The 'Gang of Four' Companion

Formal specification of design patterns in LePUS3 and Class-Z

Technical Report CSM-472, ISSN 1744-8050

Amnon H Eden, Jonathan Nicholson, Epameinondas Gasparis
Department of Computer Science, University of Essex, United Kingdom

31 December 2007

See also:

- Legend: Key to LePUS3 and Class-Z Symbols [.pdf]
- The LePUS3 and Class-Z Reference Manual [.pdf]
- The 'Gang of Four' Companion: Formal specification of design patterns in LePUS3 and Class-Z [.pdf]

Download:

- The GoF patterns in LePUS3 (Visio 2003 format)

Abstract

This document demonstrates how the informal specifications of the design patterns the 'Gang of Four' seminal catalogue [Gamma et al 1995] can be specified formally using the LePUS3 and Class-Z object-oriented Architecture Description Languages.

Table of Contents:

The 'Gang of Four' Companion:

Modelling Design Patterns: Frequently-Asked Questions

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion which details the formal specification of the Abstract Factory design pattern from the 'Gang of Four' catalogue [Gamma et al 1995].

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Abstract Factory in XML

Download:

- The GoF patterns in LePUS3 (Visio 2003 format)

Table of contents

1. What are design patterns?

Design patterns are 'recipe's for solving software design problems by describing a particular *design motif*: a category of solutions to a class of common design problems. The most widely used kind of design patterns are patterns of object-oriented design: recipes which describe a category of problems common to object-oriented design and how they should be solved. The excellent 'Gang of Four' catalogue [Gamma et al. 1995] has established the widest practice of design patterns in object-oriented design. A nice introduction to design patterns is offered in [Schmidt et al. 1996] (available online).

Most often, however, a 'pattern' in the vernacular refers to the design motif element of the design pattern, not to the recipe as a whole. While some see this as a confusion, others consider this a natural by-product in the evolution of standard software design solutions to common problems, such as *architectural styles* [Garlan & Shaw 1993], *programming paradigms*, and *component-based software engineering* technologies [Szyperski 2002]. In LePUS3 and Class-Z we adopt the common practice of using the term 'design pattern' (in short, 'pattern') with reference to the design motif that the pattern's recipe describes.

2. What is so special about modelling design patterns?

Modelling languages in standard use, such as the Unified Modeling Language and its predecessors (e.g. Object Modeling Technique, Booch Notation, etc.) are focused on modelling a particular software system—a specific program (or a specific collection of programs). Design patterns however are design motifs: there is usually an unbound number of programs that implement each pattern. Modelling a design pattern is therefore an exercise in abstraction *cum* genericity: it requires the use of *variables* (as opposed to *constants*) which range over a category of implementations.

Why variables are so important? Try 'modelling' the difference between the following two statements:

1. Class `java.awt.Container` extends class `java.awt.Component`
2. A $\mathit{Composite}$ class (such as `java.awt.Container`) must extend, either directly or indirectly, the $\mathit{Component}$ class (such as `java.awt.Component`).

The first sentence describes a specific implementation, the second describes a *design pattern*—a category of implementations. The first statement determines the relation between specific classes, whereas the second statement imposes a constraint on such classes. The first statement is closed, the second is open.

3. Why the focus on the 'Gang of Four' catalogue?

The book *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma et al. 1995] was authored by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, a quartet which came to be known as the 'gang of four' (hence: the 'Gang of Four' Catalogue). The catalogue delivers a meticulous description of twenty-three design patterns. Many of the patterns described in the book have been implemented in a wide range of well-designed programs and are currently recognized as paragons of good design.

For these reasons, The 'Gang of Four' Companion is focused on modelling the design patterns in the 'Gang of Four' catalogue.

4. What's wrong with UML as a modelling language for design patterns?

Giving a full answer to this question would require not one page but an entire encyclopaedia. We are not in the business of writing the seven volumes of *What's Wrong with UML*; others have already done so successfully (see Bertrand's Meyer's excellent UML—The Positive Spin.) Rather, let us describe what mechanisms of LePUS3 and Class-Z are specifically tailored for modelling design patterns, *none of which* applies to UML:

- **Decidability & Verification**:All LePUS/Class-Z specifications are decidable and can be verified automatically by a tool support (see also: The Two-Tier Programming Project)
- **Rigour**: Each LePUS3/Class-Z specification is equivalent to a formula in first-order classical predicate calculus, and their semantics is defined using Tarski's style truth conditions in terms of finite models.
- **Abstraction viz. genericity**: LePUS/Class-Z distinguish between *variables*, which range over (elements of) implementations, and *constants*, modelling specific (elements of) implementations.
- **Abstraction viz. scalability**: LePUS3/Class-Z offer a range of abstraction mechanisms (e.g., class hierarchies, sets of methods with same signature), whereas UML Class diagrams are limited to packages and classes.
- **Abstractionviz. information neglect**: If class/method/relation x is not modelled, what does it mean: That there is no such an entity? Or that it may or may not exist? In LePUS3/Class-Z, absence of information indicates nothing, and specifications merely impose constraints on the implementation but do not dictate it.

"But UML is so popular?!"--right.

5. Can LePUS3/Class-Z model everything about patterns?

No. The syntax of LePUS3/Class-Z is extremely limited, allowing only for three kinds of predicates and only one kind of ground formula. Furthermore, only decidable relations can be expressed using LePUS3 and Class-Z. Therefore we cannot model statements such as 'each instance of class X holds exactly 3 instances of class Y at each point in time.' Objects are not modelled directly using LePUS3 and Class-Z, only classes.

Nonetheless, you would be surprised how much about patterns can be captured with these simple concepts. See here the full list of the 'Gang of Four' patterns that can be modelled in LePUS3/Class-Z.

6. Which patterns can be modelled with LePUS3/Class-Z?

See here the full list of the 'Gang of Four' patterns that can be modelled in LePUS3/Class-Z.

7. Which other modelling languages for design patterns exist?

There are many attempts to tailor formal specification languages specifically for the purpose of modelling design patterns. Many of these efforts are listed in this page.

8. Why did you convert the original diagrams to UML and how?

Each 'Structure' diagram in our 'Gang of Four Companion' includes UML Class Diagrams for illustrating the informal description, although OMT was used by the [Gamma et al. 1995] catalogue. We could not quote the original OMT diagrams due to copyright issues, and UML is currently much more popular.

UML Class diagrams were created in Visio 2003 format (download UML Class Diagrams of the 'Gang of Four' design patterns.)

