone : boolean
  method CurrentItem : Object
end


class ConcreteIterator implements Iterator
  attr valid : boolean
  attr aggregate : Aggregate
  attr container : bag of Object
  attr visited : bag of Object
  attr current : Object
  initialization (agg : Aggregate)
    begin
      assert this.container ≠ [];
      this.valid := true;
      this.aggregate := agg;
      this.container := agg.cont;
      this.visited := [];
      this.current :∈ this.container
    end
  method First
    begin
      assert this.valid;
      this.visited := [];
      this.current :∈ this.container
    end
  method Next
    begin
```

```
        assert this.valid;
        assert this.visited ≠ this.container;
        this.visited := this.visited ∪ [this.current];
        this.current :∈ this.container − this.visited
      end
    method IsDone : boolean
      begin
        assert this.valid;
        return this.visited = this.container
      end
    method CurrentItem : Object
      begin
        assert this.valid;
        return this.current
      end
end


method Client
    var ag : Aggregate, iter : Iterator •
  begin
    ag := new ConcreteAggregate;
    iter := ag.CreateIterator;
    iter.Next
  end
```

4.2 Data Structures

```
var Iterator, Aggregate : set of Object := {}, {}
var cont : Object → bag of Object
var valid : Object → boolean
var aggregate : Object → Object
```

var *container* : *Object* $\rightarrow$ **bag of** *Object*
var *visited* : *Object* $\rightarrow$ **bag of** *Object*
var *current* : *Object* $\rightarrow$ *Object*

4.3 Pattern Invariant

Invariant *P*

$$(\forall i \in \mathit{ConcreteIterator} \bullet i.\mathit{container} = i.\mathit{aggregate}.\mathit{cont} \vee \neg\, i.\mathit{valid}) \wedge$$
$$(\forall i \in \mathit{ConcreteIterator} \bullet i.\mathit{visited} \subseteq i.\mathit{container}) \wedge$$
$$(\forall i \in \mathit{ConcreteIterator} \bullet i.\mathit{current} \in i.\mathit{container} - i.\mathit{visited} \vee i.\mathit{container} = i.\mathit{visited})$$

The above invariant reads as follows:

- The elements stored in the iterator container should be the same as those in the associated aggregate or the iterator is invalid.
- Visited elements need to belong to the iterator container.
- The current element in the iterator is an element of the container of the iterator which was not visited before, or we are at the end of the iteration.

To maintain the elements condition, an iterator may need to:

- Maintain a pointer to the underlying aggregate;
- Maintain a copy of the elements in the aggregate container.

We compare the stored copy of container elements with the original elements through the pointer to the underlying aggregate.

4.4 Special Cases

To formalize a pattern, we need to make a few assumptions about the expected behavior of methods of the pattern. An assumption used in the above formalization of *Iterator*

pattern is that the underlying data structure associated with the iterator object behaves as a *bag of objects*. Another assumption is that all elements of the data structure have to be visited. Below we introduce two special cases:

Sequential Iteration

SeqIterator and *SeqAggregate* refine *ConcreteIterator* and *ConcreteAggregate* respectively. *SeqIterator* iterates over an ordered container represent by *SeqAggregate*. The underlying data structure *cont* is represented abstractly with a *sequence* rather than a *bag*.

```
class SeqAggregate implements Aggregate
  attr cont : seq of Object
  initialization
    this.cont := ⟨⟩
  method CreateIterator : Iterator
    return new SeqIterator(this)
end


class SeqIterator implements Iterator
  attr valid : boolean
  attr aggregate : SeqAggregate
  attr container : seq of Object
  attr i : integer
  initialization (agg : SeqAggregate)
    begin
      this.valid := true;
      this.aggregate := agg;
      this.container := agg.cont;
      this.i := 0
    end
  method First
    begin
```

```
        assert this.valid;
        this.i := 0
      end
    method Next
      begin
        assert this.valid;
        assert i < length(this.container);
        this.i := this.i + 1
      end
    method IsDone : boolean
      begin
        assert this.valid;
        return this.i ≥ length(this.container)
      end
    method CurrentItem : Object
      begin
        assert this.valid;
        return this.container[i]
      end
  end
```

Selective Sequential Iteration

SelectSeqIterator refines *SeqIterator*. *SeqIterator* also iterates over an ordered container represent by *SeqAggregate*. *SelectSeqIterator* will however be able to impose a criteria on which elements are to be visited rather than visiting all of them.

