Praktikum

Software-Architektur

J. Reichardt

Inhalt

- Praktikumsversuche zu den Entwurfsmustern *Composite*, *Iterator* und *Visitor*
- FracTool-Clickzonen und Legende
- Einführung in die Entwurfsmuster

SOFTWARE-ARCHITEKTUR

Praktikum

1. Übung (Composite-Muster)

a. Definiere als Knotentypen eines Abstrakten Syntaxbaumes Composite-Klassen zur Darstellung einfacher und zusammengesetzter arithmetischer Ausdrücke.

b. Instanziiere und verkette Composite-Objekte zur Darstellung des folgenden arithmetischen Ausdrucks:

(((a+b)*(a-c))+((b*d)-a)).

c. Drucke den obigen Ausdruck mittels rekursiver Elementfunktionen der Composite-Klassen aus.

d. Werte den obigen Ausdruck mittels rekursiver Elementfunktionen der Composite-Klassen für eine beliebige Wertebelegung der Variablen aus.

e. Führe die Übung als C++-, Java- oder C#-Programm aus.

f. Führe die Übung als visuelles Fractool-Programm aus.

SOFTWARE-ARCHITEKTUR

Praktikum

2. Übung (Iterator-Muster)

a. Definiere die Print- und Evaluate-Methoden der 1. Übung als Elementfunktionen eines Iterators. Aus der Schnittstelle der Composite-Klassen sind die Print- und Evaluate-Methoden zu entfernen.

b. Drucke den arithmetischen Ausdruck der 1. Übung mittels eines internen Iterators aus.

c. Werte den arithmetischen Ausdruck für eine beliebige Wertebelegung mittels eines internen Iterators aus.

d. Führe die Übung als C++-, Java- oder C#-Programm aus.

e. Führe die Übung als visuelles Fractool-Programm aus.

SOFTWARE-ARCHITEKTUR

Praktikum

3. Übung (Visitor-Muster)

a. Die Print- und Evaluate-Methoden sind in Visitor-Klassen auszulagern.

b. Die Composite-Klassen sollen im Wesentlichen nur Accept-Methoden besitzen.

c. Die Traversen für die Print- und Evaluate-Funktionen sind in Iteratoren auszulagern.

d. Führe die Übung als C++-, Java- oder C#-Programm aus.

e. Führe die Übung als visuelles Fractool-Programm aus.

Motivation:
Durch die Integration der Muster *Composite*, *Iterator* und *Visitor* entsteht ein Aggregat, welches die drei klassischen Bestandteile einer komplexen Anwendung flexibel vereint: beliebig komplexe Datenstruktur, Navigation auf dieser Datenstruktur und Operationen auf den erreichten Knoten dieser Datenstruktur. Beispiel: Geometriekernel eines CAD-Systems für die Automobil- oder Programmkonstruktion.

Visualisierungswerkzeuge in der Praxis

Eine Umfrage unter Praktikern (2009)

- Welche Visualisierungen nutzen Sie für die graphische Darstellung einer Software-Architektur?
- Mit welchen Werkzeugen visualisieren Sie eine Software-Architektur?
- Welche Visualisierungen nutzen Sie für die Kommunikation und die Diskussion einer Software-Architektur mit anderen Projektbeteiligten?

Visualisierung	Nennungen
UML	18
Ad-hoc	6
Andere	2

Tabelle 2: Eingesetzte Software-Architekturvisualisierungen (n=18, Mehrfachnennungen möglich)

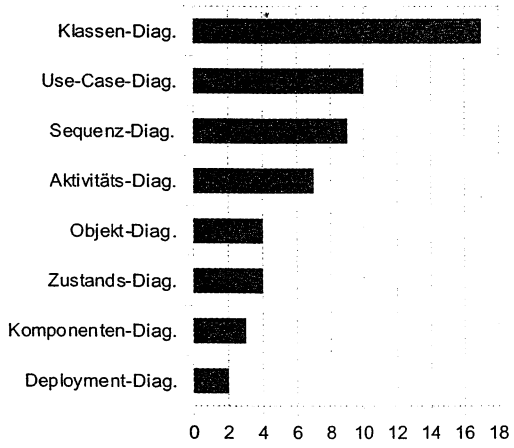


Abbildung 4: Genutzte UML-Diagramme (n=18, Mehrfachnennungen möglich)

Die Nutzung von Ad-hoc-Visualisierungen für die Kommunikation und die Diskussion einer Architektur mit Projektbeteiligten in nicht-technischen Rollen zeigt, dass eine Architektur-Visualisierung fehlt, welche die Anforderungen der nicht-technischen Projektbeteiligten an eine Software-Architekturvisualisierung erfüllt. Das Komponenten-Diagramm der UML, mit dem man am ehesten eine Software-Architektur visualisieren kann, erfüllt diese Anforderungen offensichtlich nicht, denn sonst würde es häufiger eingesetzt. Darum ist es sinnvoll, Ad-hoc-Visualisierungen genauer zu untersuchen und eine Software-Architekturvisualisierung zu entwickeln, welche die Kommunikation und die Diskussion einer Software-Architektur effektiv unterstützt.