The 'conversion' process is free-style for illustrative purposes only: namely to show the inadequacy of UML in modelling each pattern. Don't write us about so called 'errors' in the UML Class Diagrams; we could not care less about those because neither UML nor OMT is any good for modelling design patterns anyway.

The 'Gang of Four' Companion:

Abstract Factory Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

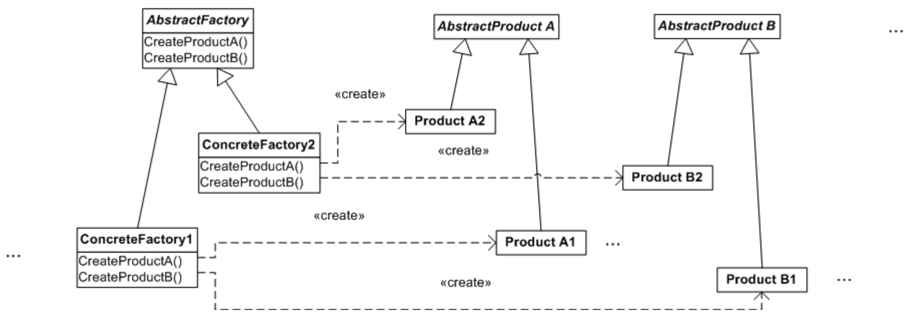
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Abstract Factory design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent:Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

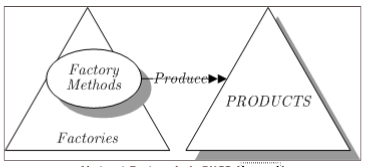
- **AbstractFactory** (WidgetFactory): declares an interface for operations that create abstract product objects.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory): implements the operations to create concrete product objects.
- **AbstractProduct** (Window, ScrollBar): declares an interface for a type of product object.
- **ConcreteProduct** (MotifWindow, MotifScrollBar): defines a product object to be created by the corresponding concrete factory; implements the AbstractProduct interface.

Collaborations: AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Abstract Factory in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (**legend**)

Abstract Factory

$Factories : \mathbb{HIERARCHY}$

$FactoryMethods : \mathbb{P}\mathrm{SIGNATURE}$

$PRODUCTS : \mathbb{P}\mathrm{HIERARCHY}$

$\textsc{Isomorphic}(Produce, FactoryMethods \otimes Factories, PRODUCTS)$

Abstract Factory in Class-Z (legend)

3. Sample Implementations

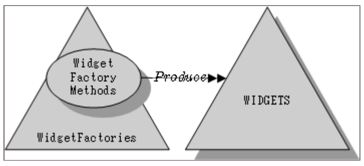
3.1. Informal description: Portable GUI Widget Factories

From [Gamma et. al 1995]:

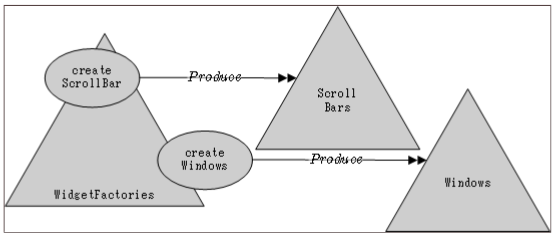
> Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons. To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel later.
>
> We can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget. There's also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards. WidgetFactory's interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients aren't aware of the concrete classes they're using. Thus clients stay independent of the prevailing look and feel.

3.2. Formal specification: Portable GUI Widget Factories

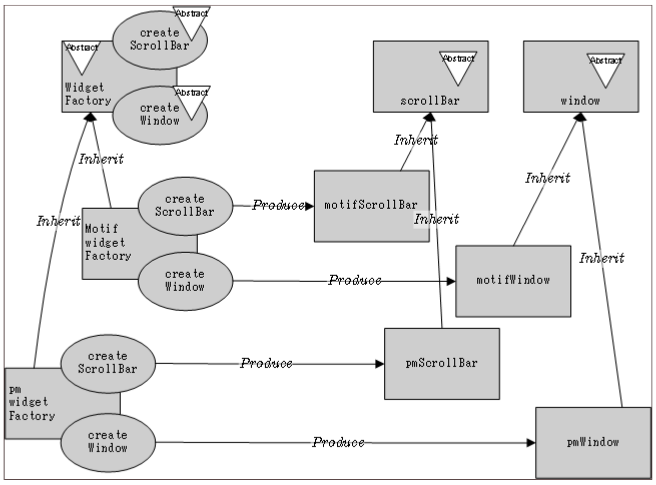
Specified in LePUS3 and Class-Z (see legend)



Abstract Factory implementation modelled in LePUS3
using 1 and 2-dimensional hierarchy constants (legend)



Abstract Factory implementation modelled in LePUS3
using 1-dimensional hierarchy constants (legend)



Abstract Factory implementation modelled in LePUS3
using 0-dimensional class and signature constants (legend)

The 'Gang of Four' Companion:

Adapter (Class) Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

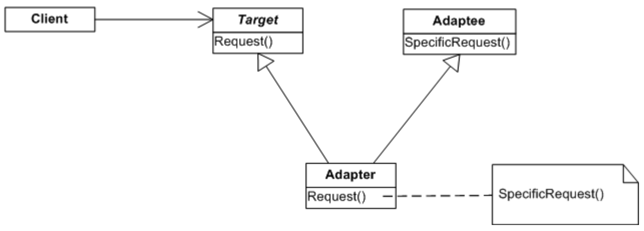
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Adapter (Class) design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

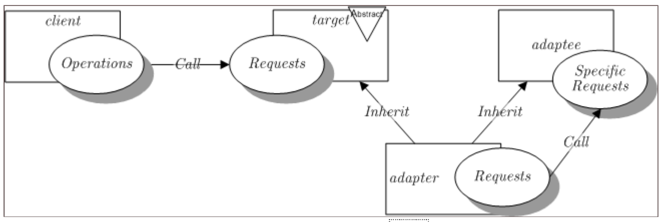
- **Target**: defines the domain-specific interface that Client uses.
- **Client**: collaborates with objects conforming to the Target interface.
- **Adaptee**: defines an existing interface that needs adapting.
- **Adapter**: adapts the interface of Adaptee to the Target interface.