```
  class SelectSeqIterator inherits SeqIterator
    method Next
        var j : integer •
      begin
        assert this.container = this.aggregate.cont;
        assert this.i < length(this.container);
        j := this.i;
        this.i :∈ {x | x ≥ j}
```

end
end

Special Cases Data Structures

var *SeqIterator*, *SeqAggregate* : **set of** *Object* := {}, {}
var *aggregate* : *SeqIterator* → *SeqAggregate*
var *container* : *SeqIterator* → **seq of** *Object*

Note that:

SeqIterator ⊆ *Iterator* and *SeqAggregate* ⊆ *Aggregate*

Special Cases Invariant *PA*
The two introduced special cases share the following invariant:

$$(\forall k \in \mathit{SeqIterator} \bullet k.\mathit{container} = k.\mathit{aggregate}.\mathit{cont} \vee \neg\, k.\mathit{valid}) \wedge$$
$$(\forall k \in \mathit{SeqIterator} \bullet 0 \leq k.i \leq \mathit{length}(k.\mathit{container})$$

The above invariant reads as follows:

- The elements stored in the iterator container should be the same as those in the associated aggregate or the iterator is invalid.
- The valid range for attribute *this*.*i* which represents the pointer to the current element at *container* is between 0 and *length*(*this*.*container*).

Chapter 5

Well-Definedness of the Pattern

The rules for a module to preserve an invariant are given in an earlier chapter. A module initialization is required to establish the invariant. Class methods and initializations are only required to preserve the invariant. *ConcreteIterator* is abbreviated to *CI* throughout this chapter.

5.1 Module Initialization Establishes Invariant

$$wlp(Iterator := \{\} \wedge Aggregate := \{\}\ , P)$$

$$\begin{aligned}
\equiv\ & (\ (\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\ i.valid) \wedge \\
& (\forall i \in CI \bullet i.visited \subseteq i.container) \wedge \\
& (\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \\
&)\ [\boldsymbol{CI}, \boldsymbol{Aggregate} \setminus \{\ \}, \{\ \}]
\end{aligned}$$

$$\begin{aligned}
\equiv\ & \ll \textbf{substitution}, \textit{replace every occurrence of Iterator with the value } [\ ] \gg \\
& (\forall i \in \{\ \} \bullet i.container = i.aggregate.cont \vee \neg\ i.valid) \wedge \\
& (\forall i \in \{\ \} \bullet i.visited \subseteq i.container) \wedge \\
& (\forall i \in \{\ \} \bullet i.current \in i.container - i.visited \vee i.container = i.visited)
\end{aligned}$$

$$\equiv\ \ll \textbf{logic}, \textit{universal quantification over empty range} \gg$$

true

5.2 ConcreteIterator Initialization Preserves Invariant

According to the relationship between classes and modules given earlier, the statements $this :\notin CI \cup \{nil\}\ ;\ CI := CI \cup \{this\}$ are added to *CI* initialization.

$wlp(CI.new, P)$

$\equiv$ $\ll$ **definition** *of* $CI.new \gg$
$wlp($*this* $:\notin CI \cup \{nil\};$
CI $:= CI \cup \{this\};$
assert *this.container* $\neq [];$
this.valid $:=$ *true*;
this.aggregate $:=$ *agg*;
this.container $:=$ *agg.cont*;
this.visited $:= [\,];$
this.current $:\in$ *this.container*
$, P)$

$\Leftarrow$ $\ll$ **wlp** *of* *this.current* $:\in$ *this.container*, *rules* (3.8) *and* $(3.9) \gg$
$wlp($*this* $:\notin CI \cup \{nil\}; CI := CI \cup \{this\};$
assert *this.container* $\neq [];$ *this.valid* $:=$ *true*; *this.aggregate* $:=$ *agg*;
this.container $:=$ *agg.cont*; *this.visited* $:= [\,]$
$, \forall h \in$ ***this.container*** $\bullet$ **(**
$(\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited)$
$)$ **[*current* \ (*current*; *this* : *h*)])**

$\Leftarrow$ $\ll$ **wlp** *of* *this.visited* $:= [\,]$, *rules* (3.7) *and* $(3.9) \gg$
$wlp($*this* $:\notin CI \cup \{nil\}; CI := CI \cup \{this\};$

$$
\begin{array}{l}
\quad \textbf{assert}\ this.container \neq [];\ this.valid := true;\ this.aggregate := agg; \\
\quad this.container := agg.cont \\
\quad ,\forall h \in this.container \bullet (\\
\qquad (\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
\qquad (\forall i \in CI \bullet i.visited \subseteq i.container) \wedge \\
\qquad (\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \\
\quad) \ [current \setminus (current; this : h)]\ \mathbf{[visited \setminus (visited; this : [\,])]}\ ) \\
\\
\equiv \quad \ll \textbf{case analysis}\ with\ i = \ this\ and\ i \neq this \gg \\
\quad wlp(this :\notin CI \cup \{nil\}; CI := CI \cup \{this\}; \\
\quad \textbf{assert}\ this.container \neq [];\ this.valid := true;\ this.aggregate := agg; \\
\quad this.container := agg.cont \\
\quad ,\forall h \in this.container \bullet (\\
\qquad (\forall i \in \mathbf{CI - \{this\}} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
\qquad (\forall i \in \mathbf{CI - \{this\}} \bullet i.visited \subseteq i.container) \wedge \\
\qquad (\forall i \in \mathbf{CI - \{this\}} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge \\
\qquad \mathbf{(this.container = this.aggregate.cont \vee \neg\, this.valid)} \wedge \\
\qquad \mathbf{(this.visited \subseteq this.container)} \wedge \\
\qquad \mathbf{(this.current \in this.container - this.visited \vee this.container = this.visited)} \\
\quad) \ [current \setminus (current; this : h)]\ [visited \setminus (visited; this : [\,])]\ ) \\
\\
\equiv \quad \ll \textbf{substitution} \gg \\
\quad wlp(this :\notin CI \cup \{nil\}; CI := CI \cup \{this\}; \\
\quad \textbf{assert}\ this.container \neq [];\ this.valid := true;\ this.aggregate := agg; \\
\quad this.container := agg.cont \\
\quad ,\forall h \in this.container \bullet (\\
\qquad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
\qquad (\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge \\
\qquad (\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge \\
\qquad (this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge \\
\qquad (this.\mathbf{(visited; this : [\,])} \subseteq this.container) \wedge \\
\qquad (this.current \in this.container - this.\mathbf{(visited; this : [\,])} \vee
\end{array}
$$

$$
\begin{array}{l}
\quad\quad this.container = this.(\textbf{\textit{visited}}; \textbf{\textit{this}} : [\,]))\\
\quad) \; [current \setminus (current; this : h)] \;)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{simplification}, (f : a; b)(a) \equiv b, \ \textit{therefore}, (visited; this : [\,])(this) \equiv [\,] \gg \\
& wlp(this :\notin CI \cup \{nil\}; CI := CI \cup \{this\};\\
& \textbf{assert } this.container \neq []; this.valid := true; this.aggregate := agg;\\
& this.container := agg.cont\\
& , \forall h \in this.container \bullet (\\
& \quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge\\
& \quad (\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge\\
& \quad (\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee\\
& \quad i.container = i.visited) \wedge\\
& \quad (this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge\\
& \quad ([\,] \subseteq this.container) \wedge\\
& \quad (this.current \in this.container - [\,] \vee this.container = [\,])\\
&) \; [current \setminus (current; this : h)] \;)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{logic}, [\,] \subseteq this.container \ \gg \\
& wlp(this :\notin CI \cup \{nil\}; CI := CI \cup \{this\};\\
& \textbf{assert } this.container \neq []; this.valid := true; this.aggregate := agg;\\
& this.container := agg.cont\\
& , \forall h \in this.container \bullet (\\
& \quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge\\
& \quad (\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge\\
& \quad (\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited\\
& \quad \vee i.container = i.visited) \wedge\\
& \quad (this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge\\
& \quad \textbf{\textit{true}} \wedge\\
& \quad (this.current \in this.container - [\,] \vee this.container = [\,])\\
&) \; [current \setminus (current; this : h)] \;)
\end{array}
$$

$$
\equiv \quad \ll \textbf{substitution}, \textbf{logic}, \ \textit{all elements of } this.container \textit{ are in } this.container \gg
$$

$wlp(this :\notin CI \cup \{nil\}; CI := CI \cup \{this\};$
assert $this.container \neq []; this.valid := true; this.aggregate := agg;$
$this.container := agg.cont$
, $(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge$
$(this.container = this.aggregate.cont \vee \neg\, this.valid) \ \wedge true \wedge \mathbf{true})$)

$\equiv$ $\ll$ **logic**, *removed* $true \wedge true$ $\gg$
$wlp(this :\notin CI \cup \{nil\}; CI := CI \cup \{this\};$
assert $this.container \neq []; this.valid := true; this.aggregate := agg;$
$this.container := agg.cont$
, $(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge$
$(this.container = this.aggregate.cont \vee \neg\, this.valid)$)

$\Leftarrow$ $\ll$ **wlp** *of the last three assignment statements, rules* (3.7) *and* (3.9) $\gg$
$wlp(this :\notin CI \cup \{nil\}; CI := CI \cup \{this\};$ **assert** $this.container \neq []$
, ($(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge$
$(this.container = this.aggregate.cont \vee \neg\, this.valid)$
) $[\mathbf{container} \setminus (\mathbf{container}; \mathbf{this} : \mathbf{agg.cont})]$ $[\mathbf{aggregate} \setminus (\mathbf{aggregate}; \mathbf{this} : \mathbf{agg})]$
$[\mathbf{valid} \setminus (\mathbf{valid}; \mathbf{this} : \mathbf{true})]$)

$\equiv$ $\ll$ **substitution** $\gg$
$wlp(this :\notin CI \cup \{nil\}; CI := CI \cup \{this\};$ **assert** $this.container \neq []$
, $(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge$
$(\mathbf{agg.cont = agg.cont} \vee \mathbf{false})$)

$\equiv$ $\ll$ **logic**, *removed last predicate* $\gg$
$wlp(this :\notin CI \cup \{nil\}; CI := CI \cup \{this\};$ **assert** $this.container \neq [])$
$, \quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited)\ )$

$\Leftarrow$ $\ll$ **wlp** *of* $CI := CI \cup \{this\}$, *rules* (3.4) *and* (3.9) $\gg$
$wlp(this :\notin CI \cup \{nil\};$ **assert** $this.container \neq [])$
$, \quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited)$
$)\ [\boldsymbol{CI \setminus CI \cup \{this\}}]$

$\equiv$ $\ll$ **substitution**, $CI \cup \{this\} - \{this\} \equiv CI$ $\gg$
$wlp(this :\notin CI \cup \{nil\};$ **assert** $this.container \neq [])$
$, \quad (\forall i \in \boldsymbol{CI} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in \boldsymbol{CI} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in \boldsymbol{CI} \bullet i.current \in i.container - i.visited \vee i.container = i.visited)\ )$

$\Leftarrow$ $\ll$ **wlp** *of* **assert** $this.container \neq []$, *rules* (3.10) *and* (3.9) $\gg$
$wlp(this :\notin CI \cup \{nil\}$
$, \quad (\boldsymbol{this.container \neq [] \Rightarrow}$
$(\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited)\ )\ )$

$\Leftarrow$ $\ll$ **definition** *of implication*, **strengthening** $\gg$
$wlp(this :\notin CI \cup \{nil\}$
$, \quad (\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited)\ )$

$\equiv$ $\ll$ **wlp** *of this* $:\notin CI \cup \{nil\}$, *rule* (3.6) $\gg$
$\forall h \in \overline{CI} \bullet ($
$(\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited)\ )$

$\equiv$ $\ll$ "*h*" *does not appear in predicate* $\gg$
$(\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited)$

$\Leftarrow$
P

5.3 First Preserves Invariant

The statement **assert** *this* $\in$ *CI* is added to *First* due to the translation from a method to a procedure.

$wlp(CI.First, P)$

$\equiv$ $\ll$ **definition** *of* $CI.First$ $\gg$
$wlp($**assert** $this \in CI$; **assert** $this.valid$;
$this.visited := [\,]; this.current :\in this.container$
$, P)$

$\Leftarrow$ $\ll$ **wlp** *of* $this.current :\in this.container$, *rules* (3.8) *and* (3.9) $\gg$
$wlp($**assert** $this \in CI$; **assert** $this.valid$; $this.visited := [\,]$
$, \forall h \in this.container \bullet ($
$(\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited)$

$$) \ [\textbf{\textit{current}} \setminus (\textbf{\textit{current}}; \textbf{\textit{this}} : \textbf{\textit{h}})] \ )$$

$\Leftarrow$ $\ll$ **wlp** *of* $this.visited := [\,]$, *rules* (3.7) *and* (3.9) $\gg$

$wlp($**assert** $this \in CI$; **assert** $this.valid$

$, \forall h \in this.container \bullet ($

$(\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$

$(\forall i \in CI \bullet i.visited \subseteq i.container) \wedge$

$(\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited)$

$) \ [current \setminus (current; this : h)] \ [\textbf{\textit{visited}} \setminus (\textbf{\textit{visited}}; \textbf{\textit{this}} : [\,])] \ )$

$\equiv$ $\ll$ **case analysis** *with* $i = this$ *and* $i \neq this$ $\gg$

$wlp($**assert** $this \in CI$; **assert** $this.valid$

$, \forall h \in this.container \bullet ($

$(\forall i \in \textbf{\textit{CI}} - \{\textbf{\textit{this}}\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$

$(\forall i \in \textbf{\textit{CI}} - \{\textbf{\textit{this}}\} \bullet i.visited \subseteq i.container) \wedge$

$(\forall i \in \textbf{\textit{CI}} - \{\textbf{\textit{this}}\} \bullet i.current \in i.container - i.visited$

$\vee i.container = i.visited) \wedge$

$(\textbf{\textit{this}}.container = \textbf{\textit{this}}.aggregate.cont \vee \neg\, \textbf{\textit{this}}.valid) \wedge$

$(\textbf{\textit{this}}.visited \subseteq \textbf{\textit{this}}.container) \wedge$

$(\textbf{\textit{this}}.current \in \textbf{\textit{this}}.container - \textbf{\textit{this}}.visited \vee \textbf{\textit{this}}.container = \textbf{\textit{this}}.visited)$

$) \ [current \setminus (current; this : h)] \ [visited \setminus (visited; this : [\,])] \ )$

$\equiv$ $\ll$ **substitution** $\gg$

$wlp($**assert** $this \in CI$; **assert** $this.valid$

$, \forall h \in this.container \bullet ($

$(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$

$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$

$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited$

$\vee i.container = i.visited) \wedge$

$(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge$

$(this.(\textbf{\textit{visited}}; \textbf{\textit{this}} : [\,]) \subseteq this.container) \wedge$

$(this.current \in this.container - this.(\textbf{\textit{visited}}; \textbf{\textit{this}} : [\,]) \vee$

$$
\begin{array}{l}
\quad\quad this.container = this.(\textbf{\textit{visited}}; \textbf{\textit{this}} : [\,])) \\
\quad) \; [current \setminus (current; this : h)] \;)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{simplification}, (f : a; b)(a) \equiv b, rule\ (3.2) \gg \\
& wlp(\textbf{assert}\ this \in CI; \textbf{assert}\ this.valid \\
& , \forall h \in this.container \bullet (\\
& \quad\quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
& \quad\quad (\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge \\
& \quad\quad (\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \\
& \quad\quad \vee i.container = i.visited) \wedge \\
& \quad\quad (this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge \\
& \quad\quad ([\,] \subseteq this.container) \wedge \\
& \quad\quad (this.current \in this.container - [\,] \vee this.container = [\,]) \\
&) \; [current \setminus (current; this : h)] \;)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{logic} \gg \\
& wlp(\textbf{assert}\ this \in CI; \textbf{assert}\ this.valid \\
& , \quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
& (\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge \\
& (\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \\
& \quad\quad \vee i.container = i.visited) \wedge \\
& (this.container = this.aggregate.cont \vee \neg\, this.valid) \;)
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \ll \textbf{wlp}\ of\ \textbf{assert}\ this.valid, rules\ (3.10)\ and\ (3.9) \gg \\
& wlp(\textbf{assert}\ this \in CI \\
& , \quad this.valid \\
& \Rightarrow \\
& \quad\quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
& \quad\quad (\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge \\
& \quad\quad (\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \\
& \quad\quad \vee i.container = i.visited) \wedge \\
& \quad\quad (this.container = this.aggregate.cont \vee \neg\, this.valid) \;)
\end{array}
$$

$\equiv$ $\ll$ *merging the last predicate* $\gg$
wlp(**assert** *this* $\in$ *CI*
, *this.valid*
$\Rightarrow$
$(\forall i \in \boldsymbol{CI} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited$
$\vee i.container = i.visited)$)

$\equiv$ $\ll$ **wlp** *of* **assert** *this* $\in$ *CI*, *rule* (3.10) $\gg$
this $\in$ *CI*
$\Rightarrow$
this.valid
$\Rightarrow$
$(\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited$
$\vee i.container = i.visited)$

$\Leftarrow$ $\ll$ **definition** *of implication*, **strengthening** $\gg$
$(\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited)$

$\Leftarrow$ $\ll$ **logic**, $(\forall i \in X - \{a\}) \Leftarrow (\forall i \in X - \{a\} \wedge a) \equiv (\forall i \in X)$ $\gg$
$\ll$ *predicates having* $CI - \{this\}$ *are weaker than those having Iterator* $\gg$
P

5.4 Next Preserves Invariant

The statement **assert** *this* $\in$ *CI* is added to *Next* due to the translation from a method to a procedure.

$$
\begin{array}{ll}
 & wlp(CI.Next, P) \\
 & \\
\equiv & \ll \textbf{definition } of\ CI.Next \gg \\
 & wlp(\textbf{assert } this \in CI; \textbf{assert } this.valid; \\
 & \quad \textbf{assert } this.visited \neq this.container; \\
 & \quad this.visited := this.visited \cup [this.current]; \\
 & \quad this.current :\in this.container - this.visited \\
 & , P) \\
 & \\
\Leftarrow & \ll \textbf{wlp } of\ this.current :\in this.container - this.visited, rules\ (3.8)\ and\ (3.9) \gg \\
 & wlp(\textbf{assert } this \in CI; \textbf{assert } this.valid; \\
 & \quad \textbf{assert } this.visited \neq this.container; \\
 & \quad this.visited := this.visited \cup [this.current] \\
 & , \forall h \in this.container - this.visited \bullet (\\
 & \quad (\forall i \in CI \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
 & \quad (\forall i \in CI \bullet i.visited \subseteq i.container) \wedge \\
 & \quad (\forall i \in CI \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \\
 &) [current \setminus (current; this : h)]) \\
 & \\
\equiv & \ll \textbf{case analysis } with\ i = this\ and\ i \neq this \gg \\
 & wlp(\textbf{assert } this \in CI; \textbf{assert } this.valid; \\
 & \quad \textbf{assert } this.visited \neq this.container; \\
 & \quad this.visited := this.visited \cup [this.current] \\
 & , \forall h \in this.container - this.visited \bullet (\\
 & \quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
 & \quad (\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge \\
 & \quad (\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \\
 & \quad \vee i.container = i.visited) \wedge
\end{array}
$$

$$
\begin{array}{l}
\quad (\textbf{\textit{this.container}} = \textbf{\textit{this.aggregate.cont}} \vee \neg\, \textbf{\textit{this.valid}}) \wedge \\
\quad (\textbf{\textit{this.visited}} \subseteq \textbf{\textit{this.container}}) \wedge \\
\quad (\textbf{\textit{this.current}} \in \textbf{\textit{this.container}} - \textbf{\textit{this.visited}} \vee \textbf{\textit{this.container}} = \textbf{\textit{this.visited}}) \\
) \; [current \setminus (current; this : h)] \;)
\end{array}
$$

$\equiv$ $\ll$ **substitution**, **logic**, *quantification* $\gg$

$$
\begin{array}{l}
wlp(\textbf{assert}\ this \in CI;\ \textbf{assert}\ this.valid; \\
\quad \textbf{assert}\ this.visited \neq this.container; \\
\quad this.visited := this.visited \cup [this.current] \\
,\quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge \\
(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge \\
(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge \\
(this.visited \subseteq this.container) \wedge \\
\textbf{\textit{true}})
\end{array}
$$

$\Leftarrow$ $\ll$ **wlp** *of* $this.visited := this.visited \cup [this.current]$, *rules* (3.7) *and* (3.9) $\gg$

$$
\begin{array}{l}
wlp(\textbf{assert}\ this \in CI;\ \textbf{assert}\ this.valid;\ \textbf{assert}\ this.visited \neq this.container \\
,\quad ((\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge \\
(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge \\
(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge \\
(this.visited \subseteq this.container) \\
)[\textbf{\textit{visited}} \setminus (\textbf{\textit{visited}}\ ; \textbf{\textit{this}} :\ \textbf{\textit{this.visited}} \cup [\textbf{\textit{this.current}}]\ )] \;)
\end{array}
$$

$\equiv$ $\ll$ **substitution**, **simplification** $\gg$

$$
\begin{array}{l}
wlp(\textbf{assert}\ this \in CI;\ \textbf{assert}\ this.valid;\ \textbf{assert}\ this.visited \neq this.container \\
,\quad (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge \\
(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge \\
(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge \\
(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge \\
(\textbf{\textit{this.visited}} \cup [\textbf{\textit{this.current}}] \subseteq this.container) \;)
\end{array}
$$

$\equiv$ $\ll$ **logic**, $X \cup \{a\} \subseteq Y \equiv X \subseteq Y \wedge a \in Y$, *split this.visited* $\cup$ $[$*this.current*$] \gg$
wlp(**assert** *this* $\in$ *CI*; **assert** *this.valid*; **assert** *this.visited* $\neq$ *this.container*
, $(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge$
$(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge$
$(\mathbf{this.visited} \subseteq this.container) \wedge$
$(\mathbf{this.current} \in this.container)$)

$\Leftarrow$ $\ll$ **wlp** *of* **assert** *this.visited* $\neq$ *this.container*, *rules* (3.10) *and* $(3.9) \gg$
wlp(**assert** *this* $\in$ *CI*; **assert** *this.valid*
, $this.visited \neq this.container$
$\Rightarrow$
$(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited$
$\vee i.container = i.visited) \wedge$
$(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge$
$(this.visited \subseteq this.container) \wedge$
$(this.current \in this.container)$)

$\Leftarrow$ $\ll$ **wlp** *of* **assert** *this.valid*, *rule* $(3.10) \gg$
wlp(**assert** *this* $\in$ *CI*
, *this.valid*
$\Rightarrow$
$this.visited \neq this.container$
$\Rightarrow$
$(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited$
$\vee i.container = i.visited) \wedge$

$$
\begin{array}{l}
\quad (\mathit{this.container} = \mathit{this.aggregate.cont} \vee \neg\, \mathit{this.valid}) \wedge \\
\quad (\mathit{this.visited} \subseteq \mathit{this.container}) \wedge \\
\quad (\mathit{this.current} \in \mathit{this.container})\ )
\end{array}
$$

$\equiv$ $\ll$ **wlp** *of* **assert** $\mathit{this} \in \mathit{CI}, \mathit{rule}\ (3.10) \gg$

$\mathit{this} \in \mathit{CI}$

$\Rightarrow$

$\mathit{this.valid}$

$\Rightarrow$

$\mathit{this.visited} \neq \mathit{this.container}$

$\Rightarrow$

$$
\begin{array}{l}
(\forall i \in \mathit{CI} - \{\mathit{this}\} \bullet i.\mathit{container} = i.\mathit{aggregate.cont} \vee \neg\, i.\mathit{valid}) \wedge \\
(\forall i \in \mathit{CI} - \{\mathit{this}\} \bullet i.\mathit{visited} \subseteq i.\mathit{container}) \wedge \\
(\forall i \in \mathit{CI} - \{\mathit{this}\} \bullet i.\mathit{current} \in i.\mathit{container} - i.\mathit{visited} \\
\vee i.\mathit{container} = i.\mathit{visited}) \wedge \\
(\mathit{this.container} = \mathit{this.aggregate.cont} \vee \neg\, \mathit{this.valid}) \wedge \\
(\mathit{this.visited} \subseteq \mathit{this.container}) \wedge \\
(\mathit{this.current} \in \mathit{this.container})
\end{array}
$$

$\Leftarrow$ $\ll$ **definition** *of implication*, **strengthening**, *removed first two predicates* $\gg$

$\mathit{this.visited} \neq \mathit{this.container}$

$\Rightarrow$

$$
\begin{array}{l}
(\forall i \in \mathit{CI} - \{\mathit{this}\} \bullet i.\mathit{container} = i.\mathit{aggregate.cont} \vee \neg\, i.\mathit{valid}) \wedge \\
(\forall i \in \mathit{CI} - \{\mathit{this}\} \bullet i.\mathit{visited} \subseteq i.\mathit{container}) \wedge \\
(\forall i \in \mathit{CI} - \{\mathit{this}\} \bullet i.\mathit{current} \in i.\mathit{container} - i.\mathit{visited} \\
\vee i.\mathit{container} = i.\mathit{visited}) \wedge \\
(\mathit{this.container} = \mathit{this.aggregate.cont} \vee \neg\, \mathit{this.valid}) \wedge \\
(\mathit{this.visited} \subseteq \mathit{this.container}) \wedge \\
(\mathit{this.current} \in \mathit{this.container})
\end{array}
$$

$\equiv$ $\ll$ **definition** *of implication* $\gg$

$\mathit{this.visited} = \mathit{this.container}$

$\vee$

$(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee i.container = i.visited) \wedge$
$(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge$
$(this.visited \subseteq this.container) \wedge$
$(this.current \in this.container)$

$\equiv$ $\ll$ **distributivity** *of* $\vee$ *over* $\wedge$ $\gg$
$\mathbf{this.visited = this.container} \vee (\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont$
$\vee \neg\, i.valid) \wedge$
$\mathbf{this.visited = this.container} \vee (\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$\mathbf{this.visited = this.container} \vee (\forall i \in CI - \{this\} \bullet$
$i.current \in i.container - i.visited$
$\vee i.container = i.visited) \wedge$
$\mathbf{this.visited = this.container} \vee (this.container = this.aggregate.cont$
$\vee \neg\, this.valid) \wedge$
$\mathbf{this.visited = this.container} \vee (this.visited \subseteq this.container) \wedge$
$\mathbf{this.visited = this.container} \vee (this.current \in this.container)$

$\Leftarrow$ $\ll$ **strengthening**, *removed this.visited = this.container from predicates* $\gg$
$(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$
$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee$
$i.container = i.visited) \wedge$
$(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge$
$(this.visited \subseteq this.container) \wedge$
$\mathbf{this.visited = this.container} \vee (this.current \in this.container)$

$\Leftarrow$ $\ll$ **strengthening**, $a \in X \Leftarrow a \in X - Y$ $\gg$
$\ll$ *replace this.container with this.container − this.visited* $\gg$
$(\forall i \in CI - \{this\} \bullet i.container = i.aggregate.cont \vee \neg\, i.valid) \wedge$

$(\forall i \in CI - \{this\} \bullet i.visited \subseteq i.container) \wedge$
$(\forall i \in CI - \{this\} \bullet i.current \in i.container - i.visited \vee$
$i.container = i.visited) \wedge$
$(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge$
$(this.visited \subseteq this.container) \wedge$
$(this.current \in this.container - \boldsymbol{this.visited}) \vee this.visited = this.container$

$\equiv$ $\ll CI = CI - \{this\} \cup \{this\} \gg$
$\boldsymbol{P}$

5.5 IsDone Preserves Invariant

The statement **assert** *this* $\in$ *CI* is added to *IsDone* due to the translation from a method to a procedure.

$wlp(CI.IsDone, P)$

$\equiv$ $\ll$ **definition** *of* $CI.IsDone \gg$
$wlp($**assert** $this \in CI$; **assert** $this.valid$; **return** $this.visited = this.container$
$, P)$

$\Leftarrow$ $\ll$ **wlp** *of* **return** $visited = container$, *rules* (3.9) *and* $(3.11) \gg$
$wlp($**assert** $this \in CI$; **assert** $this.valid$
$, P)$

$\Leftarrow$ $\ll$ **wlp** *of* **assert** $this.valid$, *rule* $(3.10) \gg$
$wlp($**assert** $this \in CI$
, $this.valid$
$\Rightarrow$
$(this.container = this.aggregate.cont \vee \neg\, this.valid) \wedge$
$(this.visited \subseteq this.container) \wedge$
$(this.current \in this.container - this.visited \vee this.container = this.visited)$)

$\equiv$ $\ll$ **wlp** *of* **assert** *this* $\in$ *CI*, *rule* (3.10) $\gg$
this $\in$ *CI*
$\Rightarrow$
this.valid
$\Rightarrow$
$(\mathit{this.container} = \mathit{this.aggregate.cont} \vee \neg\, \mathit{this.valid}) \wedge$
$(\mathit{this.visited} \subseteq \mathit{this.container}) \wedge$
$(\mathit{this.current} \in \mathit{this.container} - \mathit{this.visited} \vee \mathit{this.container} = \mathit{this.visited})$

$\Leftarrow$ $\ll$ **definition** *of* *implication*, **strengthening** $\gg$
P

5.6 CurrentItem Preserves Invariant

The statement **assert** *this* $\in$ *CI* is added to *CurrentItem* due to the translation from a method to a procedure.

$wlp(\mathit{CI.CurrentItem}, P)$

$\equiv$ $\ll$ **definition** *of* *CI.CurrentItem* $\gg$
wlp(**assert** *this* $\in$ *CI*; **assert** *this.valid*; **return** *this.current*
, *P*)

$\Leftarrow$ $\ll$ **wlp** *of* **return** *this.current*, *rules* (3.9) *and* (3.11) $\gg$
wlp(**assert** *this* $\in$ *CI*; **assert** *this.valid*
, *P*)

$\equiv$ $\ll$ **wlp** *of* **assert** *this.valid*, *rule* (3.10) $\gg$
wlp(**assert** *this* $\in$ *CI*
, *this.valid*
$\Rightarrow$

$(\mathit{this.container} = \mathit{this.aggregate.cont} \vee \neg\, \mathit{this.valid}) \wedge$
$(\mathit{this.visited} \subseteq \mathit{this.container}) \wedge$
$(\mathit{this.current} \in \mathit{this.container} - \mathit{this.visited} \vee \mathit{this.container} = \mathit{this.visited}))$

$\equiv$ $\ll$ **wlp** *of* **assert** $\mathit{this} \in \mathit{CI}, \mathit{rule}\ (3.10) \gg$
$\mathit{this} \in \mathit{CI}$
$\Rightarrow$
$\mathit{this.valid}$
$\Rightarrow$
$(\mathit{this.container} = \mathit{this.aggregate.cont} \vee \neg\, \mathit{this.valid}) \wedge$
$(\mathit{this.visited} \subseteq \mathit{this.container}) \wedge$
$(\mathit{this.current} \in \mathit{this.container} - \mathit{this.visited} \vee$
$\mathit{this.container} = \mathit{this.visited})$

$\Leftarrow$ $\ll$ **definition** *of* *implication*, **strengthening** $\gg$
$\boldsymbol{P}$

Chapter 6

Instance Compliance with the Pattern

The instance given is a simple program found in the C# documentation as an example for iteration over an ordered aggregate [24]. The program iterates over an aggregate of type *ArrayList* using an iterator of type *IEnumerator*. Note that since iteration in this instance is sequential, it is matched with the first special case of the pattern introduced earlier. Sources given below for *IEnumerable*, *ArrayList* and *IEnumerator* are extracts from the C# documentation for each of these parts. No specific implementations exist for these parts. Source given for the client part is an actual C# program that is included in the documentation for class *ArrayList* as an example on how to use this class. This instance is selected to show that the approach of the study applies both in presence and in absence of the source code provided that good documentation exists.

6.1 Instance Source

Sources are given only for the parts that appear in the instance formal description. Parts that are specific to the application are left out.

IEnumerable

```
System.Collections.IEnumerable Interface

Summary
Implemented by classes that support a simple iteration over instances of the collection.