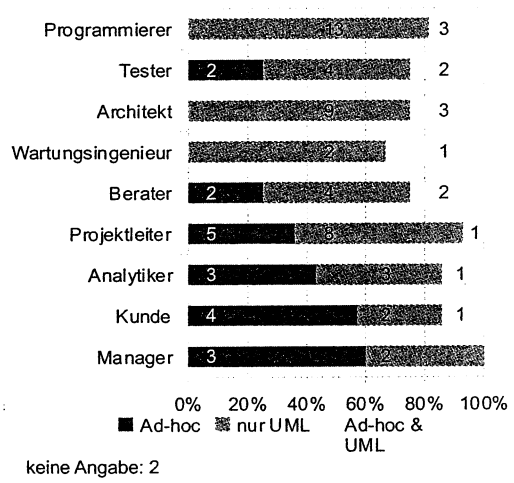


Abbildung 5: Genutzte Visualisierungen für die Kommunikation und die Diskussion einer Software-Architektur (n=16, Mehrfachnennungen möglich)

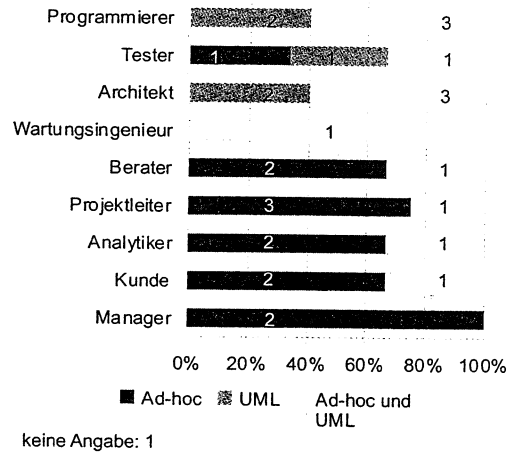


Abbildung 6: Von Architekten genutzte Visualisierungen für die Kommunikation und die Diskussion einer Software-Architektur (n=5, Mehrfachnennungen möglich)

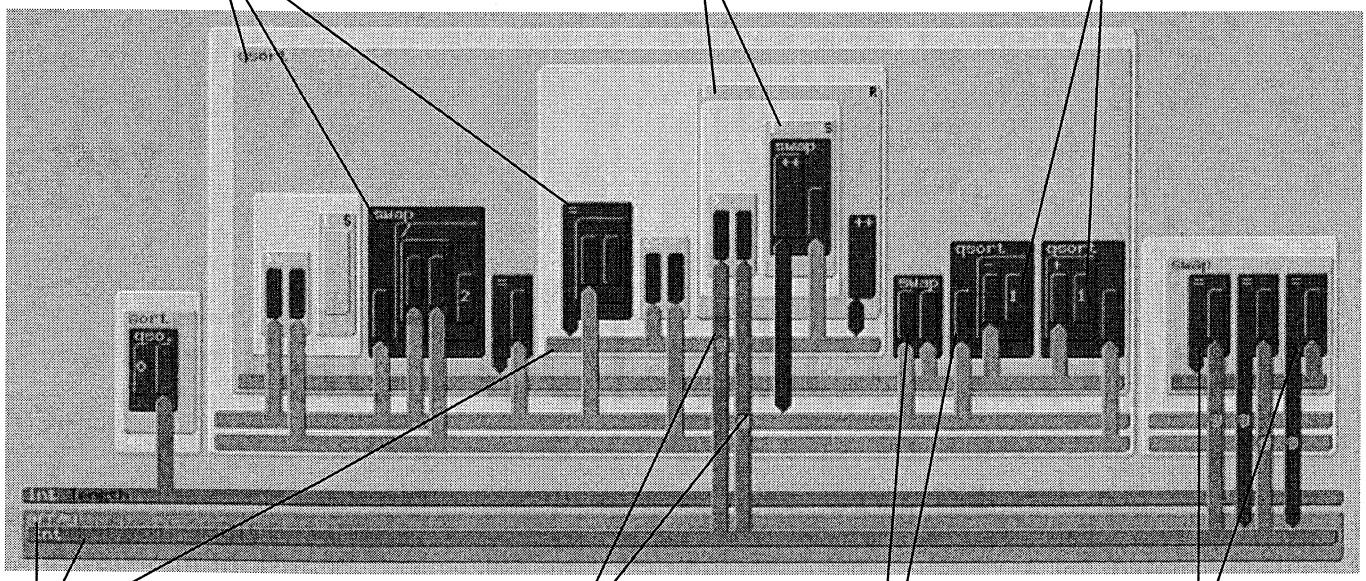
Aus: M. Knauß, Nutzung von Software-Architekturvisualisierungen in der Praxis, Softwaretechnik-Trends 29:3, GI-Fachgruppe SWA, August 2009