Collaborations: Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Adapter (Class) in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)

Adapter (Class)

$client, target, adaptee, adapter : \mathbb{CLASS}$

$Operations, Requests, SpecificRequests : \mathbb{P}\mathbb{SIGNATURE}$

$Abstract(target)$
$Inherit(adapter, target)$

$Inherit(adapter, adaptee)$

$\mathit{TOTAL}(Call, Operations \otimes client, Requests \otimes target)$

$\mathit{TOTAL}(Call, Requests \otimes adapter, SpecificRequests \otimes adaptee)$

Abstract Factory in Class-Z (**legend**)

The 'Gang of Four' Companion:

Adapter (Object) Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

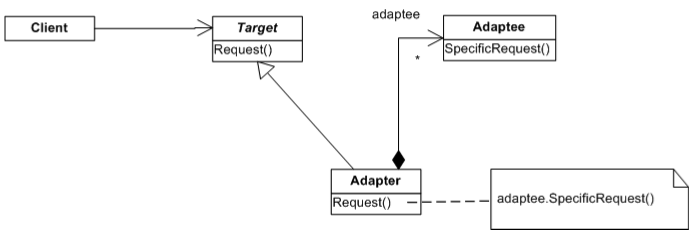
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Adapter (Object) design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

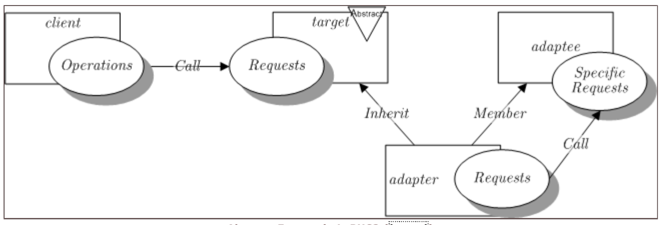
- **Target**: defines the domain-specific interface that Client uses.
- **Client**: collaborates with objects conforming to the Target interface.
- **Adaptee**: defines an existing interface that needs adapting.
- **Adapter**: adapts the interface of Adaptee to the Target interface.

Collaborations: Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Adapter (Object) in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)

Adapter (Object)

$client, target, adaptee, adapter : \mathbb{CLASS}$

$Operations, Requests, SpecificRequests : \mathcal{P}\mathbb{SIGNATURE}$

$Abstract(target)$

$Inherit(adapter, target)$
$Member(adapter, adaptee)$
$\textsc{Total}(Call, Operations \otimes client, Requests \otimes target)$
$\textsc{Total}(Call, Requests \otimes adapter, SpecificRequests \otimes adaptee)$

Abstract Factory in Class-Z (legend)

Bridge Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

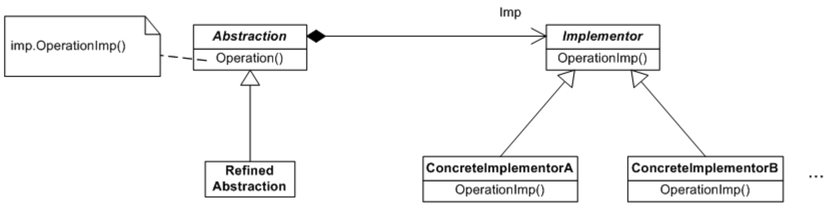
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Bridge design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Decouple an abstraction from its implementation so that the two can vary independently.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

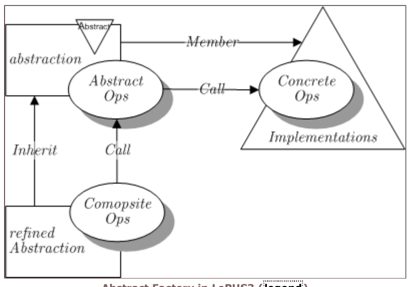
- **Abstraction**: defines the abstraction's interface and maintains a reference to an object of type Implementor.
- **RefinedAbstraction**: Extends the interface defined by Abstraction.
- **Implementor**: defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor**: implements the Implementor interface and defines its concrete implementation.

Collaborations: Abstraction forwards client requests to its Implementor object.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Bridge in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)

Bridge

$refinedAbstraction, abstraction : \mathbb{CLASS}$

$$
\begin{array}{|l}
Implementations : \mathbb{HIERARCHY} \\
CompositeOps, AbstractOps, ConcreteOps : \mathcal{P}\mathbb{SIGNATURE} \\
\hline
Abstract(abstraction) \\
Inherit(refinedAbstraction, abstraction) \\
\textit{TOTAL}(Member, abstraction, Implementations) \\
\textit{TOTAL}(Call, CompositeOps \otimes refinedAbstraction, AbstractOps \otimes abstraction) \\
\textit{TOTAL}(Call, AbstractOps \otimes abstraction, ConcreteOps \otimes Implementations) \\
\hline
\end{array}
$$

Abstract Factory in Class-Z (legend)

The 'Gang of Four' Companion:

Composite Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

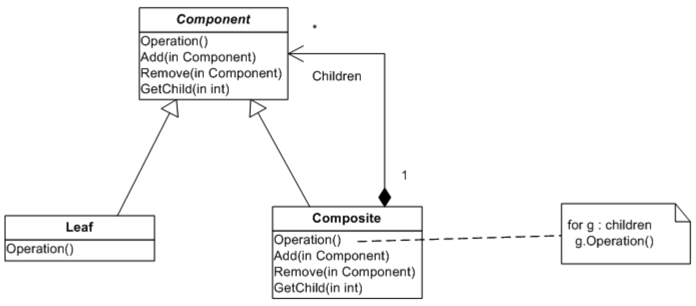
1. The Composite design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structure: Original OMT diagram converted to UML (Why and How?):

...



Participants:

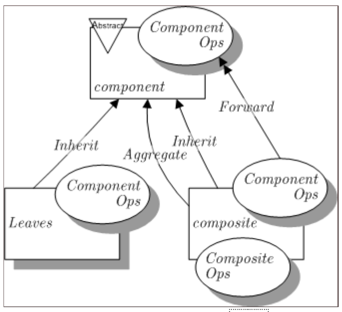
- **Component** (Graphic):
  - Declares the interface for objects in the composition.
  - Implements default behavior for the interface common to all classes, as appropriate.
  - Declares an interface for accessing and managing its child components.
- **Leaf** (Rectangle, Line, Text, etc.):
  - Represents leaf objects in the composition. A leaf has no children.
  - Defines behavior for primitive objects in the composition.
- **Composite** (Picture):
  - Defines behavior for components having children.
  - Stores child components.
  - Implements child-related operations in the Component interface.
- **Client** (Client): manipulates objects in the composition through the Component interface.