Description
[Note: System.Collections.IEnumerable contains the System.Collections.IEnumerable.
GetEnumerator method. The consumer of an object should call this method to obtain
```

```
 an enumerator for simple iteration over an instance of a collection.Implement this
 interface to support the foreach semantics of C#.]

   IEnumerable.GetEnumerator() Method

Summary
Returns a System.Collections.IEnumerator that can be used for simple iteration
over a collection.

Return Value
A System.Collections.IEnumerator that can be used for simple iteration over
a collection.
```

ArrayList

```
System.Collections.ArrayList Class
Implements:
 System.Collections.IList
 System.Collections.ICollection
 System.Collections.IEnumerable
 System.ICloneable

Summary
Implements a variable-size System.Collections.IList that uses an array of objects to
store its elements.

Description
System.Collections.ArrayList implements a variable-size System.Collections.IList
 that uses an array of
objects to store the elements. A System.Collections.ArrayList has a
System.Collections.ArrayList.Capacity, which is the allocated length
of the internal array. The total number of elements contained by a list
 is its System.Collections.ArrayList.Count. As elements are added to
 a list, its capacity is automatically increased as required by reallocating
 the internal array.

  ArrayList.GetEnumerator() Method

Summary
Returns a System.Collections.IEnumerator for the current instance.

Return Value
A System.Collections.IEnumerator for the current instance.

Description
If the the current instance is modified while an enumeration is in progress,
a call to System.Collections.IEnumerator.MoveNext or
System.Collections.IEnumerator.Reset throws System.InvalidOperationException.
```

IEnumerator

```
System.Collections.IEnumerator Interface

Summary
Implemented by classes that support a simple iteration over a collection.

Description
[Note: System.Collections.IEnumerator contains the System.Collections.
IEnumerator.MoveNext and System.Collections.IEnumerator.Reset methods
 and the System.Collections.IEnumerator.Current property. The consumer
of an object should call these methods or use this property when iterating
 over or reading the elements of a collection.When an enumerator is
instantiated or a call is made to System.Collections.IEnumerator.Reset, the
enumerator is positioned immediately before the first element of the collection
 and a snapshot of the collection is taken. When the enumerator is in this
position, a call to System.Collections.IEnumerator.MoveNext is necessary
before reading System.Collections.IEnumerator.Current from the collection.
If changes are made to the collection(such as adding, modifying or deleting
elements) the snapshot may get out of sync, causing the enumerator to throw
 a System.InvalidOperationException if the System.Collections.IEnumerator.MoveNext
or System.Collections.IEnumerator.Reset are invoked. Two enumerators
instantiated from the same collection at the same time can have different
snapshots of the collection. Enumerators are intended to be used only to
read data in the collection.An enumerator does not have exclusive access to the
collection for which it was instantiated.]

  IEnumerator.Reset() Method

Summary
Positions the enumerator immediately before the first element in the collection.

Description
[Note: When the current instance is constructed or after System.Collections.
IEnumerator.Reset is called, the current instance is positioned immediately
before the first element of the collection, use
System.Collections.IEnumerator.MoveNext to position
the current instance over the first element
of the collection.]

Behaviors
A call to System.Collections.IEnumerator.Reset is required to position the
current instance immediately before the first element of the collection. If elements are added,
removed, or repositioned in the collection after the current instance was instantiated,
it is required that a call to System.Collections.IEnumerator.Reset throw a
System.InvalidOperationException.

  IEnumerator.MoveNext() Method

Summary
Advances the current instance to the next element of the collection.
```

```
Return Value
true if the current instance was successfully advanced to the next element;
false if the current
instance has passed the end of the collection.

Description
[Note: When the current instance is constructed or after System.Collections.
IEnumerator.Reset is
called, the current instance is positioned immediately before the first element
of the collection. Use System.Collections.IEnumerator.MoveNext to position it
over the first element of the collection.]

Behaviors
A call to System.Collections.IEnumerator.MoveNext is required to position the
current instance over the next element in the collection and return true if the
 current instance was not positioned
beyond the last element of the collection when System.Collections.IEnumerator.
MoveNext was called. If the current instance is already positioned immediately
 after the last element of the collection, a call to
System.Collections.IEnumerator.MoveNext is required to return false, and
the current instance is required to remain in the same position. If elements
are added, removed, or repositioned in the collection after the current
 instance was instantiated, it is required that a call to
System.Collections.IEnumerator.MoveNext throw System.InvalidOperationException.

  IEnumerator.Current Property

Summary
Gets the element in the collection over which the current instance is positioned.

Property Value
The element in the collection over which the current instance is positioned.

Description
[Note: When the current instance is constructed or after System.Collections.
IEnumerator.
Reset is called, use System.Collections.IEnumerator.MoveNext to
position the current
instance over the first element of the collection.]

Behaviors
It is required that System.Collections.IEnumerator.Current return the
element in the collection over which the current instance is positioned
unless it is positioned before the first or after the last element of the
collection. If the current instance is positioned before the first element or
after the last element of the collection, System.Collections.IEnumerator.Current
 is required to throw System.InvalidOperationException If elements were added,
removed, or repositioned in the collection after the current instance was
```

```
instantiated, System.Collections.IEnumerator.Current returns the value it would
have returnedbefore the collection was modified.
It is also required that System.Collections.IEnumerator.Current not
change the position of the current instance: consecutive calls to System.Collections.
IEnumerator.Current are required to return the same object until either
 System.Collections.IEnumerator.MoveNext or
System.Collections.IEnumerator.Reset is called.
```

Client

```
using System;
using System.Collections;

public class SamplesArrayList {

  public static void Main() {

  // Create and initialize a new ArrayList.
  ArrayList myAL = new ArrayList();
  myAL.Add("Hello");
  myAL.Add("World");
  myAL.Add("!");

  // Display the properties and values of the ArrayList.
  Console.WriteLine( "myAL" );
  Console.WriteLine( "Count: {0}", myAL.Count );
  Console.WriteLine( "Capacity: {0}", myAL.Capacity );
  Console.Write( "Values:" );
  PrintValues( myAL );
  }

public static void PrintValues( IEnumerable myList ) {

  IEnumerator myEnumerator = myList.GetEnumerator();
  while ( myEnumerator.MoveNext() )
    Console.Write( " {0}", myEnumerator.Current );
    Console.WriteLine();
  }
}
```

Notes:

- The output for the above program is:

```
myAL
Count: 3
Capacity: 16
Values: Hello World !
```

6.2 Instance Description

class *IEnumerable*
  method *GetEnumerator* : *IEnumerator*
end

class *ArrayList* **implements** *IEnumerable*
  attr *cont* : **seq of** *Object*
  initialization
    $cont := \langle\rangle$
  method *GetEnumerator* : *IEnumerator*
    return new *Enumerator*(*this*)
end

class *IEnumerator*
  method *Reset*
  method *MoveNext* : *boolean*
  method *Current* : *Object*
end

class *Enumerator* **implements** *IEnumerator*
  attr *valid* : *boolean*

```
  attr aggregate : ArrayList
  attr container : seq of Object
  attr i : integer
  initialization (agg : ArrayList)
    begin
      this.valid := true;
      this.aggregate := agg;
      this.container := agg.cont;
      this.i := −1
    end
  method Reset
    begin
      assert this.valid;
      this.i := −1
    end
  method MoveNext : boolean
    begin
      assert this.valid;
      if this.i < length(this.container) then this.i := this.i + 1;
      return this.i ≤ length(this.container)
    end
  method Current : Object
    begin
      assert this.i > −1;
      assert this.i < length(this.container);
      return this.container[i]
    end
end


method Client
    var myAL : ArrayList, myEnumerator : IEnumerator •
  begin
```

myAL := **new** *ArrayList*
myEnumerator := *myList*.*GetEnumerator*
myEnumerator.*MoveNext*
end

Notes:

- In this instance, source code is only available for the client part. It is given as an example of constructing the description based on design documentation rather than the source code. The attributes used in the above instance description follow those introduced in the pattern description itself.

- *Enumerator* is not part of the instance. It is added based on the requirements of classes that implement *IEnumerator* as in the *C#* documentation [24].

- The elements of *container* start with element *container*$[0]$ and end with element *container*$[$*length*$($*container*$) - 1]$.

- Variable *i* is initialized to -1 because a call to *MoveNext* is required before *Current* can be called. The first call to *MoveNext* will advance *i* to 0. Note that *container*$[0]$ is the first element of *container*.

- *C#* documentation for *Reset* and *MoveNext* require that if the original container is edited, an exception should be raised. However, this is not required for *Current*.

- Exceptions required by the *C#* documentation are represented in the instance description using *assert* statements.

Data Structures

var *IEnumerator*, *IEnumerable* : **set of** *Object* := $\{\}, \{\}$
var *Enumerator*, *ArrayList* : **set of** *Object*
var *aggregate* : *Object* $\rightarrow$ *Object*
var *container* : *Object* $\rightarrow$ **seq of** *Object*

Note that:

$Enumerator \subseteq IEnumerator$ and $ArrayList \subseteq IEnumerable$

Instance Invariant $P1$

$$(\forall k \in Enumerator \bullet k.container = k.aggregate.cont \vee \neg\, k.valid) \wedge \tag{6.1}$$

$$(\forall k \in Enumerator \bullet -1 \leq k.i \leq length(k.container) \tag{6.2}$$

The above invariant reads as follows:

- The elements stored in the iterator container should be the same as those in the associated aggregate or the iterator is invalid.
- The valid range for attribute *i* which represents the pointer to the current element at *container* is between -1 and $length(container)$.

We notice that the instance invariant is weaker than the relevant pattern invariant $(PA \Rightarrow P1)$. This is an early indication of a potential mismatch of the instance with the pattern. This is going to be illustrated in the behavioral compliance given later on.

6.3 Structural Compliance

The pattern description given above involves both structural and behavioral statements. To check the structural compliance of the instance with the pattern, we need to match each structural statement in the pattern with an equivalent statement in the instance.

class *Aggregate*
  method *CreateIterator* : *Iterator*
end
is matched by :
class *IEnumerable*
  method *GetEnumerator* : *IEnumerator*
end

class *SeqAggregate* **implements** *Aggregate*
is matched by :
class *ArrayList* **implements** *IEnumerable*

class *Iterator*
  method *First*
  method *Next*
  method *IsDone* : *boolean*
  method *CurrentItem* : *Object*
end
is matched by :
class *IEnumerator*
  method *Reset*
  method *MoveNext* : *boolean*
  method *Current* : *Object*
end

Note that we do not need to match each method in the pattern with one in the instance. This is because we are going to compare the behavior of instance methods with the behavior of pattern methods. Methods in the implementing instance can be split or merged.

class *SeqIterator* **implements** *Iterator*
is matched by :
class *Enumerator* **implements** *IEnumerator*

method *Client*
    var *ag* : *Aggregate*, *iter* : *Iterator* •
  begin
    ag := **new** *ConcreteAggregate*;

iter := *ag*.*CreateIterator*;
iter.*Next*
end
is partially matched by :
method *Client*
var *myAL* : *ArrayList*, *myEnumerator* : *IEnumerator* •
begin
myAL := **new** *ArrayList*
myEnumerator := *myList*.*GetEnumerator*
myEnumerator.*MoveNext*
end

The first statement is clearly not matched by the instance. Variable *myAL* should be declared as *IEnumerable* rather than *ArrayList*. Failure to use the more abstract type *IEnumerable* removes the flexibility of using different aggregate with a minimal code changes.

6.4 Behavioral Compliance

An instance complies behaviorally with the pattern if the behavior suggested by pattern methods is supported by instance methods. An instance that misses one or more functionalities in its methods may still partially comply with the pattern.

- *Initialization*: *SeqIterator* initialization is partially matched by *Enumerator* initialization. In both cases, *i* represents a pointer to an element of *container*. In case of *SeqIterator*, *i* in initialized to 0 to point at the first element of *container*. In case of *Enumerator*, *i* in initialized to -1 to point immediately before the first element of *container*. In the later case, a call to *MoveNext* is necessary before being able to access the first element of *container*.

- *First*: This method provides the ability to reuse the same iterator. Similar to the argument used with initialization, this method is partially matched by method *Reset*. Notice that it is only thanks to formalization that we realize that *Reset* and *First* are intended to do the same thing. Method names do not imply this.

- *Next*: This method provides the ability to advance the pointer to current element. It is matched by method *MoveNext* in the instance. The only difference between the two methods is the return type. This neither affects the functionality nor adds side effects.

- *IsDone*: This method provides the ability to verify if we have more elements, without changing the state of the iterator. Even though the functionality expected from *IsDone* is included in the functionality of *MoveNext*, we see that *MoveNext* does more than that simple check. In case of *MoveNext*, the pointer to the current element will be advanced as a side effect. Therefore, the functionality of checking if we are at the end of the container without side effects is not supported by the instance.

- *CurrentItem*: This method provides the ability to retrieve the current element without changing the state of the iterator. It is partially matched by method *Current*. *C#* documentation does not require a check if the original container is edited. That check is specified in the pattern description and is missing in the instance. *CurrentItem* has two extra assertions that are not in the pattern description. These assertions are required by the *C#* documentation. However, they do not cause side effects and are not affecting the compliance.

6.5 Instance Invariant Preservation

We use rule (3.1) introduced earlier to check invariant preservation of the instance.

- Module initialization establishes the invariant as the universal quantification in both predicates is over an empty range.

- *ConcreteIterator* initialization preserves the invariant because:
  1. First predicate (6.1) of the instance invariant is directly established by the first two statement of the initialization.
  2. Second predicate (6.2) of the instance invariant is also established as the initialization sets i to -1, a value within the valid range for i.

- *Reset* preserves the invariant because:
  1. First predicate (6.1) holds as the method asserts that the copy of container main-

tained by the iterator is still equal to the associated aggregate's container. *C#* documentation requires that if the original container is edited, an exception should be raised.
2. Second predicate (6.2) holds as the method sets *i* to -1, a value within the valid range for *i*.

- *MoveNext* preserves the invariant because:
  1. First predicate (6.1) holds as shown above.
  2. Second predicate (6.2) holds as the value of *i* is only increments after checking that it is less than *length*(*container*). Therefore, the maximum possible value for *i* is equal to *length*(*container*), which is within the valid range for *i*.

- *Current* preserves the invariant because:
  The method does not change the value of any attribute in the program.

Based on the information given above, we see that the given program is indeed an instance of the *Iterator* pattern. This is because the program preserves the invariant and it provides at least some of the functionalities described by the pattern. We also see that some functionalities described by the pattern are missing from the program. Therefore, it only complies partially with the pattern.

Chapter 7

Abstract Factory

The book of *Gamma et al*. [12] describes the *Abstract Factory* pattern as an interface for creating families of related objects without specifying their concrete classes. The pattern makes exchanging product families easy and promotes consistency among products.

Abstract Factory pattern involves five participants. *AbstractFactory* declares methods that create abstract product objects. *ConcreteFactory* implements methods to create concrete product objects. *AbstractProduct* declares an interface for a type of product object, and is implemented by a *ConcreteProduct*. A *Client* declares and deals with abstract products. Concrete product objects are only returned by factory objects.

The pattern is applicable whenever we need a system to be independent of how its products are created, composed, and represented. It is also applicable in systems that should be configured with one of many families of products. Another application for the pattern is to enforce using a family of related product objects together, or to reveal only the interfaces of a library of products.

An instance of the pattern will typically need only one object of *ConcreteFactory* per product family. Therefore, *ConcreteFactory* is best implemented as a *Singleton*. This pattern is an example to demonstrate that combining patterns may amount to conjoining their invariants. The introduced invariant for *Abstract Factory* involves the invariant for *Singleton*. The invariant for *Singleton* is introduced later on in this study.

The description given below captures the essence of the pattern and does not necessarily map to a direct implementation of the pattern. For example, the initializers of concrete products assign every set of product to an empty set. This is done to make sure that all

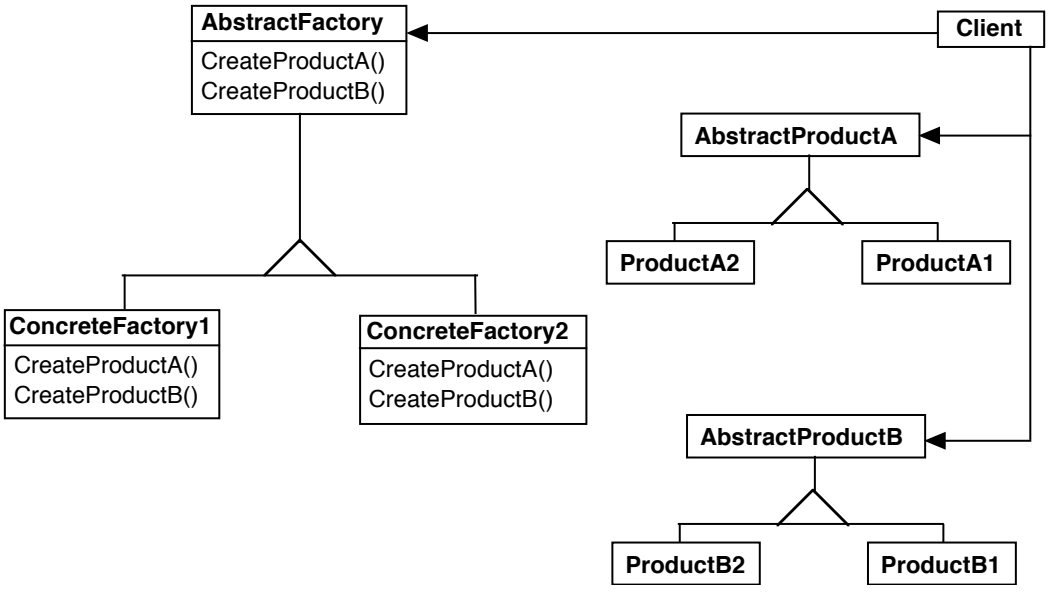

Figure 7.1: Abstract Factory Class Diagram

products will belong to one family of products.

7.1 The Pattern

class *AbstractFactory*
  method *CreateProductA* : *AbstractProductA*
  method *CreateProductB* : *AbstractProductB*
end

class *ConcreteFactory*1 **implements** *AbstractFactory*
  initialization
    AbstractProductA := {};
    AbstractProductB := {};