FracTool Click-Zonen

Click am linken oberen Rand
(nur 5 Pixel breit) für Popup
zur Eingabe von Methoden-,
Klassen und Modulnamen

Click am linken oberen Rand
für Popup zur Eingabe von
Select- und Repeat-Bedingungen

Click in der oberen Hälfte eine
I-Argument-Box für Popup zu
Eingabe einer Konstanten



lick im eingerückten Bereich
ır Popup zur Eingabe von
lassen- und Objektnamen

Kreuzungsmarken durch Rubber-
Banding von Kreuzung Zugriffspfeil-
Iterator bis zum Container

Element-Pfeile durch Rubber-Band
vom linken unteren Rand eines
Methodenaufrufs bis zum Objekt

Argument-Pfeile durch Rubber-
Banding von der unteren Hälfte
einer Argument-Box bis zum Objekt

Fractool INSERT mode

- legend of INSERT Options -

Module Context:
C - Class Definition
L - Layered Module Definition
M - Module Definition
P - Process Definition

Layered Module Context:
L - Layered Module Definition
M - Module Definition

Class Definition Context 1:
M - Method Definition

Class Definition Context 2:
C - Subclass Definition
M - Method Definition

Class Definition Context 3:
C - Subclass Definition

Class Association Context:
C -Class Definition

Method Definition Context 1:
B - Block
M - Method Call
Q - Do-while Repeat
R - While-do Repeat
S - Select
T - Thread

Method Definition Context 2:
B - Block
M - Method Call
Q - Do-while Repeat
R - While-do Repeat
S - Select
T - Thread
X - Return

Method Call Context:
M - Method Call
I - In Argument
O - InOut Argument

Return Context:
M - Method Call
I - In Argument

Flow Prologue Context:
B - Block
M - Method Call

While-do Prologue Context:
B - Block
M - Method Call
R - Repeat

Do-while Prologue Context:
B - Block
M - Method Call
S - Select

Flow Control Context 1:
S - Select

Flow Control Context 2:
R - Repeat

Flow Control Context 3:
S - Select
R - Repeat

Thread Prologue Context:
B - Block
M - Method Call
T - Thread

Thread Control Context:
T - Thread

***

Operand Context:
I - Iterator
C - Container
R - Record
S - Scalar

Patterns

What Makes a Pattern?

A *pattern for software architecture* describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

Context: a situation giving rise to a problem.

Problem: the recurring problem arising in that context.

Solution: a proven resolution of the problem.

The schema as a whole denotes a type of rule that establishes a relationship between a given context, a certain problem arising in that context, and an appropriate solution to the problem. All three parts of this schema are closely coupled.

Example pattern

Name Observer [GHJV95] or Publisher-Subscriber (339)

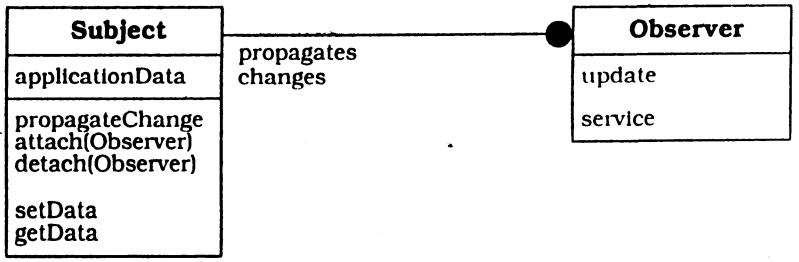
Context A component uses data or information provided by another component.

Problem Changing the internal state of a component may introduce inconsistencies between cooperating components. To restore consistency, we need a mechanism for exchanging data or state information between such components.

Two *forces* are associated with this problem:

- The components should be loosely coupled—the information provider should not depend on details of its collaborators.
- The components that depend on the information provider are not known a priori.

Solution Implement a change-propagation mechanism between the information provider—the *subject*—and the components dependent on it—the *observers*. Observers can dynamically register or unregister with this mechanism. Whenever the subject changes its state, it starts the change-propagation mechanism to restore consistency with all registered observers. Changes are propagated by invoking a special update function common to all observers. To implement change propagation—the passing of data and state information from the subject to the observers—you can use a *pull-model*, a *push-model*, or a combination of both.



Patterns

Pattern Categories

Architectural Patterns

An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Design Patterns

A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context [GHJV95].

Idioms

An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

J.O. Coplien: "Advanced Programming Styles & Idioms" Addison Wesley 1992

Patterns

Pattern Description

pattern description template

Name The name and a short summary of the pattern.

Also Known As Other names for the pattern, if any are known.