Collaborations: Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Composite in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)

Composite

$composite, component : \mathbb{CLASS}$

$Leaves : \mathbb{P}\mathbb{CLASS}$

$ComponentOps, CompositeOps : \mathbb{P}\mathbb{SIGNATURE}$

$Abstract(component)$
$Aggregate(composite, component)$
$Inherit(composite, component)$
$\textit{TOTAL}(Inherit, Leaves, component)$
$\textit{ALL}(Method, ComponentOps \otimes Leaves)$
$\textit{ALL}(Method, CompositeOps \otimes composite)$
$\textit{ISOMORPHIC}(Forward, ComponentOps \otimes composite, ComponentOps \otimes component)$

Abstract Factory in Class-Z (legend)

3. Sample Implementations

3.1. Informal description: Composite Graphics

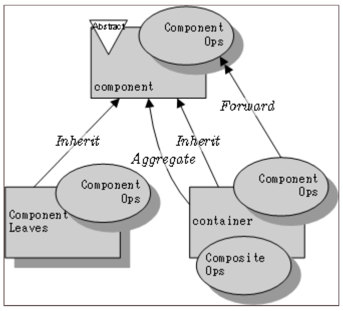
From [Gamma et. al 1995]:

> Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.
>
> But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.
>
> The key to the Composite pattern is an abstract class that represents both primitives and their containers. For the graphics system, this class is Graphic. Graphic declares operations like Draw that are specific to graphical objects.
>
> The subclasses Line, Rectangle, and Text (see preceding class diagram) define primitive graphical objects. These classes implement Draw to draw lines, rectangles, and text, respectively. Since primitive graphics have no child graphics, none of these subclasses implements child-related operations.
>
> The Picture class defines an aggregate of Graphic objects. Picture implements Draw to call Draw on its children, and it implements child-related operations accordingly. Because the Picture interface conforms to the Graphic interface, Picture objects can compose other Pictures recursively.

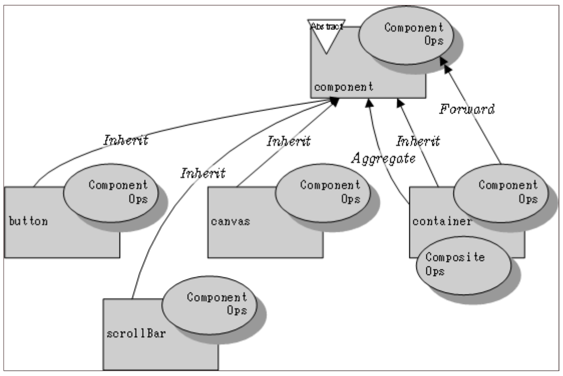
This design pattern, as informally described above, can be seen implemented within the Java Foundation Classes. Specifically within the Java Abstract Window Toolkit (AWT). How this is done is presented below, and unless otherwise stated all classes reside within the package `java.awt`.

3.2. Formal specification: Composite Graphics in `java.awt`

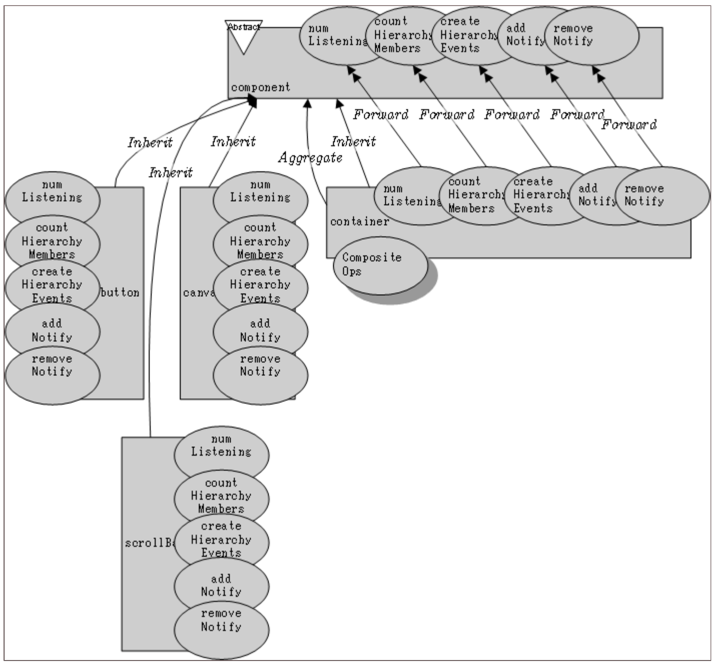
Specified in LePUS3 and Class-Z (see legend)



Composite implementation modelled in LePUS3 using 0 and 1-dimensional class (signature) constants (legend)



Composite implementation modelled in LePUS3
using 0-dimensional class and 1-dimensional signature constants (legend)



Composite implementation modelled in LePUS3
using 0-dimensional class and signature constants (legend)
[1-dimensional signature constant CompositeOps left unexpanded for simplicity]

The 'Gang of Four' Companion:

Decorator Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

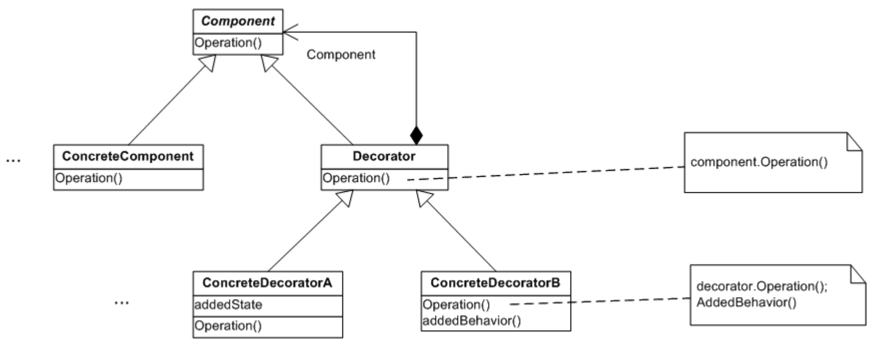
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Decorator design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

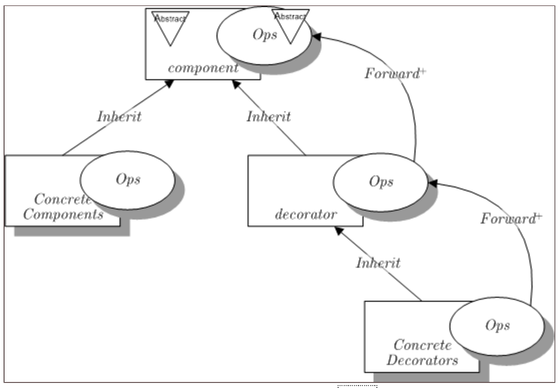
- **Component**: defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent**: defines an object to which additional responsibilities can be attached.
- **Decorator**: maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**: adds responsibilities to the component.