```
    ConcreteFactory1 := {this};
    AbstractFactory := ConcreteFactory1
  method CreateProductA : AbstractProductA
    var c : Object •
    begin
      c :∉ ProductA1 ∪ {nil};
      ProductA1 := ProductA1 ∪ {c};
      return c
    end
  method CreateProductB : AbstractProductB
    var c : Object •
    begin
      c :∉ ProductB1 ∪ {nil};
      ProductB1 := ProductB1 ∪ {c};
      return c
    end
end

class ConcreteFactory2 implements AbstractFactory
  initialization
    AbstractProductA := {};
    AbstractProductB := {};
    ConcreteFactory2 := this;
    AbstractFactory := ConcreteFactory2
  method CreateProductA : AbstractProductA
    var c : Object •
    begin
      c :∉ ProductA2 ∪ {nil};
      ProductA2 := ProductA2 ∪ {c};
      return c
    end
  method CreateProductB : AbstractProductB
```

```
        var c : Object •
        begin
          c :∉ ProductB2 ∪ {nil};
          ProductB2 := ProductB2 ∪ {c};
          return c
        end
end


class AbstractProductA
  method AnyOperation
end


class ProductA1 implements AbstractProductA
  method AnyOperation
end


class ProductA2 implements AbstractProductA
  method AnyOperation
end


class AbstractProductB
  method AnyOperation
end


class ProductB1 implements AbstractProductB
  method AnyOperation
end
```

class *ProductB*2 **implements** *AbstractProductB*
  method *AnyOperation*
end

method *Client*
  var *factory* : *AbstractFactory*, *product* : *AbstractProductA* •
  begin
    factory := **new** *ConcreteFactory*1;
    product := *factory*.*CreateProductA*;
    product.*AnyOperation*
  end

The return of a new product instance in methods *CreateProductA* and *CreateProductB* is equivalent to adding the new product instance to the set of all this product objects. For example, $ProductA1 := ProductA1 \cup \{\textbf{new}\ ProductA1\}$.

Data Structures

var *AbstractFactory*, *AbstractProductA*, *AbstractProductB* : **set of** *Object*
:= {}, {}, {}
var *ProductA*1, *ProductA*2, *ProductB*1, *ProductB*2 : **set of** *Object*
var *ConcreteFactory*1, *ConcreteFactory*2 : **set of** *Object*
$ProductA1, ProductA2 \subseteq AbstractProductA$
$ProductB1, ProductB2 \subseteq AbstractProductB$
$ConcreteFactory1, ConcreteFactory2 \subseteq AbstractFactory$

Pattern Invariant *P*

$| AbstractFactory | \leq 1 \wedge ($
$(AbstractProductA = ProductA1) \wedge$

$$
\begin{array}{l}
\quad (\mathit{AbstractProductB} = \mathit{ProductB}1) \wedge \\
\quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}1) \\
\vee \\
\quad (\mathit{AbstractProductA} = \mathit{ProductA}2) \wedge \\
\quad (\mathit{AbstractProductB} = \mathit{ProductB}2) \wedge \\
\quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}2)\ )
\end{array}
$$

The above invariant reads as follows:

- Only one factory instance is allowed.
- In case the only factory object is of type *ConcreteFactory*1, all objects declared as *AbstractProductA* are in *ProductA*1, and all objects declared as *AbstractProductB* are in *ProductB*1.
  In case the only factory object is of type *ConcreteFactory*2, all objects declared as *AbstractProductA* are in *ProductA*2, and all objects declared as *AbstractProductB* are in *ProductB*2.

7.2 Well-Definedness of the Pattern

A module initialization is required to establish the invariant. Class methods and initializations are only required to preserve the invariant. We apply the proofs on *ConcreteFactory*1. *ConcreteFactory*2 is identical except for identifiers. Within *ConcreteFactory*1, it is also sufficient to give proofs for *initialization* and *CreateProductA*. Proof for *CreateProductB* is identical to *CreateProductA* except for identifiers. Concrete product classes are not involved in proofs.

Module Initialization Establishes Invariant

$$
\mathit{wlp}(\mathit{AbstractFactory} := \{\ \} \wedge \mathit{AbstractProductA} := \{\ \} \wedge \mathit{AbstractProductB} := \{\ \}\ , P)
$$

$$
\begin{array}{ll}
\equiv & (|\ \mathit{AbstractFactory}\ | \leq 1 \wedge (\\
& \quad (\mathit{AbstractProductA} = \mathit{ProductA}1) \wedge
\end{array}
$$

$$
\begin{array}{l}
\quad (\mathit{AbstractProductB} = \mathit{ProductB}1) \wedge \\
\quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}1) \\
\vee \\
\quad (\mathit{AbstractProductA} = \mathit{ProductA}2) \wedge \\
\quad (\mathit{AbstractProductB} = \mathit{ProductB}2) \wedge \\
\quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}2)\ ) \\
)\ [\mathit{AbstractFactory}, \mathit{AbstractProductA}, \mathit{AbstractProductB} \setminus \{\ \}, \{\ \}, \{\ \}]
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{substitution}, \mathit{ProductA}1 \subseteq \mathit{AbstractProductA},\ \mathit{etc.} \gg \\
& (|\ \{\ \}\ | \leq 1 \wedge (\\
& \quad (\{\ \} = \{\ \}) \wedge \\
& \quad (\{\ \} = \{\ \}) \wedge \\
& \quad (\{\ \} = \{\ \}) \\
& \vee \\
& \quad (\{\ \} = \{\ \}) \wedge \\
& \quad (\{\ \} = \{\ \}) \wedge \\
& \quad (\{\ \} = \{\ \})\ )
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{logic} \gg \\
& \textbf{\textit{true}}
\end{array}
$$

ConcreteFactory1 Initialization Preserves Invariant

The statement $\mathit{this} :\notin \mathit{ConcreteFactory}1 \cup \{\mathit{nil}\}$is added to $\mathit{ConcreteFactory}1$ initialization due to the translation to a procedure.

$$
\begin{array}{ll}
& \mathit{wlp}(\mathit{ConcreteFactory}1.\mathit{new}, P) \\
\\
\equiv & \ll \textbf{definition}\ \mathit{of}\ \mathit{ConcreteFactory}1.\mathit{new} \gg \\
& \mathit{wlp}(\mathit{this} :\notin \mathit{ConcreteFactory}1 \cup \{\mathit{nil}\}; \mathit{ConcreteFactory}1 := \{\mathit{this}\}; \\
& \qquad \mathit{AbstractProductA} := \{\}; \mathit{AbstractProductB} := \{\}; \\
& \qquad \mathit{AbstractFactory} := \mathit{ConcreteFactory}1 \\
& , P) \\
\\
\Leftarrow & \ll \textbf{wlp}\ \mathit{of}\ \mathit{AbstractFactory} := \mathit{ConcreteFactory}1, \mathit{rule}\ (3.4), \mathit{rule}\ (3.9) \gg
\end{array}
$$

$$
\begin{array}{ll}
 & \ll \text{, } \textbf{substitution} \gg \\
 & \mathit{wlp}(\mathit{this} :\notin \mathit{ConcreteFactory}1 \cup \{\mathit{nil}\}; \mathit{ConcreteFactory}1 := \{\mathit{this}\}; \\
 & \quad \mathit{AbstractProductA} := \{\}; \mathit{AbstractProductB} := \{\}; \\
 & \text{, } | \ \boldsymbol{\mathit{ConcreteFactory}1} \ | \leq 1 \wedge (\\
 & \quad (\mathit{AbstractProductA} = \mathit{ProductA}1) \wedge \\
 & \quad (\mathit{AbstractProductB} = \mathit{ProductB}1) \wedge \\
 & \quad (\boldsymbol{\mathit{ConcreteFactory}1} = \mathit{ConcreteFactory}1) \\
 & \vee \\
 & \quad (\mathit{AbstractProductA} = \mathit{ProductA}2) \wedge \\
 & \quad (\mathit{AbstractProductB} = \mathit{ProductB}2) \wedge \\
 & \quad (\boldsymbol{\mathit{ConcreteFactory}1} = \mathit{ConcreteFactory}2) \ ) \\
 &) \\
 & \\
\Leftarrow & \ll \textbf{wlp } \textit{of the last two statements respectively}, \textit{rule } (3.4), \textit{rule } (3.9) \gg \\
 & \ll \text{, } \textbf{substitution} \gg \\
 & \mathit{wlp}(\mathit{this} :\notin \mathit{ConcreteFactory}1 \cup \{\mathit{nil}\}; \mathit{ConcreteFactory}1 := \{\mathit{this}\} \\
 & \text{, } | \ \mathit{ConcreteFactory}1 \ | \leq 1 \wedge (\\
 & \quad (\{\} = \mathit{ProductA}1) \wedge \\
 & \quad (\{\} = \mathit{ProductB}1) \wedge \\
 & \quad (\mathit{ConcreteFactory}1 = \mathit{ConcreteFactory}1) \\
 & \vee \\
 & \quad (\{\} = \mathit{ProductA}2) \wedge \\
 & \quad (\{\} = \mathit{ProductB}2) \wedge \\
 & \quad (\mathit{ConcreteFactory}1 = \mathit{ConcreteFactory}2) \ ) \\
 &) \\
 & \\
\equiv & \ll \textbf{logic}, \mathit{ProductA}1 \subseteq \mathit{AbstractProductA}, \mathit{ProductB}1 \subseteq \mathit{AbstractProductB} \gg \\
 & \mathit{wlp}(\mathit{this} :\notin \mathit{ConcreteFactory}1 \cup \{\mathit{nil}\}; \mathit{ConcreteFactory}1 := \{\mathit{this}\} \\
 & \text{, } | \ \mathit{ConcreteFactory}1 \ | \leq 1 \wedge (\\
 & \quad (\{\} = \boldsymbol{\{\}}) \wedge \\
 & \quad (\{\} = \boldsymbol{\{\}}) \wedge \\
 & \quad (\mathit{ConcreteFactory}1 = \mathit{ConcreteFactory}1)
\end{array}
$$

$$
\begin{array}{l}
\quad \vee \\
\quad\quad (\{\} = \mathit{ProductA}2) \wedge \\
\quad\quad (\{\} = \mathit{ProductB}2) \wedge \\
\quad\quad (\mathit{ConcreteFactory}1 = \mathit{ConcreteFactory}2)\ ) \\
\quad)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{logic} \gg \\
& \mathit{wlp}(\mathit{this} :\notin \mathit{ConcreteFactory}1 \cup \{\mathit{nil}\}; \mathit{ConcreteFactory}1 := \{\mathit{this}\} \\
& , \mid \mathit{ConcreteFactory}1 \mid \leq 1 \wedge \\
& \textbf{\textit{true}}\ )
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{logic}, \textit{the cardinality of a set of one element } = 1 \gg \\
& \mathit{wlp}(\mathit{this} :\notin \mathit{ConcreteFactory}1 \cup \{\mathit{nil}\}; \mathit{ConcreteFactory}1 := \\
& \mathit{ConcreteFactory}1 \cup \{\mathit{this}\} \\
& , \textbf{\textit{true}}\ )
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \ll \textbf{wlp } \textit{of any statement with respect to true } \equiv \textit{ true} \gg \\
& \textbf{\textit{P}}
\end{array}
$$

CreateProductA Preserves Invariant

The statement **assert** *this* $\in$ *ConcreteFactory*1 is added to *CreateProductA*.

$$
\begin{array}{ll}
& \mathit{wlp}(\mathit{ConcreteFactory}1.\mathit{CreateProductA}, P)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{definition } \textit{of } \mathit{ConcreteFactory}1.\mathit{CreateProductA} \gg \\
& \mathit{wlp}(\textbf{assert } \mathit{this} \in \mathit{ConcreteFactory}1; c :\notin \mathit{ProductA}1 \cup \{\mathit{nil}\}; \\
& \quad \mathit{ProductA}1 := \mathit{ProductA}1 \cup \{c\}; \textbf{return } C, P)
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \ll \textbf{wlp } \textit{of } \textbf{return } c, \textit{rules } (3.9) \textit{ and } (3.11) \gg \\
& \mathit{wlp}(\textbf{assert } \mathit{this} \in \mathit{ConcreteFactory}1; c :\notin \mathit{ProductA}1 \cup \{\mathit{nil}\}; \\
& \quad \mathit{ProductA}1 := \mathit{ProductA}1 \cup \{c\}, P)
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \ll \textbf{wlp } \textit{of } \mathit{ProductA}1 := \mathit{ProductA}1 \cup \{c\}, \textit{rules } (3.9) \textit{ and } (3.4) \gg \\
& \mathit{wlp}(\textbf{assert } \mathit{this} \in \mathit{ConcreteFactory}1; c :\notin \mathit{ProductA}1 \cup \{\mathit{nil}\}
\end{array}
$$

$$
\begin{array}{l}
\quad , (|\ \mathit{AbstractFactory}\ | \leq 1 \wedge (\\
\quad\quad (\mathit{AbstractProductA} = \mathit{ProductA}1) \wedge \\
\quad\quad (\mathit{AbstractProductB} = \mathit{ProductB}1) \wedge \\
\quad\quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}1) \\
\quad \vee \\
\quad\quad (\mathit{AbstractProductA} = \mathit{ProductA}2) \wedge \\
\quad\quad (\mathit{AbstractProductB} = \mathit{ProductB}2) \wedge \\
\quad\quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}2)\ ) \\
\quad)[\boldsymbol{\mathit{ProductA}1 \setminus \mathit{ProductA}1 \cup \{c\}}]
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{substitution}, \mathit{ProductA}1 \subseteq \mathit{AbstractProductA} \gg \\
& \mathit{wlp}(\textbf{assert}\ \mathit{this} \in \mathit{ConcreteFactory}1; c :\notin \mathit{ProductA}1 \cup \{\mathit{nil}\} \\
& , |\ \mathit{AbstractFactory}\ | \leq 1 \wedge (\\
& \quad (\mathit{AbstractProductA} \cup \boldsymbol{\{c\}} = \mathit{ProductA}1 \cup \boldsymbol{\{c\}}) \wedge \\
& \quad (\mathit{AbstractProductB} = \mathit{ProductB}1) \wedge \\
& \quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}1) \\
& \vee \\
& \quad (\mathit{AbstractProductA} = \mathit{ProductA}2) \wedge \\
& \quad (\mathit{AbstractProductB} = \mathit{ProductB}2) \wedge \\
& \quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}2)\ )\ )
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \ll \textbf{logic} \gg , \mathit{rule}\ (3.9) \\
& \mathit{wlp}(\textbf{assert}\ \mathit{this} \in \mathit{ConcreteFactory}1; c :\notin \mathit{ProductA}1 \cup \{\mathit{nil}\} \\
& , (|\ \mathit{AbstractFactory}\ | \leq 1 \wedge (\\
& \quad (\mathit{AbstractProductA} = \mathit{ProductA}1) \wedge \\
& \quad (\mathit{AbstractProductB} = \mathit{ProductB}1) \wedge \\
& \quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}1) \\
& \vee \\
& \quad (\mathit{AbstractProductA} = \mathit{ProductA}2) \wedge \\
& \quad (\mathit{AbstractProductB} = \mathit{ProductB}2) \wedge \\
& \quad (\mathit{AbstractFactory} = \mathit{ConcreteFactory}2)\ ) \\
&)
\end{array}
$$

$\equiv$ $\ll$ **wlp** *of* $c :\notin$ *ProductA*$1 \cup \{$*nil*$\}$, *rule* $(3.6) \gg$
wlp(**assert** *this* $\in$ *ConcreteFactory*1
,$\forall$ ***h*** $\in$ $\overline{\textbf{\textit{ConcreteFactory1}}}$ • (
(| *AbstractFactory* $|\leq 1 \wedge$ (
(*AbstractProductA* $=$ *ProductA*$1)\wedge$
(*AbstractProductB* $=$ *ProductB*$1)\wedge$
(*AbstractFactory* $=$ *ConcreteFactory*1)
$\vee$
(*AbstractProductA* $=$ *ProductA*$2)\wedge$
(*AbstractProductB* $=$ *ProductB*$2)\wedge$
(*AbstractFactory* $=$ *ConcreteFactory*2))
))

$\equiv$ $\ll$ *empty range*, "*c*" *does not appear in predicate* $\gg$
wlp(**assert** *this* $\in$ *ConcreteFactory*$1, P$)

$\equiv$ $\ll$ **wlp** *of* **assert** *this* $\in$ *ConcreteFactory*1, *rule* $(3.10) \gg$
this $\in$ *ConcreteFactory*1
$\Rightarrow$ *P*

$\Leftarrow$ $\ll$ **definition** *of implication*, **strengthening** $\gg$
P

7.3 First Instance

This example represents a solution to a common design problem. That problem is the need to have programs making use of persistent data, such that the underlying data storage is subject to change from one implementation to another. The example uses as a solution a *DAO* (Data Access Object) to abstract access to data source. Changing the underlying data storage from *Oracle* to *Cloudscape* for example should then be done with minimal client code changes [3].

Instance Source

Sources are given only for the parts that appear in the instance formal description. Parts that are specific to the application are left out.

DAOFactory