Example A real-world example demonstrating the existence of the problem and the need for the pattern.

Context The situations in which the pattern may apply.

Problem The problem the pattern addresses, including a discussion of its associated forces.

Solution The fundamental solution principle underlying the pattern.

Structure A detailed specification of the structural aspects of the pattern, including an OMT class diagram [RBPEL91].

Dynamics Typical scenarios describing the run-time behavior of the pattern.

Implementation Guidelines for implementing the pattern.

Example Resolved Discussion of any important aspects for resolving the example that are not yet covered in the Solution, Structure, Dynamics and Implementation sections.

Variants A brief description of variants or specializations of a pattern.

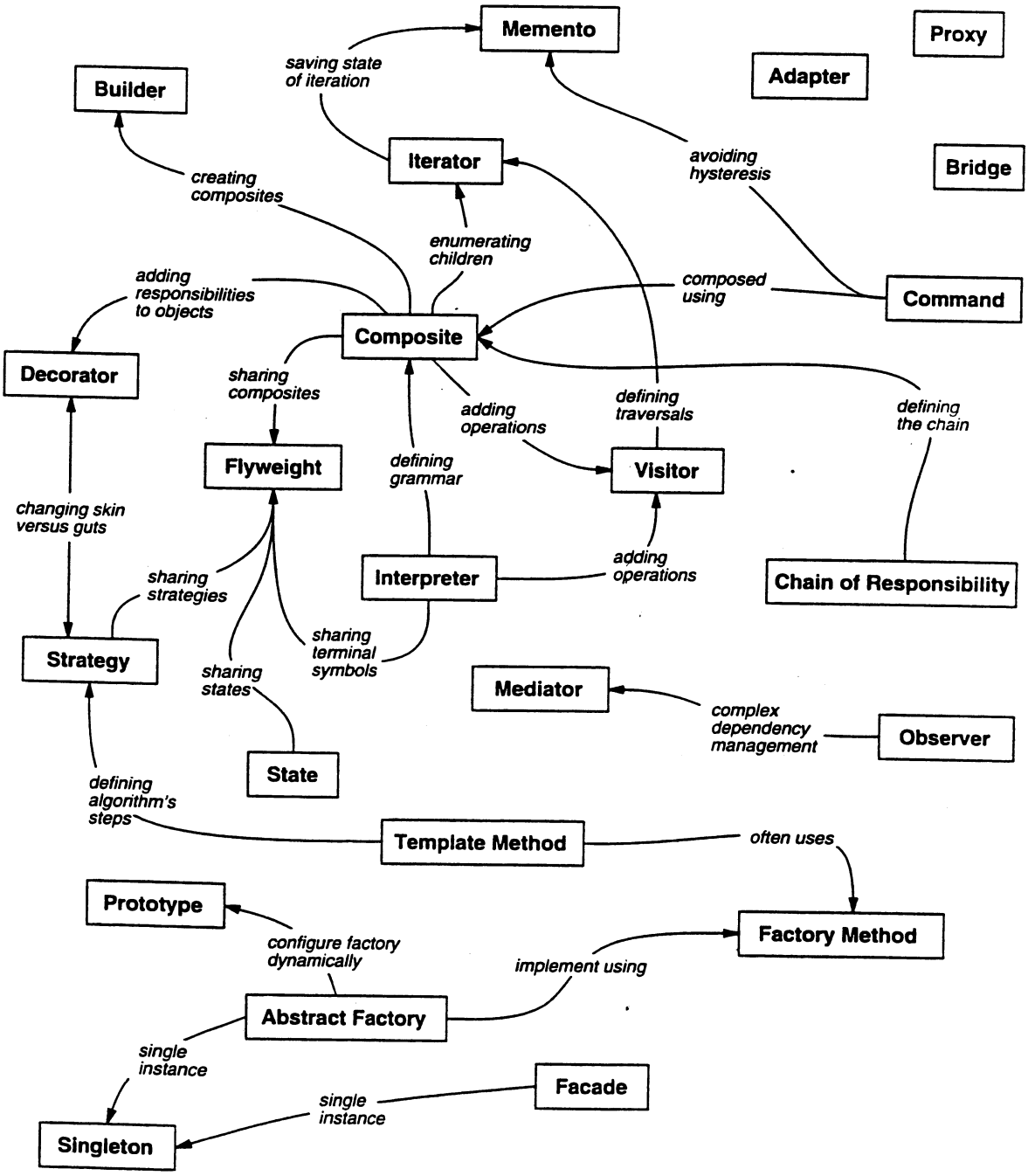
Known Uses Examples of the use of the pattern, taken from existing systems.

Consequences The benefits the pattern provides, and any potential liabilities.

See Also References to patterns that solve similar problems, and to patterns that help us refine the pattern we are describing.