Collaborations: Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Decorator in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)

Decorator

$component, decorator : \mathbb{CLASS}$

$ConcreteDecorators, ConcreteComponents : \mathcal{P}\mathbb{CLASS}$

$Ops : \mathcal{P}\mathbb{SIGNATURE}$

$Abstract(component)$

$\textit{ALL}(Abstract, Ops \otimes component)$

$Inherit(decorator, component)$

$\textit{TOTAL}(Inherit, ConcreteComponents, component)$

$\textit{TOTAL}(Inherit, ConcreteDecorators, decorator)$

$\textit{TOTAL}(Forward^{+}, Ops \otimes ConcreteDecorators, Ops \otimes decorator)$

$\textit{TOTAL}(Forward^{+}, Ops \otimes decorator, Ops \otimes component)$

$\textit{ALL}(Method, Ops \otimes ConcreteComponents)$

Abstract Factory in Class-Z (legend)

The 'Gang of Four' Companion:

Factory Method Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

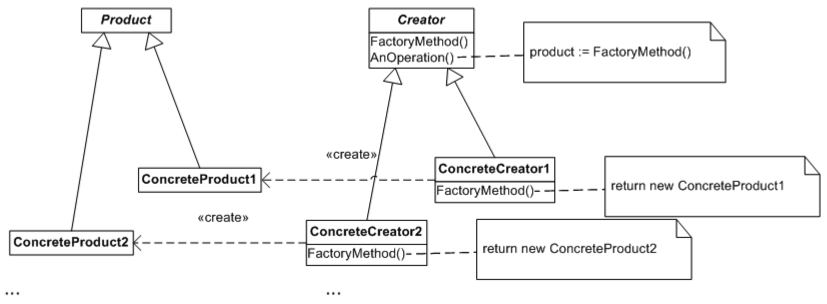
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Factory Method design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

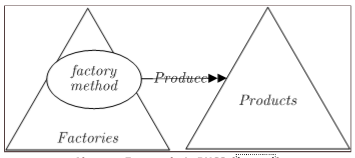
- **Product**: defines the interface of objects the factory method creates.
- **ConcreteProduct**: implements the Product interface.
- **Creator**: declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- **ConcreteCreator**: overrides the factory method to return an instance of a ConcreteProduct.

Collaborations: Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Factory Method in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)

Factory Method

$Factories, Products : \mathbb{HIERARCHY}$

$factoryMethod : \mathbb{SIGNATURE}$

$\textsc{Isomorphic}(Produce, factoryMethod \otimes Factories, Products)$

Abstract Factory in Class-Z (legend)

The 'Gang of Four' Companion:

Flyweight Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

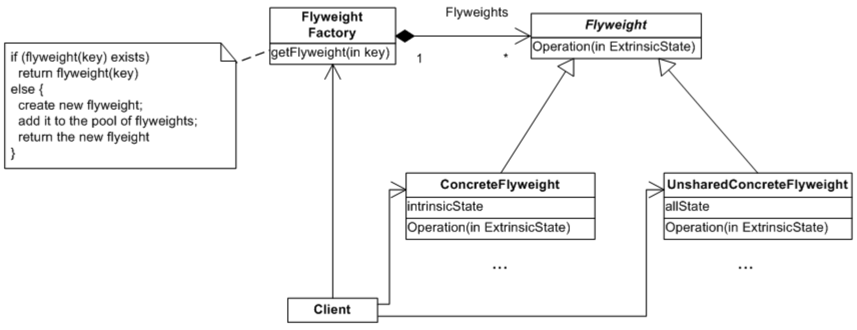
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Flyweight design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Use sharing to support large numbers of fine-grained objects efficiently.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

- **Flyweight**: declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight**: implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight**: not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- **FlyweightFactory**: creates and manages flyweight objects, ensuring that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
- **Client**: maintains a reference to flyweight(s), and computes (or stores) their extrinsic state.

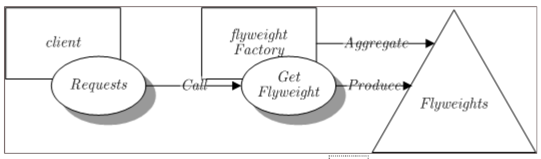
Collaborations:

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic. Intrinsic state is stored in the ConcreteFlyweight object; extrinsic state is stored or computed by Client objects. Clients pass this state to the flyweight when they invoke its operations.
- Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Flyweight in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (**legend**)

Flyweight

$client, flyweightFactory : \mathbb{CLASS}$

$Flyweights : \mathbb{HIERARCHY}$

$Requests, GetFlyweight : \mathcal{P}\mathbb{SIGNATURE}$

$\textsc{Total}(Call, Requests \otimes client, GetFlyweight \otimes flyweightFactory)$

$\textsc{Total}(Produce, GetFlyweight \otimes flyweightFactory, Flyweights)$

$\textsc{Total}(Aggregate, flyweightFactory, Flyweights)$

Abstract Factory in Class-Z (legend)

The 'Gang of Four' Companion:

Iterator Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

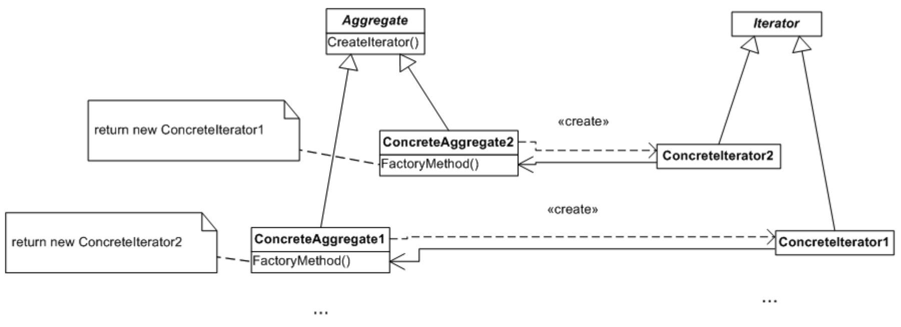
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Iterator design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

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

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

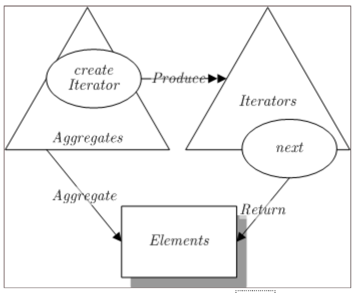
- **Iterator**: defines an interface for accessing and traversing elements.
- **ConcreteIterator**: implements the Iterator interface and keeps track of the current position in the traversal of the aggregate.
- **Aggregate**: defines an interface for creating an Iterator object.
- **ConcreteAggregate**: implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Collaborations: A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Iterator in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (**legend**)