```
public abstract class DAOFactory {

  public static final int CLOUDSCAPE = 1;
  public static final int ORACLE = 2;
  public static final int SYBASE = 3;
  ...
  public abstract CustomerDAO getCustomerDAO();
  public abstract AccountDAO getAccountDAO();
  public abstract OrderDAO getOrderDAO();
  ...
  public static DAOFactory getDAOFactory(
      int whichFactory) {

    switch (whichFactory) {
      case CLOUDSCAPE:
          return new CloudscapeDAOFactory();
      case ORACLE    :
          return new OracleDAOFactory();
      case SYBASE    :
          return new SybaseDAOFactory();
      ...
      default           :
          return null;
    }
  }
}
```

CloudscapeDAOFactory

```
public class CloudscapeDAOFactory extends DAOFactory {
  public static final String DRIVER=
    "COM.cloudscape.core.RmiJdbcDriver";
```

```
  public static final String DBURL=
    "jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";

  // method to create Cloudscape connections
  public static Connection createConnection() {
    // Use DRIVER and DBURL to create a connection
    // Recommend connection pool implementation/usage
  }
  public CustomerDAO getCustomerDAO() {
    // CloudscapeCustomerDAO implements CustomerDAO
    return new CloudscapeCustomerDAO();
  }
  public AccountDAO getAccountDAO() {
    // CloudscapeAccountDAO implements AccountDAO
    return new CloudscapeAccountDAO();
  }
  public OrderDAO getOrderDAO() {
    // CloudscapeOrderDAO implements OrderDAO
    return new CloudscapeOrderDAO();
  }
  ...
}
```

CustomerDAO

```
public interface CustomerDAO {
  public int insertCustomer(...);
  public boolean deleteCustomer(...);
  public Customer findCustomer(...);
  public boolean updateCustomer(...);
  public RowSet selectCustomersRS(...);
  public Collection selectCustomersTO(...);
  ...
}
```

CloudscapeCustomerDAO

```
public class CloudscapeCustomerDAO implements CustomerDAO {
   public int insertCustomer(...) {
    // Implement insert customer here.
```

```
    // Return newly created customer number
    // or a -1 on error
  }
  ...
}
```

Auxiliary Class

```
public class Customer implements java.io.Serializable {
  // member variables
  int CustomerNumber;
  String name;
  String streetAddress;
  String city;
  ...
  // getter and setter methods...
  ...
}
```

Client

```
...
// create the required DAO Factory
DAOFactory cloudscapeFactory =
  DAOFactory.getDAOFactory(DAOFactory.DAOCLOUDSCAPE);

// Create a DAO
CustomerDAO custDAO =
  cloudscapeFactory.getCustomerDAO();

// create a new customer
int newCustNo = custDAO.insertCustomer(...);

// Find a customer object. Get the value object.
Customer cust = custDAO.findCustomer(...);

// modify the values in the value object.
cust.setAddress(...);
cust.setEmail(...);
// update the customer object using the DAO
```

```
custDAO.updateCustomer(cust);

// delete a customer object
custDAO.deleteCustomer(...);
// select all customers in the same city
Customer criteria=new Customer();
criteria.setCity("New York");
Collection customersList =
  custDAO.selectCustomersVO(criteria);
// returns customersList - collection of Customer
// value objects. iterate through this collection to
// get values.
...
```

Notes:

- Client does not instanciate a *CloudscapeDAOFactory* directly, but rather makes a call to method *getDAOFactory* which returns one concrete factory based on the value of passed argument.

Instance Description

class *DAOFactory*
  method *getCustomerDAO* : *CustomerDAO*
  method *getAccountDAO* : *AccountDAO*
end

class *CloudscapeDAOFactory* **implements** *DAOFactory*
  method *getCustomerDAO* : *CustomerDAO*
    return new *CloudscapeCustomerDAO*
  method *getAccountDAO* : *AccountDAO*
    return new *CloudscapeAccountDAO*
end

class *OracleDAOFactory* **implements** *DAOFactory*

```
  method getCustomerDAO : CustomerDAO
    return new OracleCustomerDAO
  method getAccountDAO : AccountDAO
    return new OracleAccountDAO
end


class CustomerDAO
  method insertCustomer : integer
  ...
end


class CloudscapeCustomerDAO implements CustomerDAO
  method insertCustomer : integer
  ...
end


class OracleCustomerDAO implements CustomerDAO
  method insertCustomer : integer
  ...
end


method Client
    var cloudscapeFactory : DAOFactory, custDAO : CustomerDAO •
  begin
    cloudscapeFactory := new CloudscapeDAOFactory;
    custDAO := cloudscapeFactory.getCustomerDAO;
    custDAO.insertCustomer
  end
```

Data Structures

$$DAOFactory, CustomerDAO, AccountDAO : \textbf{set of } Object$$
$$CloudscapeCustomerDAO, OracleCustomerDAO \subseteq CustomerDAO$$
$$CloudscapeAccountDAO, OracleAccountDAO \subseteq AccountDAO$$
$$CloudscapeDAOFactory, OracleDAOFactory \subseteq DAOFactory$$

Instance Invariant $P1$

$$\begin{aligned}
&| DAOFactory | \leq 1 \wedge (\\
&\quad (CustomerDAO = CloudscapeCustomerDAO) \wedge \\
&\quad (AccountDAO = CloudscapeAccountDAO) \wedge \\
&\quad (DAOFactory = CloudscapeDAOFactory) \\
&\vee \\
&\quad (CustomerDAO = OracleCustomerDAO) \wedge \\
&\quad (AccountDAO = OracleAccountDAO) \wedge \\
&\quad (DAOFactory = OracleDAOFactory) \ )
\end{aligned}$$

Invariant $P1$ **Preservation**

Invariant $P1$ is identical to pattern invariant P, except for identifiers. Invariant $P1$ is weekly preserved in this instance, this is done implicitly through the naming convention. We see that the only instance of *DAOFactory* in the client application is *cloudscapeFactory*. That name suggests that it can only be assigned objects of *CloudscapeDAOFactory* and that there is no need to change the type at runtime as it is the case with GUI applications for example. That application still gets the benefit of applying *AbstractFactory* pattern. In case the application needs to use a different database connection type, then all we need to change is the line where the factory object is instantiated.

7.4 Second Instance

The pattern is applied in this program that is used to plan garden layouts. Different types of gardens (vegetable gardens, annual gardens, etc.) are considered. Gardens contain

different types of plants (center, border, etc.). Different garden types have different suitable center plants and different suitable border plants. User Interface has radio buttons representing the different garden types (vegetable, annual, etc.), buttons representing plant types (center, border, etc.), and an area to display the suitable plant name. Program user selects at runtime the garden type using radio buttons and clicks on a plant type to display the suitable plant name [5].

Instance Source

Sources are given only for the parts that appear in the instance formal description. Parts that are specific to the application are left out.

Garden

```
public interface Garden {
    public Plant getShade();
    public Plant getCenter();
    public Plant getBorder();
}
```

VeggieGarden

```
public class VeggieGarden implements Garden {
    public Plant getShade() {
        return new Plant("Broccoli");
    }
    public Plant getCenter() {
        return new Plant("Corn");
    }
    public Plant getBorder() {
        return new Plant("Peas");
    }
}
```

AnnualGarden

```
public class AnnualGarden implements Garden {
    public Plant getShade() {
        return new Plant("Coleus");
    }
```

```
    public Plant getCenter() {
        return new Plant("Marigold");
    }
    public Plant getBorder() {
        return new Plant("Alyssum");
    }
}
```

Plant

```
public class Plant {
    private String name;
    public Plant(String pname) {
        name = pname;     //save name
    }
    public String getName() {
        return name;
    }
}
```

Client

```
public class Gardener extends Frame
implements ActionListener {
    private Checkbox Veggie, Annual, Peren;
    private Button Center, Border, Shade, Quit;
    private Garden garden = null;
    private String borderPlant = "", centerPlant = "", shadePlant = "";

    public Gardener() {
        super("Garden planner");
        setGUI();
    }
    private void setGUI() {
        ...
        Veggie = new Checkbox("Vegetable", grp, false);
        ...
        Veggie.addItemListener(new VeggieListener());
        ...
    }
```

```
   public void actionPerformed(ActionEvent e) {
        Object obj = e.getSource();
        if (obj == Center)
            setCenter();
       ...
    }
    private void setCenter() {
        if (garden != null) centerPlant = garden.getCenter().getName();
        gardenPlot.repaint();
    }
    private void clearPlants() {
        shadePlant=""; centerPlant=""; borderPlant = "";
        gardenPlot.repaint();
    }
    static public void main(String argv[]) {
        new Gardener();
    }
    class GardenPanel extends Panel {
        public void paint (Graphics g) {
           ...
        }
    }
    class VeggieListener implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            garden = new VeggieGarden();
            clearPlants();
        }
    }
}     //end of Gardener class
```

Instance Description

class *Garden*
method *getCenter* : *Plant*
method *getBorder* : *Plant*
end

```
class VeggieGarden implements Garden
  method getCenter : Plant
    return new Plant("Corn")
  method getBorder : Plant
    return new Plant("Peas")
end


class AnnualGarden implements Garden
  method getCenter : Plant
    return new Plant("Marigold")
  method getBorder : Plant
    return new Plant("Alyssum")
end


class Plant
  attr name : String
  initialization (pname : String)
    name := pname
  method getName : String
    return name
end


method Client
    var centerPlant, borderPlant : String, garden : Garden •
  begin
    garden := new VeggieGarden;
    centerPlant = ""; borderPlant = "";
    centerPlant := garden.getCenter.getName
```

end

Notes:

- Resetting *centerPlant* and *borderPlant* is done in auxiliary method *clearPlants*.
- The call $centerPlant := garden.getCenter.getName;$ is done in method *setCenter* which is called by the event handler of buttons. That event handler checks which button $(Center, Border, etc.)$ was clicked and calls the appropriate method $(setCenter, setBorder, etc.)$ accordingly.
- A similar technique is followed with the assignment $garden :=$ **new** *VeggieGarden*;. Radio buttons representing different garden types (vegetable, annual, etc.) are implemented using a *Checkbox* for each garden type. All these are grouped together to form a set of radio buttons such that only one can be selected at a time. Each *Checkbox* is mapped to an inner class event handler that actually does the above assignment based on the garden type selected.

Data Structures

This example is a simple one, no significant work is done by product instances. That is why the same class *Plant* is used to represent more than one product. Different products may be viewed here as subsets of *Plant* with different values for the *String* attribute *name*. *Plant* was also not sub-classed based on different concrete factories. However, the program still follows the same spirit as suggested by the pattern. This is because sub-classes of *Plant* are also viewed as subsets of *Plant* with different values for attribute *name*.

$Garden, Plant$: **set of** *Object*

- Let $VeggieCenterPlant \subseteq Plant$ be the set of all $plant : Plant$ where $plant.getName = "Corn"$
- Let $AnnualCenterPlant \subseteq Plant$ be the set of all $plant : Plant$ where $plant.getName = "Marigold"$

- Let *VeggieBorderPlant* $\subseteq$ *Plant* be the set of all *plant* : *Plant* where *plant.getName* $=$ "*Peas*"

- Let *AnnualBorderPlant* $\subseteq$ *Plant* be the set of all *plant* : *Plant* where *plant.getName* $=$ "*Alyssum*"

Instance Invariant $P2$

$$
\begin{aligned}
&| \mathit{Garden} | \leq 1 \wedge (\\
&\quad (\mathit{centerPlant} = \text{"}\mathit{Corn}\text{"}) \wedge \\
&\quad (\mathit{borderPlant} = \text{"}\mathit{Peas}\text{"}) \wedge \\
&\quad (\mathit{Garden} = \mathit{VeggieGarden}) \\
&\vee \\
&\quad (\mathit{centerPlant} = \text{"}\mathit{Marigold}\text{"}) \wedge \\
&\quad (\mathit{borderPlant} = \text{"}\mathit{Alyssum}\text{"}) \wedge \\
&\quad (\mathit{Garden} = \mathit{AnnualGarden}) \;)
\end{aligned}
$$

Invariant $P2$ **Preservation**

Invariant *P*2 is still equal to pattern invariant *P*, except for identifiers. *AbstractProductA* defined with pattern invariant contains exactly one element in that example, namely *centerPlant*. Similarly *AbstractProductB* contains only *borderPlant*. Classes $\mathit{ProductA}1, \mathit{ProductA}2, \mathit{ProductB}1$ *and* $\mathit{ProductB}2$ can be viewed as $\mathit{VeggieCenterPlant}, \mathit{AnnualCenterPlant}, \mathit{VeggieBorderPlant}$ *and AnnualBorderPlant* respectively. The distinction between those is based on the value of attribute *name* defined in class *Plant*. Invariant *P*2 is preserved in this instance by reseting the values for *centerPlant* and *borderPlant* to an empty *String* after an assignment or a reassignment to the factory instance *garden*.

7.5 Third Instance

This example introduces an interface *IAVDevice* as the abstract factory. A factory can create audio and video objects as products. Concrete factories are "cd" and dvd". The

program can deal with both audio and video files, and manipulate them on different media types, namely ”cd” and ”dvd” [20].

Instance Source

Sources are given only for the parts that appear in the instance formal description. Parts that are specific to the application are left out.

IAVDevice

```
public interface IAVDevice
{
 IAudio GetAudio();
 IVideo GetVideo();
}
```

CCd

```
class CCd:IAVDevice
{
 public IAudio GetAudio()
 {
  return new CCdAudio();
 }
 public IVideo GetVideo()
 {
  return new CCdVideo();
  }
}
```

CDvd

```
class CDvd:IAVDevice
{
        public IAudio GetAudio()
        {
                 return new CDvdAudio();
        }
        public IVideo GetVideo()
        {
```

```
                return new CDvdVideo();
        }
}
```

IAudio

```
public interface IAudio
{
 string GetSoundQuality();
}
```

CCdAudio

```
class CCdAudio:IAudio
{
 public string  GetSoundQuality()
 {
  return "CD Audio is better then DVD Audio";
 }
}
```

CDvdAudio

```
class CDvdAudio:IAudio
{
 public string  GetSoundQuality()
 {
  return "DVD Audio is not as good as CD Audio";
  }
}
```

Auxiliary Class

```
class CAVMaker
{
 public IAVDevice  AVMake(string xWhat)
 {
  switch (xWhat.ToLower())
  {
  case "cd":
  return new CCd();
  case "dvd":
```

```
  return new CDvd();
  default:
  return new CCd();
  }
 }
}
```

Client

```
public class AbstractFactory
{
 static void Main(string[]  args)
 {
  CAVMaker objFactMaker = new CAVMaker();
  IAVDevice objFact;
  IAudio objAudio;
  IVideo objVideo;
  string strWhat;
  strWhat = args[0];
  objFact = objFactMaker.AVMake(strWhat);
  objAudio = objFact.GetAudio();
  objVideo = objFact.GetVideo();
  Console.WriteLine(objAudio.GetSoundQuality());
  Console.WriteLine(objVideo.GetPictureQuality());
 }
}
```

Instance Description

class *IAVDevice*
method *GetAudio* : *IAudio*
method *GetVideo* : *IVideo*
end

class *CCd* **implements** *IAVDevice*
method *GetAudio* : *IAudio*
return new *CCdAudio*