Design Patterns

Design Pattern Relationships



Composite

Object Structural Pattern

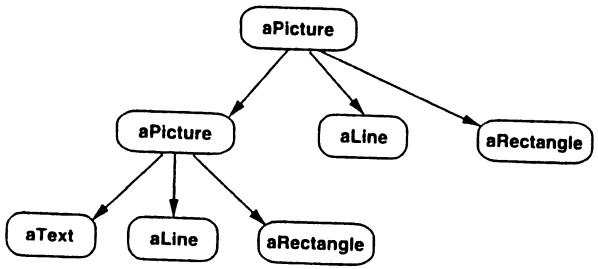
Intent

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

Applicability

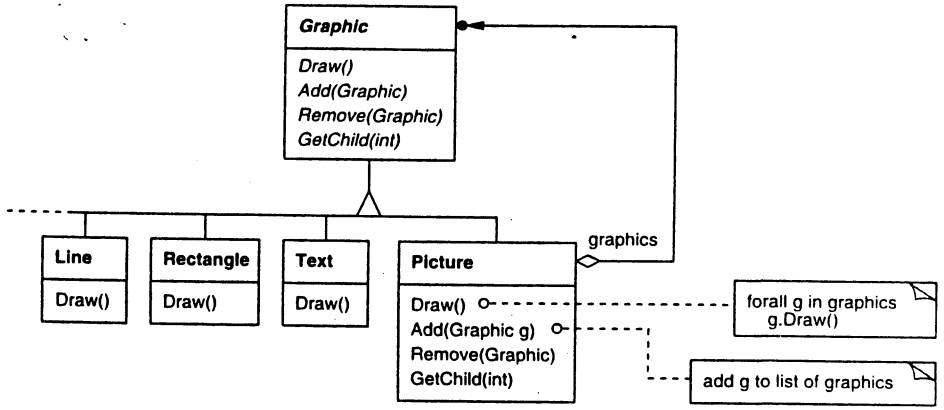
Use the Composite pattern when

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.



Motivation

Graphics applications like drawing editors and schematic capture systems l users build complex diagrams out of simple components. The user can grou components to form larger components, which in turn can be grouped to form st larger components.

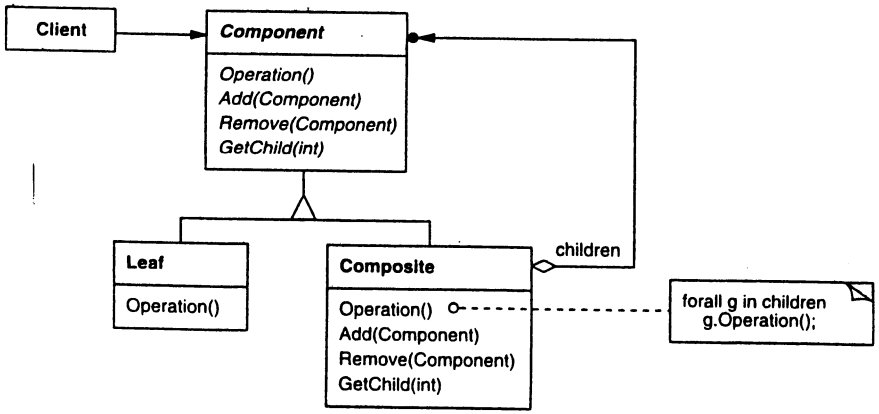


The key to the Composite pattern is an abstract class that represents *both* primitives and their containers. For the graphics system, this class is Graphic. Graphic declares operations like Draw that are specific to graphical objects. It also declares operations that all composite objects share, such as operations for accessing and managing its children.

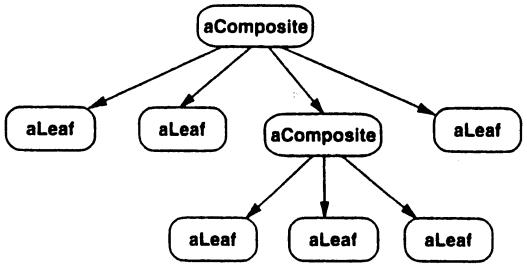
Composite

Structure, Participants, Collaborations

Structure



A typical Composite object structure might look like this:



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.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **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**
  - 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. → recursion

Composite

Consequences

The Composite pattern

- defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.
- makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.
- makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.
- can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.
- promotes recursive algorithms on components

Composite

Implementation

There are many issues to consider when implementing the Composite pattern:

1. *Explicit parent references.* Maintaining references from child components to their parent can simplify the traversal and management of a composite structure. The parent reference simplifies moving up the structure and deleting a component. Parent references also help support the Chain of Responsibility (223) pattern.

   The usual place to define the parent reference is in the Component class. Leaf and Composite classes can inherit the reference and the operations that manage it.