Iterator

$Elements : \mathbb{P}\mathsf{CLASS}$

$Aggregates, Iterators : \mathsf{HIERARCHY}$

$createIterator, next : \mathsf{SIGNATURE}$

$\mathit{ISOMORPHIC}(Produce, createIterator \otimes Aggregates, Iterators)$

$\mathit{TOTAL}(Return, next \otimes Iterators, Elements)$

$\mathit{TOTAL}(Aggregate, Aggregates, Elements)$

Abstract Factory in Class-Z (legend)

3. Sample Implementations

3.1. Informal description: Collections and Iterators

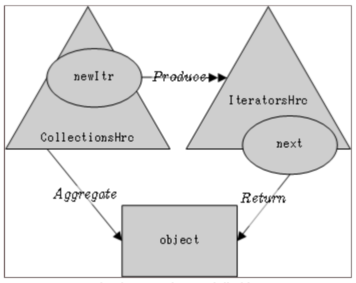
From [Gamma et. al 1995]:

> An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.
>
> The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.
>
> Notice that the iterator and the list are coupled, and the client must know that it is a *list* that's traversed as opposed to some other aggregate structure. Hence the client commits to a particular aggregate structure. It would be better if we could change the aggregate class without changing client code. We can do this by generalizing the iterator concept to support **polymorphic iteration**.
>
> We define an AbstractList class that provides a common interface for manipulating lists. Similarly, we need an abstract Iterator class that defines a common iteration interface. Then we can define concrete Iterator subclasses for the different list implementations. As a result, the iteration mechanism becomes independent of concrete aggregate classes.

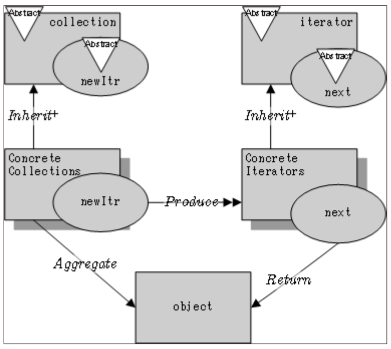
This design pattern, as informally described for lists above, can be seen implemented within the Java Collections framework. How this is done is presented below, and unless otherwise stated all classes reside within the package `java.util`.

3.2. Formal specification: Collections and Iterators

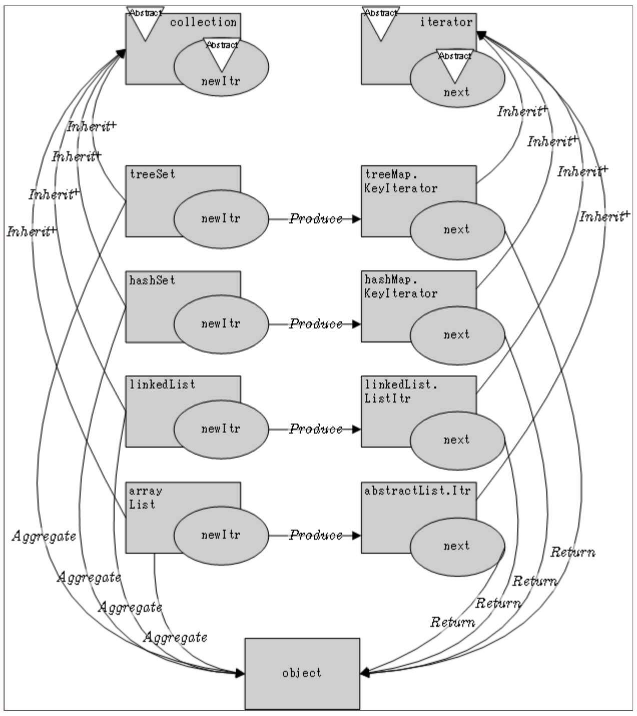
Specified in LePUS3 and Class-Z (see legend)



**Iterator implementation modelled in LePUS3
using 1-dimensional hierarchy constants (legend)**



**Iterator implementation modelled in LePUS3
using 1-dimensional class constants (legend)**



Iterator implementation modelled in LePUS3
using 0-dimensional class (signature) constants (legend)

The 'Gang of Four' Companion:

Observer Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

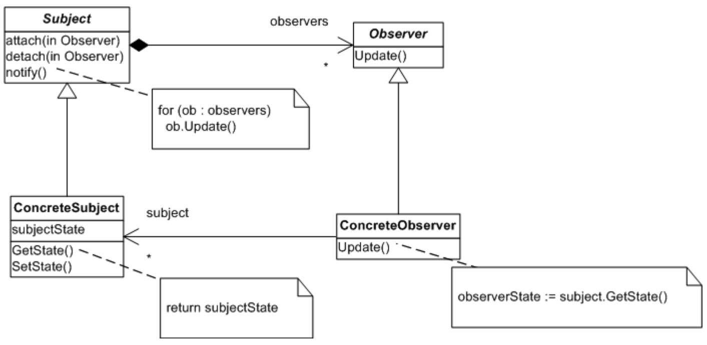
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Observer design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

- **Subject**:
  - Knows its observers. Any number of Observer objects may observe a subject.
  - Provides an interface for attaching and detaching Observer objects.
- **Observer**: defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**:
  - Stores state of interest to ConcreteObserver objects.
  - Sends a notification to its observers when its state changes.
- **ConcreteObserver**:
  - Maintains a reference to a ConcreteSubject object.
  - Stores state that should stay consistent with the subject's.
  - Implements the Observer updating interface to keep its state consistent with the subject's.