```
    method GetVideo : IVideo
        return new CCdVideo
end


class CDvd implements IAVDevice
    method GetAudio : IAudio
        return new CDvdAudio
    method GetVideo : IVideo
        return new CDvdVideo
end


class IAudio
    method GetSoundQuality : String
end


class CCdAudio implements IAudio
    method GetSoundQuality : String
        ...
end


class CDvdAudio implements IAudio
    method GetSoundQuality : String
        ...
end


method Main(args : seq of String)
        var objFact : IAVDevice, objAudio : IAudio •
```

begin
if $(args[0] = "cd")$ **then** *objFact* := **new** *CCd*;
else if $(args[0] = "dvd")$ **then** *objFact* := **new** *CDvd*;
objAudio := *objFact*.*GetAudio*;
objAudio.*GetSoundQuality*
end

Notes:

- Instantiating *CCd* or *CDvd* is done in method *AVMake* of auxiliary class *CAVMaker*.
- The *case* statement in the above auxiliary method is translated to an *if* statement.
- *args*$[0]$ is the value passed to main application method *Main* as a command line argument.

Data Structures

$$IAVDevice, IAudio, IVideo : \textbf{set of } Object$$
$$CCdAudio, CDvdAudio \subseteq IAudio$$
$$CCdVideo, CDvdVideo \subseteq IVideo$$
$$CCd, CDvd \subseteq IAVDevice$$

Instance Invariant $P3$

$$\begin{array}{l} \mid IAVDevice \mid \leq 1 \wedge (\\ \quad (IAudio = CCdAudio) \wedge \\ \quad (IVideo = CCdVideo) \wedge \\ \quad (IAVDevice = CCd) \\ \vee \\ \quad (IAudio = CDvdAudio) \wedge \\ \quad (IVideo = CDvdVideo) \wedge \\ \quad (IAVDevice = CDvd) \ ) \end{array}$$

Invariant $P3$ **Preservation**

Invariant $P3$ is identical to pattern invariant *P*, except for identifiers. Invariant $P3$ is also

implicitly preserved by this instance. This is done by making the decision of which concrete factory to instantiate based on a command line argument. This suggests that the program will typically have only one factory instance, and that the value of this instances should never be changed at runtime. That application gets the full benefit of applying *AbstractFactory* pattern, and no changes are required to client code when switching from ”cd” to ”dvd”.

Chapter 8

Composite

As described in the book of *Gamma et al*. [12], the *Composite* pattern composes objects into tree structures to represent containment hierarchies. The pattern lets clients treat individual and composite objects uniformly. In the class diagram, *Component* is a common interface between *Leaf* and *Composite*.

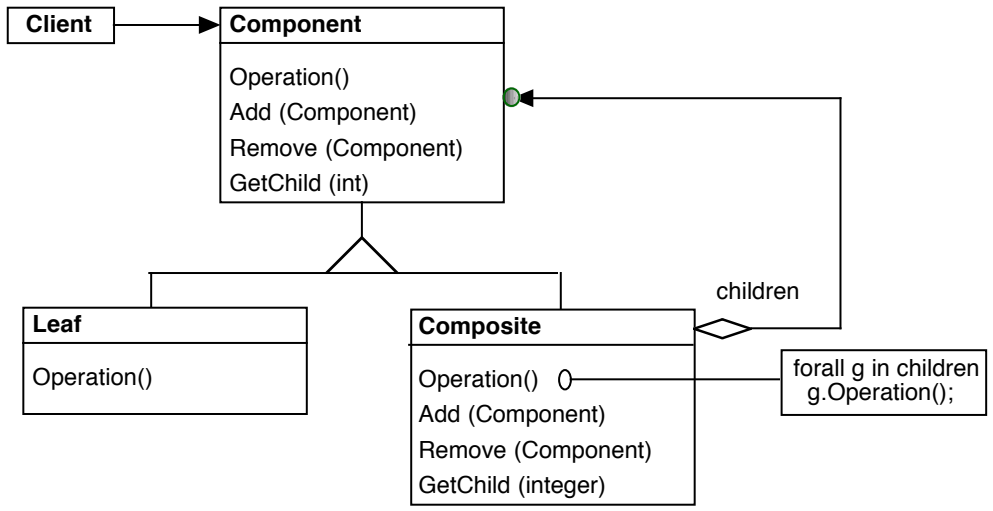

Figure 8.1: Composite Class Diagram

The pattern involves four participants. *Component* declares the interface for objects in the composition and also for accessing and managing child components. *Leaf* represents leaf objects in the composition. A leaf has no children. *Composite* defines behavior for components having children. It stores child components. *Client* declares and deals uniformly with the common interface *Component*.

The pattern is applicable whenever we want to present a part-whole hierarchies of objects. It is also applicable when we need a client to be able to treat all objects in the composite structure uniformly.

8.1 The Pattern

class *Component*
  method *Operation*
  method *Add*$(c : Component)$
  method *Remove*$(c : Component)$
  method *GetChild*$(i : integer) : Component$
end

class *Leaf* **implements** *Component*
  method *Operation*
end

class *Composite* **implements** *Component*
  attr *children* : **seq of** *Object*
  initialization
    $this.children := \langle\rangle$
  method *Operation*
    for $i \in this.children$ **do** $i.Operation$
  method *Add*$(c : Component)$
    assert $\neg c\ parent^{+}\ this$;

$this.children := this.children \ \& \ \langle c \rangle$
method *Remove*$(c : Component)$
$this.children := this.children \ - \ \langle c \rangle$
method *GetChild*$(i : integer) : Component$
return $this.children[i]$
end

Notes:

- In this study, we relax the requirement that object composition is a tree structure as explained later on.
- We abstractly represent the children of a composite as a sequence. This allows the existence of the same object multiple times in the same container of children. This assumptions extends the applicability of the pattern. An example would be to represent components of a machine as *Component*. A machine may contain multiple parts of the same component, in which case it may not be efficient to represent each as a different object.
- The relation *parent* is defined as follows:
  $x \ parent \ y = y \in x.children$
  It maps objects of *Component* to objects of *Composite*. *x parent y* means that *x* is a parent of *y*.
- The relation $parent^+$ is the transitive closure of *parent*. *x* $parent^+$ *y* means that *x* is an ancestor of *y*.

Data Structures

var $Component, Composite, Leaf$: **set of** *Object* $:= \{\}, \{\}, \{\}$
var *children* : *Object* $\rightarrow$ **seq of** *Object*

Note that:

$Composite, Leaf \subseteq Component$

Pattern Invariant P

$$\forall\, i \in \mathit{Composite} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ )$$

The above invariant reads as follows:
Composite objects contain no ancestor objects within their *children* containers.
The above invariant allows any object in *Component* within the containment hierarchy to be contained by multiple objects of *Composite*. Therefore, the containment hierarchy does not need to be a balanced tree, or even a tree.

8.2 Well-Definedness of the Pattern

A module initialization is required to establish the invariant. Other parts of the system are only required to preserve the invariant. Method *Operation* represents a functionality to be decided by pattern instances. It is not involved in proofs.

Module Initialization Establishes Invariant

$$\mathit{wlp}(\mathit{Component} := \{\ \} \wedge \mathit{Composite} := \{\ \} \wedge \mathit{Leaf} := \{\ \}, P)$$

$$\equiv \ \ll \textbf{wlp}\ \mathit{of}\ \mathit{Component} := \{\ \}, \mathit{rule}\ (3.4) \gg$$
$$\forall\, i \in \mathit{Composite} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ )[\mathit{Component} \setminus \{\ \}]$$

$$\equiv \ \ll \textbf{substitution}, \mathit{Composite} \subseteq \mathit{Component} \gg$$
$$\forall\, i \in \textbf{\{ \}} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ )$$

$$\equiv \ \ll \textbf{logic}, \mathit{universal\ quantification\ over\ empty\ range} \gg$$
$$\textbf{\textit{true}}$$

Composite Initialization Preserves Invariant
The statements $\mathit{this} :\notin \mathit{Composite} \cup \{\mathit{nil}\}\ ;\ \mathit{Composite} := \mathit{Composite} \cup \{\mathit{this}\}$ are added to *Composite* initialization due to the translation to a procedure.

$$\mathit{wlp}(\mathit{Composite.new}, P)$$

$\equiv$ $\ll$ **definition** *of Composite.new* $\gg$
$wlp(this :\notin Composite \cup \{nil\}\ ;\ Composite := Composite \cup \{this\};$
$this.children := \langle\rangle$
$, \forall\, i \in Composite \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i))\ )$

$\Leftarrow$ $\ll$ **wlp** *of* $this.children := \langle\rangle$, *rule* (3.7) $\gg$
$wlp(this :\notin Composite \cup \{nil\}\ ;\ Composite := Composite \cup \{this\}$
$, (\forall\, i \in Composite \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i))$
$)[\mathbf{children \setminus (children; this : \langle\rangle)}]\ )$

$\equiv$ $\ll$ **case analysis** *with* $i =$ *this and* $i \neq this$ $\gg$
$wlp(this :\notin Composite \cup \{nil\}\ ;\ Composite := Composite \cup \{this\}$
$, (\forall\, i \in \mathbf{Composite - \{this\}} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i))\ ) \wedge$
$\mathbf{(\forall\, j \in this.children \bullet \neg(j\ parent^+\ this))}$
$)[children \setminus (children; this : \langle\rangle)]\ )$

$\equiv$ $\ll$ **substitution**, **simplification**, *rules* (3.2) *and* (3.3) $\gg$
$wlp(this :\notin Composite \cup \{nil\}\ ;\ Composite := Composite \cup \{this\}$
$, \forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i))\ ) \wedge$
$(\forall\, j \in \langle\rangle \bullet \neg(j\ parent^+\ this))\ )$

$\equiv$ $\ll$ **logic** $\gg$
$wlp(this :\notin Composite \cup \{nil\}\ ;\ Composite := Composite \cup \{this\}$
$, \forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i))\ )$

$\Leftarrow$ $\ll$ **wlp** *of* $Composite := Composite \cup \{this\}$, *rules* (3.4) *and* (3.9) $\gg$
$wlp(this :\notin Composite \cup \{nil\}$
$, \forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i))$
$)\ [\mathbf{Composite \setminus Composite \cup \{this\}}]$

$\equiv$ $\ll$ **substitution**, $Composite \cup \{this\} - \{this\} = Composite$ $\gg$

$$
\begin{array}{ll}
 & wlp(this :\notin Composite \cup \{nil\} \\
 & , \forall\, i \in \textbf{\textit{Composite}} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i))\ ) \\
 & \\
\equiv & \ll \textbf{wlp}\ of\ this :\notin Composite \cup \{nil\}, rule\ (3.6) \gg \\
 & \forall h \in \overline{\textbf{\textit{Composite}}} \bullet (\\
 & \qquad \forall\, i \in Composite \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i))\ ) \\
 & \\
\equiv & \ll empty\ range,\ "this"\ does\ not\ appear\ in\ predicate \gg \\
 & \forall\, i \in Composite \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i))\ ) \\
 & \\
\Leftarrow & \\
 & \textbf{\textit{P}}
\end{array}
$$

Add Preserves Invariant

The statement **assert** *this* $\in$ *Composite* is added to *Add* due to the translation from a method to a procedure.

$$
\begin{array}{ll}
 & wlp(Composite.Add, P) \\
 & \\
\equiv & \ll \textbf{definition}\ of\ Composite.Add \gg \\
 & wlp(\textbf{assert}\ this \in Composite; \textbf{assert}\ \neg(c\ parent^{+} this); \\
 & this.children := this.children\ \&\ \langle c \rangle \\
 & , \forall\, i \in Composite \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i))\ ) \\
 & \\
\Leftarrow & \ll \textbf{wlp}\ of\ this.children := this.children\ \&\ \langle c \rangle, rule\ (3.7), rule\ (3.9) \gg \\
 & wlp(\textbf{assert}\ this \in Composite; \textbf{assert}\ \neg(c\ parent^{+} this) \\
 & , (\forall\, i \in Composite \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i))\ ) \\
 &)[\textbf{\textit{children}} \setminus (\textbf{\textit{children}}\ ; \textbf{\textit{this}} :\ \textbf{\textit{this.children}}\ \&\ \langle \textbf{\textit{c}} \rangle)]\ ) \\
 & \\
\equiv & \ll \textbf{case analysis}\ with\ i = this\ and\ i \neq this \gg \\
 & wlp(\textbf{assert}\ this \in Composite; \textbf{assert}\ \neg(c\ parent^{+} this) \\
 & , (\forall\, i \in \textbf{\textit{Composite}} - \{\textbf{\textit{this}}\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i))\ ) \wedge \\
 & (\forall\, \textbf{\textit{j}} \in \textbf{\textit{this.children}} \bullet \neg(\textbf{\textit{j}}\ \textbf{\textit{parent}}^{+}\ \textbf{\textit{this}})\ ) \\
 &)[children \setminus (children\ ; this :\ this.children\ \&\ \langle c \rangle)]\ )
\end{array}
$$

$\equiv$ $\ll$ **substitution**, **simplification** $\gg$
wlp(**assert** *this* $\in$ *Composite*; **assert** $\neg$(*c parent*$^+$*this*)
, $\forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i)\ ) \wedge$
$(\forall\, j \in \mathbf{this.children}\ \&\ \langle c \rangle \bullet \neg(j\ parent^+\ this)\ )\ )$

$\Leftarrow$ $\ll$ **wlp** *of* **assert** $\neg$(*c parent*$^+$*this*), *rule* (3.10) $\gg$
wlp(**assert** *this* $\in$ *Composite*
, $\neg$(*c parent*$^+$*this*)
$\Rightarrow$
$\forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i)\ ) \wedge$
$(\forall\, j \in this.children\ \&\ \langle c \rangle \bullet \neg(j\ parent^+\ this)\ )\ )$

$\equiv$ $\ll$ **logic**, *range split* $\gg$
wlp(**assert** *this* $\in$ *Composite*
, $\neg$(*c parent*$^+$*this*)
$\Rightarrow$
$\forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i)\ ) \wedge$
$(\forall\, j \in \mathbf{this.children} \bullet \neg(j\ parent^+\ this)\ ) \wedge$
$\mathbf{\neg(c\ parent^+\ this)}\ )$

$\equiv$ $\ll$ **logic**, *for any* $p, q : (p \Rightarrow q \wedge p) \equiv (p \Rightarrow q)$ $\gg$
wlp(**assert** *this* $\in$ *Composite*
$\neg$(*c parent*$^+$*this*)
$\Rightarrow$
$\forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^+\ i)\ ) \wedge$
$(\forall\, j \in this.children \bullet \neg(j\ parent^+\ this)\ )\ )$

$\equiv$ $\ll$ **wlp** *of* **assert** *this* $\in$ *Composite*, *rule* (3.10) $\gg$
this $\in$ *Composite*
$\Rightarrow$
$\neg$(*c parent*$^+$*this*)

$$
\begin{array}{ll}
 & \Rightarrow \\
 & \quad \forall\, i \in \mathit{Composite} - \{\mathit{this}\} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ ) \wedge \\
 & \quad (\forall\, j \in \mathit{this}.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ \mathit{this})\ )
\end{array}
$$

$\equiv$ $\ll$ *join the last two predicates* $\gg$

$$
\begin{array}{l}
\quad \mathit{this} \in \mathit{Composite} \\
\Rightarrow \\
\quad \neg(c\ \mathit{parent}^{+} \mathit{this}) \\
\quad \Rightarrow \\
\quad \forall\, i \in \mathbf{Composite} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ ) \wedge
\end{array}
$$

$\Leftarrow$ $\ll$ **definition** *of implication*, **strengthening** $\gg$

$$
\forall\, i \in \mathit{Composite} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ ) \wedge
$$

$\equiv$

$\boldsymbol{P}$

Remove Preserves Invariant

The statement **assert** *this* $\in$ *Composite* is added to *Remove* due to the translation from a method to a procedure.

$$
\mathit{wlp}(\mathit{Composite}.\mathit{Remove}, P)
$$

$\equiv$ $\ll$ **definition** *of Composite.Remove* $\gg$

$$
\begin{array}{l}
\mathit{wlp}(\textbf{assert}\ \mathit{this} \in \mathit{Composite}; \mathit{this}.\mathit{children} := \mathit{this}.\mathit{children}\ -\ \langle c \rangle \\
, \forall\, i \in \mathit{Composite} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ )\ )
\end{array}
$$

$\Leftarrow$ $\ll$ **wlp** *of this.children* := *this.children* $-$ $\langle c \rangle$, *rule* (3.7) $\gg$

$$
\begin{array}{l}
\mathit{wlp}(\textbf{assert}\ \mathit{this} \in \mathit{Composite} \\
, (\forall\, i \in \mathit{Composite} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ ) \\
)[\boldsymbol{\mathit{children} \setminus (\mathit{children}\ ;\ \mathit{this}\ :\ \mathit{this}.\mathit{children} - \langle c \rangle)}]\ )
\end{array}
$$

$\equiv$ $\ll$ **case analysis** *with* $i =$ *this and* $i \neq$ *this* $\gg$

$$
\mathit{wlp}(\textbf{assert}\ \mathit{this} \in \mathit{Composite}
$$

$$
\begin{array}{l}
\quad , (\forall\, i \in \textbf{\textit{Composite}} - \{\textbf{\textit{this}}\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i)\ ) \wedge \\
\quad \textbf{(}\forall\, \textbf{\textit{j}} \in \textbf{\textit{this.children}} \bullet \neg\textbf{(}\textbf{\textit{j parent}}^{+}\ \textbf{\textit{this}}\textbf{)}\ \textbf{)} \\
\quad)[children \setminus (children\ ;this :\ \ this.children - \langle c \rangle)]\ )
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{substitution}, \textbf{simplification} \gg \\
& wlp(\textbf{assert}\ this \in Composite \\
& , \forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i)\ ) \wedge \\
& (\forall\, j \in \textbf{\textit{this.children}}\ -\ \langle \textbf{\textit{c}} \rangle \bullet \neg(j\ parent^{+}\ this)\ )\ )
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{wlp}\ of\ \ \textbf{assert}\ this \in Composite, rule\ (3.10) \gg \\
& \quad this \in Composite \\
& \Rightarrow \\
& \quad \forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i)\ ) \wedge \\
& \quad (\forall\, j \in \textbf{\textit{this.children}}\ -\ \langle \textbf{\textit{c}} \rangle \bullet \neg(j\ parent^{+}\ this)\ )
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \ll \textbf{definition}\ of\ implication, \textbf{strengthening} \gg \\
& \forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i)\ ) \wedge \\
& (\forall\, j \in \textbf{\textit{this.children}}\ -\ \langle \textbf{\textit{c}} \rangle \bullet \neg(j\ parent^{+}\ this)\ )
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \ll \textbf{logic}, (\forall i \in X - \{a\}) \Leftarrow (\forall i \in X - \{a\}\ \wedge a) \equiv (\forall i \in X)\ \gg \\
& \forall\, i \in Composite - \{this\} \bullet (\forall\, j \in i.children \bullet \neg(j\ parent^{+}\ i)\ ) \wedge \\
& (\forall\, j \in \textbf{\textit{this.children}} \bullet \neg(j\ parent^{+}\ this)\ )
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \ll join\ the\ last\ two\ predicates \gg \\
& \textbf{\textit{P}}
\end{array}
$$

GetChild Preserves Invariant

The statement **assert** *this* $\in$ *Composite* is added to *GetChild* due to the translation from a method to a procedure.

$$wlp(Composite.GetChild, P)$$

$$
\begin{array}{ll}
\equiv & \ll \textbf{definition}\ of\ Composite.GetChild \gg \\
& wlp(\textbf{assert}\ this \in Composite; \textbf{return}\ \ this.children[i]
\end{array}
$$

$$, \forall\, i \in \mathit{Composite} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ )\ )$$

$\Leftarrow$ $\ll$ **wlp** *of* **return** $\mathit{this.children}[i], \mathit{rule}\ (3.11) \gg$
$\mathit{wlp}($ **assert** $\mathit{this} \in \mathit{Composite}, P)$

$\equiv$ $\ll$ **wlp** *of* **assert** $\mathit{this} \in \mathit{Composite}, \mathit{rule}\ (3.10) \gg$
$\mathit{this} \in \mathit{Composite}$
$\Rightarrow$
$\forall\, i \in \mathit{Composite} \bullet (\forall\, j \in i.\mathit{children} \bullet \neg(j\ \mathit{parent}^{+}\ i)\ )$

$\Leftarrow$ $\ll$ **definition** *of* *implication*, **strengthening** $\gg$
$\boldsymbol{P}$

8.3 First Instance

This part of JHotDraw framework allows graphical applications to build composite figures with a hierarchical structure of components such that all contained components act as one unit. A common application is programs that draw class diagrams. *CompositeFigure* is then sub-classed to be a graphical representation of classes, while *AttributeFigure* is a graphical representation of class attributes [21].

Instance Source
Sources are given only for the parts that appear in the instance formal description. Parts that are specific to the application are left out.

Figure