   With parent references, it's essential to maintain the invariant that all children of a composite have as their parent the composite that in turn has them as children. The easiest way to ensure this is to change a component's parent *only* when it's being added or removed from a composite. If this can be implemented once in the Add and Remove operations of the Composite class, then it can be inherited by all the subclasses, and the invariant will be maintained automatically.

3. *Maximizing the Component interface.* One of the goals of the Composite pattern is to make clients unaware of the specific Leaf or Composite classes they're using. To attain this goal, the Component class should define as many common operations for Composite and Leaf classes as possible. The Component class usually provides default implementations for these operations, and Leaf and Composite subclasses will override them.

Composite

Implementation

4. *Declaring the child management operations.* Although the Composite class *implements* the Add and Remove operations for managing children, an important issue in the Composite pattern is which classes *declare* these operations in the Composite class hierarchy. Should we declare these operations in the Component and make them meaningful for Leaf classes, or should we declare and define them only in Composite and its subclasses?

The decision involves a trade-off between safety and transparency:

- Defining the child management interface at the root of the class hierarchy gives you transparency, because you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.
- Defining child management in the Composite class gives you safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language like C++. But you lose transparency, because leaves and composites have different interfaces.

If you opt for safety, then at times you may lose type information and have to convert a component into a composite. How can you do this without resorting to a type-unsafe cast?

One approach is to declare an operation `Composite* GetComposite()` in the Component class. Component provides a default operation that returns a null pointer. The Composite class redefines this operation to return itself through the `this` pointer:

type-safe solution:

```
class Composite;

class Component {
public:
    //...
    virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
    void Add(Component*);
    // ...
    virtual Composite* GetComposite() { return this; }
};

class Leaf : public Component {
    // ...
};
```

`GetComposite` lets you query a component to see if it's a composite. You can perform `Add` and `Remove` safely on the composite it returns.

```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()) {
    test->Add(new Leaf);
}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {
    test->Add(new Leaf); // will not add leaf
}
```

Composite

Implementation

5 *Should Component implement a list of Components?* You might be tempted to define the set of children as an instance variable in the Component class where the child access and management operations are declared. But putting the child pointer in the base class incurs a space penalty for every leaf, even though a leaf never has children. This is worthwhile only if there are relatively few children in the structure.

6 *Child ordering.* Many designs specify an ordering on the children of Composite. In the earlier Graphics example, ordering may reflect front-to-back ordering. If Composites represent parse trees, then compound statements can be instances of a Composite whose children must be ordered to reflect the program.

When child ordering is an issue, you must design child access and management interfaces carefully to manage the sequence of children. The Iterator (257) pattern can guide you in this.

7 *Caching to improve performance.* If you need to traverse or search compositions frequently, the Composite class can cache traversal or search information about its children. The Composite can cache actual results or just information that lets it short-circuit the traversal or search. For example, the Picture class from the Motivation example could cache the bounding box of its children. During drawing or selection, this cached bounding box lets the Picture avoid drawing or searching when its children aren't visible in the current window.

Changes to a component will require invalidating the caches of its parents. This works best when components know their parents. So if you're using caching, you need to define an interface for telling composites that their caches are invalid.

8 *Who should delete components?* In languages without garbage collection, it's usually best to make a Composite responsible for deleting its children when it's destroyed. An exception to this rule is when Leaf objects are immutable and thus can be shared.

9 *What's the best data structure for storing components?* Composites may use a variety of data structures to store their children, including linked lists, trees, arrays, and hash tables. The choice of data structure depends (as always) on efficiency. In fact, it isn't even necessary to use a general-purpose data structure at all. Sometimes composites have a variable for each child, although this requires each subclass of Composite to implement its own management interface. See Interpreter (243) for an example.

Composite

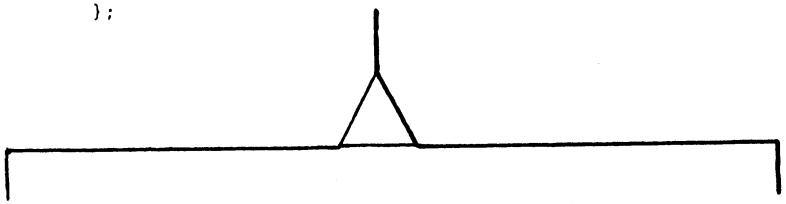
Sample Code

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```



```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List<Equipment*> _equipment;
};
```

Composite

Sample Code

CompositeEquipment

```
class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

Application

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

cout << "The net price is " << chassis->NetPrice() << endl;
```

```
Currency CompositeEquipment::NetPrice () {
    Iterator<Equipment*>* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

Composite

Subpattern "Interpreter"

An AndExp represents an expression made by ANDing two Boolean expressions together.

```
class AndExp : public BooleanExp {
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~AndExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
    _operand1 = op1;
    _operand2 = op2;
}
```

Evaluating an AndExp evaluates its operands and returns the logical "and" of the results.