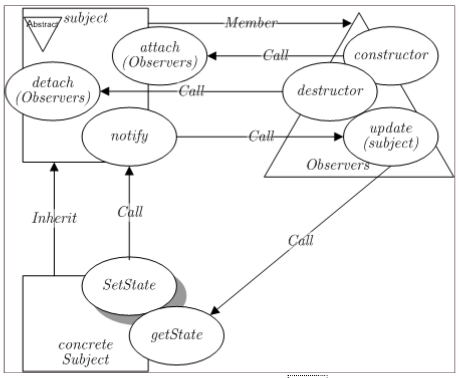
Collaborations:

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Observer in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)

Observer

$subject, concreteSubject : \mathbb{CLASS}$

$Observers : \mathbb{HIERARCHY}$

$getState, notify, attach(Observers), detach(Observers) : \mathbb{SIGNATURE}$

$constructor, destructor, update(subject) : \mathbb{SIGNATURE}$

$SetState : \mathcal{P}\mathbb{SIGNATURE}$

$Abstract(subject)$

$Inherit(concreteSubject, subject)$

$\textsc{Total}(Member, subject, Observers)$

$\textsc{Total}(Call, SetState \otimes concreteSubject, notify \otimes subject)$

$\textsc{Total}(Call, notify \otimes subject, update(subject) \otimes Observers)$

$\textsc{Total}(Call, update(subject) \otimes Observers, getState \otimes concreteSubject)$

$\textsc{Total}(Call, destructor \otimes Observers, detach(Observers) \otimes subject)$

$\textsc{Total}(Call, constructor \otimes Observers, attach(Observers) \otimes subject)$

Abstract Factory in Class-Z (legend)

Proxy Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

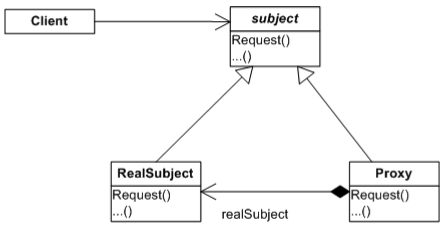
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Proxy design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Provide a surrogate or placeholder for another object to control access to it.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

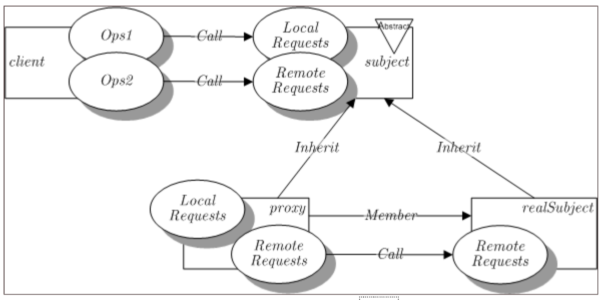
- **Proxy**:
  - Maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - Provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
  - Controls access to the real subject and may be responsible for creating and deleting it.
  - Other responsibilities depend on the kind of proxy:
    - Remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - Virtual proxies may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
    - Protection proxies check that the caller has the access permissions required to perform a request.
- **Subject**: defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject**: defines the real object that the proxy represents.

Collaborations: Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Proxy in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)

Proxy

$client, subject, proxy, realSubject : \mathbb{CLASS}$

$Ops1, Ops2, LocalRequests, RemoteRequests : \mathcal{P}\mathbb{SIGNATURE}$

$Abstract(subject)$

$Inherit(proxy, subject)$

$Inherit(realSubject, subject)$

$\textsc{Total}(Call, Ops1 \otimes client, LocalRequests \otimes subject)$

$\textsc{Total}(Call, Ops2 \otimes client, RemoteRequests \otimes subject)$

$\textsc{Total}(Call, RemoteRequests \otimes proxy, RemoteRequests \otimes realSubject)$

$\textsc{All}(Method, LocalRequests \otimes proxy)$

Abstract Factory in Class-Z (legend)

The 'Gang of Four' Companion:

State Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

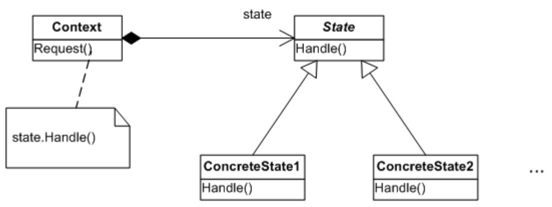
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The State design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

- **Context**: defines the interface of interest to clients, and maintains an instance of a ConcreteState subclass that defines the current state.
- **State**: defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **ConcreteState subclasses**: each subclass implements a behavior associated with a state of the Context.

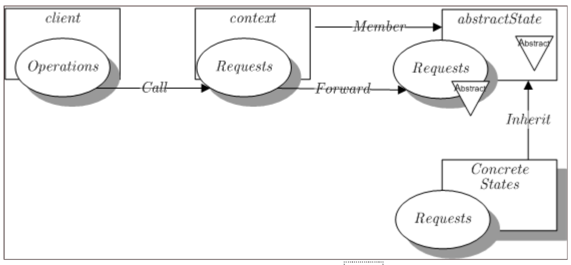
Collaborations:

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- State in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)

State

$client, context, abstractState : \mathbb{CLASS}$

$ConcreteStates : \mathcal{P}\mathbb{CLASS}$

$Operations, Requests : \mathcal{P}\mathbb{SIGNATURE}$

$Abstract(abstractState)$

$\textsc{All}(Abstract, Requests \otimes abstractState)$

$Member(context, abstractState)$

$\textsc{Total}(Inherit, concreteStates, abstractState)$

$\textsc{Total}(Call, Operations \otimes client, Requests \otimes context)$

$\textsc{Total}(Forward, Requests \otimes context, Requests \otimes abstractState)$

$\textsc{All}(Method, Requests \otimes ConcreteStates)$

Abstract Factory in Class-Z (legend)

The 'Gang of Four' Companion:

Strategy Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

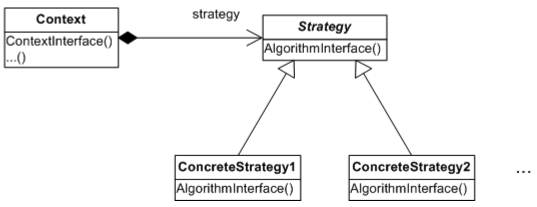
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Strategy design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

- **Strategy**: declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy**: implements the algorithm using the Strategy interface.
- **Context**:
  - Is configured with a ConcreteStrategy object.
  - Maintains a reference to a Strategy object.
  - May define an interface that lets Strategy access its data.