```
public interface Figure extends Storable, Cloneable, Serializable {
    public void draw(Graphics g);

public abstract class AbstractFigure implements Figure {
  ...
}
```

AttributeFigure

```
public abstract class AttributeFigure extends AbstractFigure {
    ...
    public void draw(Graphics g) {
        Color fill = getFillColor () ;
        ...
        Color frame = getFrameColor ();
        if (!ColorMap.isTransparent(frame)) {
            g.setColor(frame);
            drawFrame(g);
        }
    }
```

CompositeFigure

```
public abstract class CompositeFigure extends AbstractFigure
 implements FigureChangeListener {
    protected Vector fFigures;
    ...
    public void draw(Graphics g) {
        FigureEnumeration k = figures();
        while (k.hasMoreElements())
            k.nextFigure().draw(g);
    }
   public Figure add(Figure figure) {
        if (!fFigures.contains(figure)) {
            fFigures.addElement(figure);
            figure.addToContainer(this);
        }
        return figure;
    }
   public Figure remove(Figure figure) {
        if (fFigures.contains(figure)) {
            figure.removeFromContainer(this);
            fFigures.removeElement(figure);
        }
        return figure;
    }
   public Figure figureAt(int i) {
        return (Figure)fFigures.elementAt(i);
    }
```

```
   public final FigureEnumeration figures() {
        return new FigureEnumerator(fFigures);
    }
    ...
}
```

Instance Description

class *Figure*
  method *draw*(*g* : *Graphics*)
end

class *AttributeFigure* **implements** *Figure*
  method *draw*(*g* : *Graphics*)
end

class *CompositeFigure* **implements** *Figure*
  attr *fFigures* : **seq of** *Figure*
  method *draw*(*g* : *Graphics*)
    for $i \in$ *fFigures* **do** *i.draw*
  method *add*(*figure* : *Figure*) : *Figure*
    begin
      $fFigures := fFigures \ \& \ \langle figure \rangle$;
      ...
      return *figure*
  method *remove*(*figure* : *Figure*) : *Figure*
    begin
      $fFigures := fFigures \ - \ \langle figure \rangle$;
      ...
      return *figure*
    end

method *figureAt*(*i* : *integer*) : *Figure*
return *fFigures*[*i*]
end

Data Structures

Figure : **set of** *Object*
CompositeFigure, *AttributeFigure* $\subseteq$ *Figure*
fFigures : *Object* $\rightarrow$ **seq of** *Object*

Instance Invariant $P1$

$\forall$ *c* $\in$ *CompositeFigure* • ($\forall$ *ch* $\in$ *c.fFigures* • $\neg$(*ch parent*$^+$ *c*))

Invariant $P1$ **Preservation**
Invariant $P1$ is identical to pattern invariant *P*, except for identifiers. It is given as a proposed invariant for the instance vs. an invariant that is maintained by the instance.
An example of violating the proposed invariant, which may lead to an infinite loop is:

Figure $f0$:= **new** *CompositeFigure*;
Figure $f1$:= **new** *CompositeFigure*;
Figure $f2$:= **new** *CompositeFigure*;
$f1$.*add*($f2$);
$f0$.*add*($f1$);
$f2$.*add*($f0$)

Apparently any call to $f0$.*draw* or $f2$.*draw* will cause an infinite loop. As the program does not preserve the proposed invariant, it may not be considered as an instance of the *Composite* pattern.

8.4 Second Instance

Java AWT supports both components and containers. Components such as *Button* can be added to containers. Containers can still be added to other containers because they are also

components [18].

Instance Source

Sources are given only for the parts that appear in the instance formal description. Parts that are specific to the application are left out.

Component

```
public abstract class Component implements
ImageObserver, MenuContainer, Serializable
{
    ...
    public void update(Graphics g) {
        if ((this instanceof java.awt.Canvas) ||
            (this instanceof java.awt.Panel)  ||
            (this instanceof java.awt.Window)) {
           g.clearRect(0, 0, width, height);
        }
        paint(g);
    }
    ...
}
```

Button

```
public class Button extends Component implements Accessible {
   ...
}
```

Container

```
public class Container extends Component {
    ...
    Component component[] = new Component[4];
    ...
    public void update(Graphics g) {
        if (isShowing()) {
            if (! (peer instanceof java.awt.peer.LightweightPeer)) {
                g.clearRect(0, 0, width, height);
            }
```

```
            paint(g);
        }
    }
    public Component add(Component comp) {
        addImpl(comp, null, -1);
        return comp;
    }
    public void remove(Component comp) {
        if (comp.parent == this)  {
            Component component[] = this.component;
            for (int i = ncomponents; --i >= 0; ) {
              if (component[i] == comp) {
                 remove(i);
              }
            }
        }
    }
    public Component getComponent(int n) {
        if ((n < 0) || (n >= ncomponents)) {
          throw new ArrayIndexOutOfBoundsException
          ("No such child: " + n);
        }
        return component[n];
    }
    ...
    public void paint(Graphics g) {
        if (isShowing() &&
            (!printing ||
            !printingThreads.contains(Thread.currentThread())) ) {
          GraphicsCallback.PaintCallback.getInstance().
          runComponents(component, g, GraphicsCallback.LIGHTWEIGHTS);
       }
    }
    public void remove(int index) {
        Component comp = component[index];
             if (peer != null) {
               comp.removeNotify();
             }
             if (layoutMgr != null) {
```

```
              layoutMgr.removeLayoutComponent(comp);
            }
            ...
            comp.parent = null;
            System.arraycopy(component, index + 1,
                             component, index,
                             ncomponents - index - 1);
            component[--ncomponents] = null;
            if (valid) {
              invalidate();
            }
            ...
            }
   }
   protected void addImpl(Component comp, Object constraints,
   int index) {
        ...
        if (ncomponents == component.length) {
          Component newcomponents[] =
          new Component[ncomponents * 2];
          System.arraycopy(component, 0, newcomponents, 0,
           ncomponents);
          component = newcomponents;
        }
        if (index == -1 || index == ncomponents) {
          component[ncomponents++] = comp;
        } else {
          System.arraycopy(component, index, component,
          index + 1, ncomponents - index);
          component[index] = comp;
          ncomponents++;
        }
        comp.parent = this;
        ...
}
```

Auxiliary Class

```
abstract class GraphicsCallback extends SunGraphicsCallback {
   static final class PaintCallback extends GraphicsCallback {
```

```
        private static PaintCallback instance = new PaintCallback();
       private PaintCallback() {}
        public void run(Component comp, Graphics cg) {
            comp.paint(cg);
        }
        static PaintCallback getInstance() {
            return instance;
        }
    }
    ...
}
```

Instance Description

class *Component*
  method *update*(*g* : *Graphics*)
end

class *Button* **implements** *Component*
end

class *Container* **implements** *Component*
  attr *component* : **seq of** *Component*
  method *update*(*g* : *Graphics*)
    for $i \in$ *component* **do** *i.update*
  method *add*(*comp* : *Component*) : *Component*
    $component := component \ \& \ \langle comp \rangle$
  method *remove*(*comp* : *Component*)
    $component := component \ - \ \langle comp \rangle$
  method *getComponent*(*n* : *integer*) : *Component*
    return *component*[*n*]
end

Notes:

- Class *Button* does not implement *update* but rather calls the parent method.
- Method *update* in class *Container* calls *paint* in the same class. The later call is forwarded to methods of class *GraphicsCallback*. However, the actual work is done by class *SunGraphicsCallback* in the internal package *sun.awt*.
- Method *add* in class *Container* calls *addImpl* in the same class, which does the actual addition.
- Method $remove(comp : Component)$ in class *Container* calls $remove(index : integer)$ in the same class. The first determines the index of the component to be removed, and the later does the actual removal.

Data Structures

$$Component : \textbf{set of } Object$$
$$Container, Button \subseteq Component$$
$$component : Object \rightarrow \textbf{seq of } Object$$

Instance Invariant $P2$

$$\forall\, c \in Container \bullet (\forall\, ch \in c.component \bullet \neg(ch\ parent^{+}\ c))$$

Invariant $P2$ **Preservation**
Invariant $P2$ is identical to pattern invariant P, except for identifiers. It is also given as a proposed invariant for the instance vs. an invariant that is maintained by the instance.
A similar argument to the one used in the first instance can be used here to show that failure to comply with this invariant may cause infinite loops. In that instance, this can occur when containers directly or indirectly contain each others.

8.5 Third Instance

The program can be used to represent a company's organizational chart with employees and bosses. Methods like *getSalaries* simply return an employee's salary if a regular

employee, and include salaries of all managed employees in case of a boss [5].

Instance Source

Sources are given only for the parts that appear in the instance formal description. Parts that are specific to the application are left out.

AbstractEmployee

```
public abstract class AbstractEmployee {
    protected String name;
    protected long salary;
    protected Employee parent = null;
    protected boolean leaf = true;
   public abstract long getSalary();
    public abstract String getName();
    public abstract boolean add(Employee e) throws NoSuchElementException;
    public abstract void remove(Employee e);
    public abstract Enumeration subordinates();
    public abstract Employee getChild(String s);
    public abstract long getSalaries();
    public boolean isLeaf() {
        return leaf;
    }
}
```

Employee

```
public class Employee extends AbstractEmployee {
    public Employee(String _name, long _salary) {
        name = _name;
        salary = _salary;
        leaf = true;
    }
    public Employee(Employee _parent, String _name, long _salary) {
        name = _name;
        salary = _salary;
        parent = _parent;
        leaf = true;
    }
```

```
    public long getSalary() {
        return salary;
    }
    public boolean add(Employee e) throws NoSuchElementException {
        throw new NoSuchElementException("No subordinates");
    }
    public void remove(Employee e) throws NoSuchElementException {
        throw new NoSuchElementException("No subordinates");
    }
    public Employee getChild(String s) throws NoSuchElementException {
        throw new NoSuchElementException("No children");
    }
    public long getSalaries() {
        return salary;
    }
    ...
}
```

Boss

```
public class Boss extends Employee {
    Vector employees;
   public Boss(String _name, long _salary) {
        super(_name, _salary);
        leaf = false;
        employees = new Vector();
    }
    public boolean add(Employee e) throws NoSuchElementException {
        employees.add(e);
        return true;
    }
    public void remove(Employee e) throws NoSuchElementException {
        employees.removeElement(e);
    }
    public Employee getChild(String s) throws NoSuchElementException {
       Employee newEmp = null;
       if (getName().equals(s))
            return this;
        else {
            boolean found = false;
```

```
            Enumeration e = subordinates();
            while (e.hasMoreElements() && (! found)) {
                newEmp = (Employee)e.nextElement();
                found = newEmp.getName().equals(s);
                if (! found) {
                    if (! newEmp.isLeaf ()) {
                        newEmp = newEmp.getChild(s);
                    } else
                        newEmp = null;
                    found =(newEmp != null);
                }
            }
            if (found)
                return newEmp;
            else
                return null;
        }
    }
    public long getSalaries() {
        long sum = salary;
        for (int i = 0; i < employees.size(); i++) {
            sum += ((Employee)employees.elementAt(i)).getSalaries();
        }
        return sum;
    }
    ...
}
```

Instance Description

class *AbstractEmployee*
attr *salary* : *long*
attr *name* : *String*
method *getSalaries* : *long*
method *add*(*e* : *Employee*) : *boolean*
method *remove*(*e* : *Employee*)
method *getChild*(*s* : *String*) : *Employee*
end

```
class Employee implements AbstractEmployee
  method getSalaries : long
    return salary
end


class Boss implements Employee
  attr employees : seq of Employee
  method getSalaries : long
    begin
      for i ∈ employees do salary := salary + i.getSalaries
      return salary
    end
  method add(e : Employee) : boolean
    employees := employees & ⟨e⟩
  method remove(e : Employee)
    employees := employees − ⟨e⟩
  method getChild(s : String) : Employee
    var e : Employee •
    begin
      result :∈ {e ∈ employees ∧ e.name = s};
      return result
    end
end
```

Data Structures

```
AbstractEmployee : set of Object
Employee ⊆ AbstractEmployee
Boss ⊆ Employee
employees : Object → seq of Object
```

Instance Invariant $P3$

$$\forall\, c \in \mathit{Boss} \bullet (\forall\, \mathit{ch} \in c.\mathit{employees} \bullet \neg(\mathit{ch}\ \mathit{parent}^{+}\ c)\ )$$

Invariant $P3$ **Preservation**

Invariant $P3$ is identical to pattern invariant *P*, except for identifiers. It is also given as a proposed invariant for the instance vs. an invariant that is maintained by the instance.

A similar argument to the one used in the first instance can be used here to show that failure to comply with this invariant may cause infinite loops. In that instance, this can occur when bosses directly or indirectly manage each others.

Unlike the class diagram describing the pattern, here the composite class extends leaf and the leaf extends the component.

Chapter 9

Singleton

The *Singleton* pattern is introduced in the book of *Gamma et al*. [12] as a way to ensure that a class only has one instance, and provides a global point of access to it. The pattern is an improvement over global variables. It avoids complicating the name space with global variables that store sole instances.

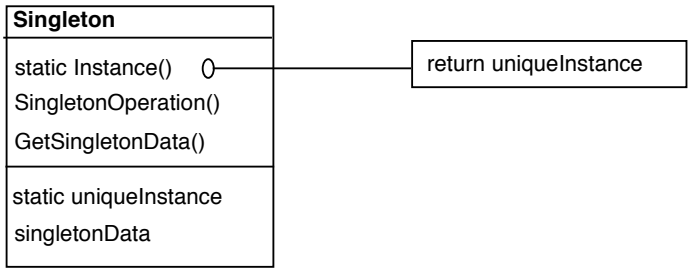

Figure 9.1: Singleton Class Diagram

Singleton pattern has only one participant. *Singleton* class defines a *static* method *Instance* that returns the only object of the class. The class uses a *static* attribute *singletonData* to store the only object of the class.

The pattern is applicable whenever there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point. The pattern is also applicable

when the sole instance of a class should be extensible by subclassing.

9.1 The Pattern