```
bool AndExp::Evaluate (Context& aContext) {
    return
        _operand1->Evaluate(aContext) &&
        _operand2->Evaluate(aContext);
}
```

Now we can define the Boolean expression

```
(true and x) or (y and (not x))
```

and evaluate it for a given assignment of `true` or `false` to the variables x and y:

```
BooleanExp* expression;
Context context;

VariableExp* x = new VariableExp("X");
VariableExp* y = new VariableExp("Y");

expression = new OrExp(
    new AndExp(new Constant(true), x),
    new AndExp(y, new NotExp(x))
);

context.Assign(x, false);
context.Assign(y, true);

bool result = expression->Evaluate(context);
```

Iterator

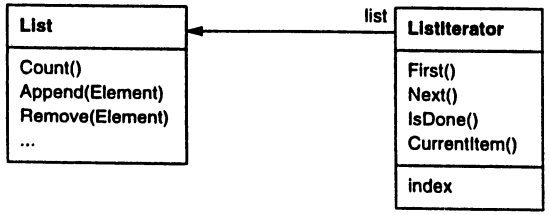
Object Behavioural Pattern

Intent

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

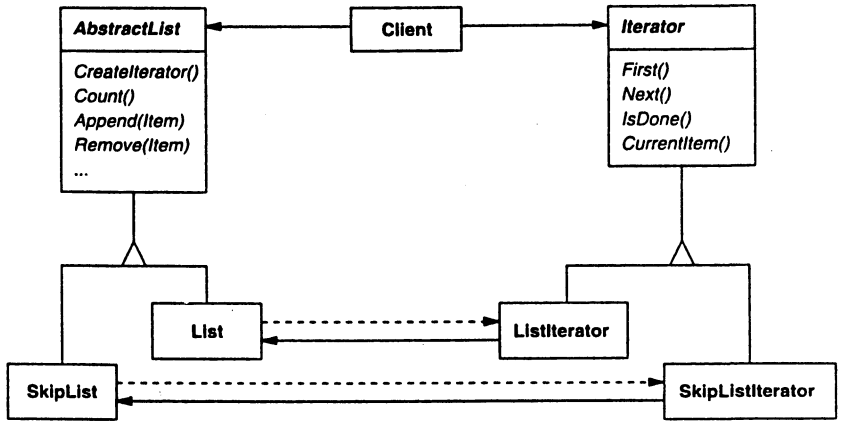
Motivation

An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.



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.**

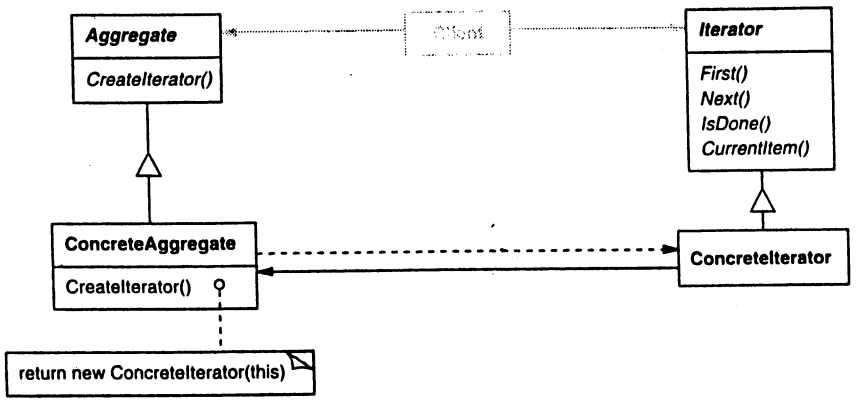
We define an AbstractList class that provides a common interface for manipulating lists. Similarly, we need an abstract Iterator class that defines a common iteration interface. Then we can define concrete Iterator subclasses for the different list implementations. As a result, the iteration mechanism becomes independent of concrete aggregate classes.



The remaining problem is how to create the iterator. Since we want to write code that's independent of the concrete List subclasses, we cannot simply instantiate a specific class. Instead, we make the list objects responsible for creating their corresponding iterator. This requires an operation like CreateIterator through which clients request an iterator object.

Iterator

Structure



Participants

- **Iterator**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator**
  - implements the Iterator interface.
  - 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.

Applicability

Use the Iterator pattern

- to access an aggregate object's contents without exposing its internal repre sentation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structure (that is, to support polymorphic iteration).

Collaborations

- A ConcreteIterator keeps track of the current object in the aggregate and ca compute the succeeding object in the traversal.

Consequences

The Iterator pattern has three important consequences:

1. *It supports variations in the traversal of an aggregate.* Complex aggregates ma be traversed in many ways. For example, code generation and semanti checking involve traversing parse trees. Code generation may traverse th parse tree inorder or preorder. Iterators make it easy to change the traversa algorithm: Just replace the iterator instance with a different one. You can als define Iterator subclasses to support new traversals.
2. *Iterators simplify the Aggregate interface.* Iterator's traversal interface obviate the need for a similar interface in Aggregate, thereby simplifying the aggre gate's interface.
3. *More than one traversal can be pending on an aggregate.* An iterator keeps trac of its own traversal state. Therefore you can have more than one traversal i progress at once.

Iterator

Implementation

1. *Who controls the iteration?* A fundamental issue is deciding which party controls the iteration, the iterator or the client that uses the iterator. When the client controls the iteration, the iterator is called an **external iterator**, and when the iterator controls it, the iterator is an **internal iterator**.[2] Clients that use an external iterator must advance the traversal and request the next element explicitly from the iterator. In contrast, the client hands an internal iterator an operation to perform, and the iterator applies that operation to every element in the aggregate.

2. *Who defines the traversal algorithm?* The iterator is not the only place where the traversal algorithm can be defined. The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration. We call this kind of iterator a **cursor**, since it merely points to the current position in the aggregate. A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor.[3]

   the traversal algorithm might need to access the private variables of the aggregate. If so, putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.

5. *Using polymorphic iterators in C++.* Polymorphic iterators have their cost. They require the iterator object to be allocated dynamically by a factory method. Hence they should be used only when there's a need for polymorphism. Otherwise use concrete iterators, which can be allocated on the stack.

6. *Iterators may have privileged access.* An iterator can be viewed as an extension of the aggregate that created it. The iterator and the aggregate are tightly coupled. We can express this close relationship in C++ by making the iterator a `friend` of its aggregate. Then you don't need to define aggregate operations whose sole purpose is to let iterators implement traversal efficiently.

   However, such privileged access can make defining new traversals difficult, since it'll require changing the aggregate interface to add another friend. To avoid this problem, the Iterator class can include `protected` operations for accessing important but publicly unavailable members of the aggregate. Iterator subclasses (and *only* Iterator subclasses) may use these protected operations to gain privileged access to the aggregate.

7. *Iterators for composites.* External iterators can be difficult to implement over recursive aggregate structures like those in the Composite (163) pattern, because a position in the structure may span many levels of nested aggregates. Therefore an external iterator has to store a path through the Composite to keep track of the current object. Sometimes it's easier just to use an internal iterator. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack.

   If the nodes in a Composite have an interface for moving from a node to its siblings, parents, and children, then a cursor-based iterator may offer a better alternative. The cursor only needs to keep track of the current node; it can rely on the node interface to traverse the Composite.

8. *Null iterators.* A **NullIterator** is a degenerate iterator that's helpful for handling boundary conditions. By definition, a NullIterator is *always* done with traversal; that is, its IsDone operation always evaluates to true.

   NullIterator can make traversing tree-structured aggregates (like Composites) easier. At each point in the traversal, we ask the current element for an iterator for its children. Aggregate elements return a concrete iterator.

Iterator

Sample Code

List and Iterator interfaces.

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;

private:
    const List<Item>* _list;
    long _current;
};
```