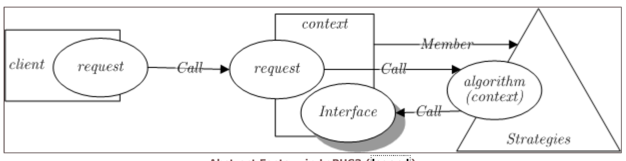
Collaborations:

- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

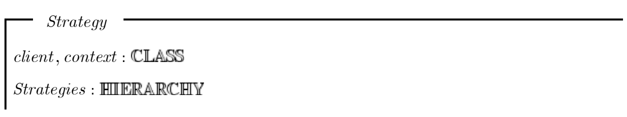
2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Strategy in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (legend)



$client, context : \mathbb{CLASS}$

$Strategies : \mathbb{HIERARCHY}$

$request, algorithm(context) : \mathbb{SIGNATURE}$

$Interface : \mathcal{P}\mathbb{SIGNATURE}$

$Call(request \otimes client, request \otimes context)$

$\textsc{Total}(Call, request \otimes context, algorithm(context) \otimes Strategies)$

$\textsc{Total}(Call, algorithm(context) \otimes Strategies, Interface \otimes context)$

$\textsc{Total}(Member, context, Strategies)$

Abstract Factory in Class-Z (legend)

The 'Gang of Four' Companion:

Template Method Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

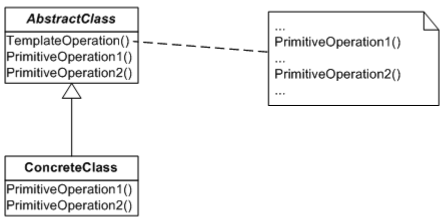
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Template Method design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Structure: Original OMT diagram converted to UML (Why and How?):



Participants:

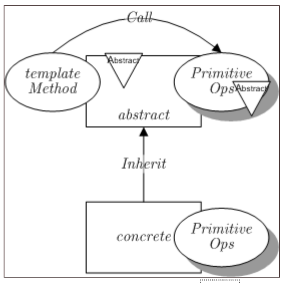
- **AbstractClass**:
  - Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
  - Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass**: implements the primitive operations to carry out subclass-specific steps of the algorithm.

Collaborations: ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Template Method in XML
- Modelling Design Patterns: Frequently-Asked Questions



Abstract Factory in LePUS3 (**legend**)

Template Method

$abstract, concrete : \mathbb{CLASS}$

$templateMethod : \mathbb{SIGNATURE}$

$PrimitiveOps : \mathbb{P}\,\mathbb{SIGNATURE}$

$Abstract(abstract)$

$\textit{ALL}(Abstract, PrimitiveOps \otimes abstract)$

$Inherit(concrete, abstract)$

$\textit{TOTAL}(Call, templateMethod \otimes abstract, PrimitiveOps \otimes abstract)$

$\textit{ALL}(Method, PrimitiveOps \otimes concrete)$

Abstract Factory in Class-Z (legend)

The 'Gang of Four' Companion:

Visitor Design Pattern

Formal specification of design patterns in LePUS3 and Class-Z

This page is part of the The 'Gang of Four' Companion dedicated to the formal specification in LePUS3 and Class-Z of patterns from the 'Gang of Four' catalogue [Gamma et al 1995].

Table of contents

Links

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Modelling Design Patterns: Frequently-Asked Questions

Download:

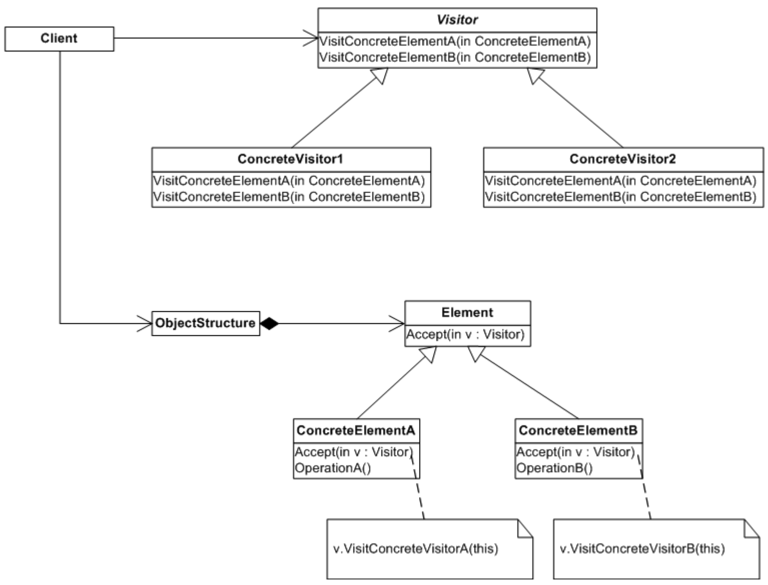
- The 'gang of four' patterns in LePUS3 (Visio 2003 format)
- The 'gang of four' patterns in UML (Visio 2003 format)

1. The Visitor design motif

The informal description: Excerpts from [Gamma et al. 1995] (adapted for this purpose):

Intent: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Structure: Original OMT diagram converted to UML (Why and How?):


Client
Visitor
VisitConcreteElementA(in ConcreteElementA)
VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(in ConcreteElementA)
VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(in ConcreteElementA)
VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
Accept(in v : Visitor)
ConcreteElementA
Accept(in v : Visitor)
OperationA()
ConcreteElementB
Accept(in v : Visitor)
OperationB()
v.VisitConcreteVisitorA(this)
v.VisitConcreteVisitorB(this)

Participants:

- **Visitor**: declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.
- **ConcreteVisitor**: implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- **Element**: defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement**: implements an Accept operation that takes a visitor as an argument.
- **ObjectStructure**:
  - Can enumerate its elements.
  - May provide a high-level interface to allow the visitor to visit its elements.
  - May either be a composite or a collection such as a list or a set.

Collaborations:

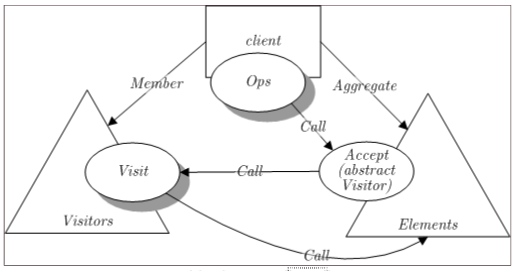
- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

2. Formal specification

See also:

- Legend: Key to LePUS3 and Class-Z Symbols
- LePUS3 & Class-Z Reference Manual
- Visitor in XML

- Modelling Design Patterns: Frequently-Asked Questions



Visitor in LePUS3 (legend)

Visitor

$client : \mathbb{CLASS}$

$Visitors, Elements : \mathbb{HIERARCHY}$

$accept(abstractVisitor) : \mathbb{SIGNATURE}$

$Visit, Ops : \mathcal{P}\mathbb{SIGNATURE}$

$\textsc{Total}(Aggregate, client, Elements)$

$\textsc{Total}(Member, client, Visitors)$

$\textsc{Total}(Call, Ops \otimes client, accept(abstractVisitor) \otimes Elements)$

$\textsc{Total}(Call, accept(abstractVisitor) \otimes Elements, Visit \otimes Visitors)$

$\textsc{Total}(Call, Visit \otimes Visitors, Elements)$

Visitor in Class-Z (legend)