```
class Singleton
   static attr uniqueInstance : Singleton := nil
   attr SingletonData : Type
   private initialization
      ...
   static method Instance : Singleton
      if uniqueInstance = nil then uniqueInstance := new Singleton;
      Singleton := {uniqueInstance};
      return uniqueInstance
   method SingletonOperation
      ...
   method GetSingletonData
      return SingletonData
end



method Client
      var singleton : Singleton •
   begin
      singleton := Singleton.Instance;
      singleton.SingletonOperation
   end
```

Notes:

- *SingletonData* : *Type* is used to indicate that *SingletonData* can be of any type.
- *SingletonData*, *SingletonOperation* and *GetSingletonData* represent any data or operations that can be added to class *Singleton*.

- The return of *uniqueInstance* in the initialization of *Singleton* is equivalent to adding the instance to set *Singleton*. This is represented by *Singleton* $:= \{$*uniqueInstance*$\}$.

Data Structures

var *Singleton* : **set of** *Object* $:= \{\}$

Pattern Invariant *P*

$|$ *Singleton* $| \leq 1$

The above invariant reads as follows:
The cardinality of set *Singleton* should be less than or equal to one.
Making the cardinality of set *Singleton* less than or equal to one gives the flexibility to instantiate *uniqueInstance* as it is declared or only when needed (lazy initialization). The later option is an advantage of using singletons rather than static class members [15].

9.2 Well-Definedness of the Pattern

A module initialization is required to establish the invariant. Class methods and initializations are only required to preserve the invariant. Methods *GetSingletonData*, and *SingletonOperation* represent functionalities to be decided by pattern instances. They are not involved in proofs. The empty class constructor is also not involved in proofs.

Module Initialization Establishes Invariant

wlp(*Singleton* $:= \{\ \}, P$)

$\equiv$ $\ll$ **wlp** *of* *Singleton* $:= \{\ \}$, *rule* $(3.4) \gg$
($|$ *Singleton* $| \leq 1$) [*Singleton* $\setminus \{\ \}$]

$\equiv$ $\ll$ **substitution** $\gg$

$(|\ \{\ \}\ | \leq 1)$

$\equiv$ $\ll$ **logic** $\gg$
true

Instance Preserves Invariant

The statement **assert** *this* $\in$ *Singleton* is added to *Instance* due to the translation from a method to a procedure.

wlp(*Singleton.Instance*, *P*)

$\equiv$ $\ll$ **definition** *of Singleton.Instance* $\gg$
wlp(**assert** *this* $\in$ *Singleton*;
if *uniqueInstance* $=$ *nil* **then** *uniqueInstance* $:=$ **new** *Singleton*;
Singleton $:= \{$*uniqueInstance*$\}$; **return** *uniqueInstance*
, | *Singleton* $| \leq 1$)

$\Leftarrow$ $\ll$ **wlp** *of* **return** *uniqueInstance*, *rule* (3.11), **substitution** $\gg$
wlp(**assert** *this* $\in$ *Singleton*;
if *uniqueInstance* $=$ *nil* **then** *uniqueInstance* $:=$ **new** *Singleton*;
Singleton $:= \{$*uniqueInstance*$\}$
, | *Singleton* $| \leq 1$)

$\Leftarrow$ $\ll$ **wlp** *of Singleton* $:= \{$*uniqueInstance*$\}$, *rule* (3.4), **substitution** $\gg$
wlp(**assert** *this* $\in$ *Singleton*;
if *uniqueInstance* $=$ *nil* **then** *uniqueInstance* $:=$ **new** *Singleton*
, | **{*uniqueInstance*}** $| \leq 1$)

$\equiv$ $\ll$ **logic**, *the cardinality of a set of one element* $= 1$ $\gg$
wlp(**assert** *this* $\in$ *Singleton*;
if *uniqueInstance* $=$ *nil* **then** *uniqueInstance* $:=$ **new** *Singleton*
, ***true***)

$\equiv$ $\ll$ **wlp** *of any statement with respect to true* $\equiv$ *true* $\gg$

true

9.3 First Instance

A simple program in which we need to have only one instance of class *PrintSpooler* to handle print jobs [5].

Instance Source

Sources are given only for the parts that appear in the instance formal description. Parts that are specific to the application are left out.

PrintSpooler

```
public class PrintSpooler {
    private static PrintSpooler spooler;
    private PrintSpooler () {
    }
    public static synchronized PrintSpooler getSpooler () {
        if (spooler == null)
            spooler =  new PrintSpooler ();
        return spooler;
    }
    public void print(String s) {
        System.out.println(s);
    }
}
```

Client

```
public class finalSpool {
    public finalSpool () {
        PrintSpooler spl = PrintSpooler.getSpooler () ;
        spl.print ("Printing data");
    }
    static public void main(String argv[]) {
        new finalSpool ();
```

```
    }
}
```

Instance Description

class *PrintSpooler*
static attr *spooler* : *PrintSpooler* := *nil*
private initialization
...
static method *getSpooler* : *PrintSpooler*
begin
if (*spooler* = *nil*) **then** *spooler* := **new** *PrintSpooler*;
return *spooler*
end
method *print*(*s* : *String*)
System.out.println(*s*)
end

class *finalSpool*
initialization
var *spl* : *PrintSpooler* •
begin
spl := *PrintSpooler.getSpooler*;
spl.print("*Printingdata*")
end
end

Data Structures

PrintSpooler : **set of** *Object*

Instance Invariant $P1$

$| \mathit{PrintSpooler} | \leq 1$

Invariant $P1$ Preservation
Invariant $P1$ is identical to pattern invariant P, except for identifiers. Invariant $P1$ is preserved by this instance. This is done by making the only constructor of class *Singleton* private. Therefore, the only way to create an instance of this class is through a call to method *getSpooler*. Such a call will return *spooler*, the only instance of class *PrintSpooler*, which was declared as a static attribute.

9.4 Second Instance

Another simple program in which we need to have only one instance of a class. The only instance of class *MySingleton* is *_instance*. It is instantiated at class-loading time [15].

Instance Source
Sources are given only for the parts that appear in the instance formal description. Parts that are specific to the application are left out.

MySingleton

```
public class MySingleton {
private static MySingleton _instance =
new MySingleton ();
private MySingleton () {
// construct object . . .
}
public static MySingleton getInstance () {
return _instance;
}
// Remainder of class definition . . .
```

Instance Description

class *MySingleton*
static *MySingleton _instance* := **new** *MySingleton*
private initialization

```
    ...
  static method getInstance : MySingleton
    return _instance
end
```

Notes:

- In this case, *_instance* is declared and initialized in the same line.

Data Structures

MySingleton : **set of** *Object*

Instance Invariant $P2$

$$\mid MySingleton \mid \; = 1$$

Invariant $P2$ **Preservation**
Since $\mid Singleton \mid \; = 1 \Rightarrow \mid Singleton \mid \; \leq 1$, then clearly $P2 \Rightarrow P$.
The instance preserves the invariant the same way as in the first instance introduced above. The only difference is that the only instance of the class is initialized with the declaration vs. when needed.
It was not needed to introduce any other instances of the pattern, this is because most implementing programs follow closely one of the two introduced instances.

Chapter 10

Future Work

Limitations of the Approach

Procedures introduced in this study look after most of the issues related to using natural language to describe patterns [19].

Yet, the approach needs to make more balance between the structural side and the behavioral side of a pattern [32]. A balanced formal language could also describe frameworks in the same abstract notation.

A pattern formalization language should also be the basis for creating tools to create and verify the correctness of patterns in a design [2]. The use of such tools can address the usual problem that design patterns are usually used to create code, but then they are forgotten. Patterns within the code are not maintained as code changes [31].

It is also noted that many patterns use basic delegation, encapsulation and other repeated concepts. This suggests factoring out these concepts as smaller building blocks to describe patterns [29].

Suggested Features

- To design a formal language that can describe patterns in terms of pre-defined structures. Such language can cover all types of patterns in a consistent representation.

- Extend the formalization approach to include frameworks such as *Java Swing* and *JHotDraw*.

- To use the grammar of the formal description language to create a tool to manipulate patterns, such that code is always associated with its underlying pattern.

Chapter 11

Conclusion

The study introduces a formal approach to choose and apply design patterns. It is shown how dealing with patterns becomes more precise when it is guided by a formal approach. The same uniform notation is used to describe patterns and their instances. Examples given clearly show that comparing different patterns or comparing a pattern with an instance is much easier and more precise than when done using informal approaches.

The study also introduces a way of checking compliance of an instance with a pattern, a task that can be challenging without following a formal approach.

Pattern descriptions are general enough as they are constructed after analyzing carefully selected instances. Produced descriptions include structural and behavioral aspects of patterns.

The applicability of the process introduced in this study is not limited to design patterns. It can also be extended to deal with frameworks and concurrent systems. Frameworks can be described in the same introduced notation. Concurrency may be expressed by extending classes with actions and allowing methods to be guarded [28]. Concurrent systems can be described in the same introduced notation. All methods and actions of a concurrent system will have to preserve the invariant if one exists.

Patterns are classified in the study based on components needed to describe them. They are classified into structure-based patterns, behavior-based patterns, and invariant-based patterns. In case of invariant-based patterns, invariants are introduced to complement the description. It is shown that these invariants capture restrictions that are implicit otherwise.

The process of establishing a pattern description gives much better insight about the

essence and details of the pattern. Such insight can ensure much better use of the pattern in a design.

Analyzing *Iterator* design pattern demonstrates the complete process of describing a pattern, verification of the description, and checking compliance of an instance with the pattern.

It is shown that the need for an invariant is not limited to one pattern. The introduced invariants for *Abstract Factory*, *Composite*, and *Singleton* are based on analyzing instances from diversified sources. The weakest possible invariants are given, such that they are not specific to an application. Instances can have stronger invariants than the pattern invariant.

Glossary

$[\,]$ An empty bag (also known as multi-set).

$\langle\rangle$ An empty sequence.

$\{\}$ An empty set.

$A \cup B$ The union of bags A and B.

$A - B$ The subtraction of bag B from bag A, or the subtraction of sequence B from sequence A. Sequence subtraction $A - B$ removes all the occurrences of any element in B from A.

$A \,\&\, B$ The concatenation of two sequences A and B.

$\mathit{length}(A)$ The length of sequence A.

$|A|$ The cardinality of set A.

$x :\in s$ The nondeterministic assignment $x :\in s$ assigns to x any element from set, bag or sequence s as long as at least one element exists, otherwise, x is assigned the value *nil*.

$x :\in s$ The nondeterministic assignment $x :\notin s$ assigns to x any element such that this element is not in s.

R^+ The transitive closure of a binary relation R. It is defined to be the set of pairs (u, v) such that there is a path of length one or more from u to v.

$P\,[x \setminus e]$ An expression denoting P, where every occurrence of x is replaced by the

value e.

$(a; x : e)$ Container a where the element at position x is replaced by the value e.

wp For a system denoted by S and having a desired post-condition denoted by R, we denote the corresponding weakest pre-condition by $wp(S, R)$. It means that if the initial state satisfies $wp(S, R)$, the system is certain to establish eventually the truth of R.

wlp The weakest liberal precondition $wlp(S, R)$ is weaker than $wp(S, R)$ defined above. $wlp(S, R)$ only guaranties that the system will not produce the wrong result, i.e. will not reach a final state not satisfying R, but nontermination is left as an alternative.

Bibliography

[1] J. Abrial, *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[2] P. Alencar, D. Cowan, K. Lichtner, C. Lucena, and L. Nova, "Tool support for formal design patterns," Technical Report CS-9536, University of Waterloo, Waterloo, Ontario, Canada, 1995.

[3] D. Alur, D. Malks, and J. Crupi, *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2001.

[4] G. Aranda and R. Moore, "Gof creational patterns: A formal specification," Tech. Rep. 224, UNU/IIST, Macau, December 2000.

[5] J. W. Cooper, *Java Design Patterns: A Tutorial*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[6] A. Cowan, "Foundations for design pattern application." ftp://csg.uwaterloo.ca/pub/ADV/theory/japan95.ps.gz, available from: citeseer.nj.nec.com/120598.html.

[7] J. Davis and J. Woodcock, *Using Z : Specification, Refinement, and Proof*. Prentice Hall, 1996.

[8] E. W. Dijkstra and W. H. Feijen, *A Method of Programming*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[9] E. W. Dijkstra and C. S. Scholten, *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., 1990.

[10] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall PTR, 1997.

[11] D. Duego, ed., *Java Gems: Jewels from Java Report*. New York: SIGS Books, 1997.

[12] E Gamma, R Helm, R Johnson and J Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1986.

[13] A. H. Eden and A. Yehudai, "Patterns of the agenda," in *Object-Oriented Technology: ECOOP'97 Workshop Reader* (J. Bosch and S. Mitchell, eds.), vol. 1357 of *Lecture Notes in Computer Science*, pp. 100–104, Springer, 1997. Workshop on Language Support for Design Patterns and Frameworks.

[14] A. Flores, R. Moore, and L. Reynoso, "A formal model of object-oriented design and GoF design patterns," *Lecture Notes in Computer Science*, vol. 2021, 2001.

[15] J. Fox, "When is a Singleton not a Singleton," January 2001. http://www.javaworld.com/javaworld/jw-01-2001/jw-0112-singleton.html.

[16] E. Gamma, "Applying design patterns in java," *Java Report*, vol. 1, no. 6, Nov./Dec. 1996.

[17] E. Gamma and T. Eggenschwiler, "JHotDraw as an Open-Source Project." http://www.jhotdraw.org.

[18] D. Geary, "Amaze your developer friends with design patterns," October 2001. http://www.javaworld.com/javaworld/jw-10-2001/jw-1012-designpatterns.html.

[19] G. Hedin, "Language support for design patterns using attribute extension," in *Object-Oriented Technology: ECOOP'97 Workshop Reader* (J. Bosch and S. Mitchell, eds.), vol. 1357 of Lecture Notes in Computer Science, pp. 137–140, Springer, 1997. Workshop on Language Support for Design Patterns and Frameworks.

[20] A. Jaiman, "Abstract factory pattern." http://www.dotnetextreme.com/articles /abstractfactory.asp.

[21] W. Kaiser, "Become a programming picasso with jhotdraw," February 2001. http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw.html.

[22] K. Lano, J. Bicarregui, and S. Goldsack, "Formalising design patterns," in *1st BCS-FACS Northern Formal Methods Workshop, Ilkley, UK* (D. Duke and A. Evans, eds.), Electronic Workshops in Computing, Springer-Verlag, 1996.

[23] N. Lévy, F. Losavio, and A. Matteo, "Comparing architectural styles: Broker specializes mediator," in *ISAW '98: Proceedings of the Third International Workshop on Software Architecture*, pp. 93–96, 1998.

[24] Microsoft Corporation, http://msdn.microsoft.com/net/ecma/, *ECMA and ISO/IEC C# and Common Language Infrastructure Standards*, December 2001.

[25] A. Mikhajlova and E. Sekerinski, "Ensuring Correctness of Java Frameworks: A Formal Look at JCF," Technical Report TUCS-TR-250, Turku Centre for Computer Science, Finland, Mar. 16, 1999.

[26] T. Mikkonen, "Formalizing design patterns," in *Proceedings of the 1998 International Conference on Software Engineering*, pp. 115–124, IEEE Computer Society Press / ACM Press, 1998.

[27] M. Ohtsuki, J. Segawa, N. Yoshida, and A. Makinouchi, "Structured Document Framework for Design Patterns Based on SGML," in *Proceedings, the Twenty-First Annual International Computer Software & Applications Conference (COMPSAC'97), August 13–15, 1997, Washington, DC*, pp. 320–323, IEEE Computer Society Press, 1997.

[28] E. Sekerinski, "Concurrent object-oriented programs: From specification to code," in *First International Symposium on Formal Methods for Components and Objects, FMCO 02, Lecture Notes in Computer Science 2852*, (Leiden, The Netherlands), pp. 403–423, Springer-Verlag, 2003.

[29] J. Smith and D. Stotts, "Elemental design patterns: A formal semantics for composition of oo software architecture," in *27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02) December 05 - 06, 2002 Greenbelt, Maryland*, p. 183, IEEE Computer Society Press, 2002.

[30] R. Software, Microsoft, Hewlett-Packard, Oracle, S. Software, M. Systemhouse, Unisys, I. Computing, IntelliCorp, i Logix, IBM, ObjecTime, P. Technology, Ptech, Taskon, R. Technologies, and Softeam, *Object Constraint Language Specification (version 1.1)*. Rational Software Corporation, Sept. 1997.

[31] Steven P. Reiss , "Working with patterns and code," in *33rd Hawaii International Conference on System Sciences-Volume 8 January, 2000 Maui, Hawaii*, p. 8054, IEEE Computer Society Press, 2000.

[32] Toufik Taibi, David Check Ling Ngo, "Formal Specification of Design Patterns - A Balanced Approach," *in Journal of Object Technology*, vol. 2, no. 4, pp. 127–140, July-August 2003.

[33] J. Vlissides, "GoF a la Java," *Java Report*, March 2001. http://www.research.ibm.com/designpatterns/pubs/jr-mar01.pdf.

.