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}


template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}


template <class Item>
void ListIterator<Item>::Next () {
    _current++;
}


template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->Get(_current);
}


template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}
```

Iterator

Sample Code 2

Using the iterators.

```
void PrintEmployees (Iterator<Employee*>& i) {
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Print();
    }
}
```

```
List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```

Let's consider how a skiplist variation of `List` would affect our iteration code.

```
SkipList<Employee*>* employees;
// ...

SkipListIterator<Employee*> iterator(employees);
PrintEmployees(iterator);
```

Avoiding commitment to a specific list implementation.

To enable polymorphic iteration, `AbstractList` defines a factory method `CreateIterator`, which subclasses override to return their corresponding iterator:

```
template <class Item>
class AbstractList {
public:
    virtual Iterator<Item>* CreateIterator() const = 0;
    // ...
};
```

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

```
// we know only that we have an AbstractList
AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

Iterator

Implementation 2

Using polymorphic iterators in C++.

Polymorphic iterators have another drawback: the client is responsible for deleting them. This is error-prone, because it's easy to forget to free a heap-allocated iterator object when you're finished with it. That's especially likely when there are multiple exit points in an operation. And if an exception is triggered, the iterator object will never be freed.

The Proxy (207) pattern provides a remedy. We can use a stack-allocated proxy as a stand-in for the real iterator. The proxy deletes the iterator in its destructor. Thus when the proxy goes out of scope, the real iterator will get deallocated along with it. The proxy ensures proper cleanup, even in the face of exceptions. This is an application of the well-known C++ technique "resource allocation is initialization" [ES90]. The Sample Code gives an example.

Making sure iterators get deleted. Notice that `CreateIterator` returns a newly allocated iterator object. We're responsible for deleting it. If we forget, then we've created a storage leak. To make life easier for clients, we'll provide an `IteratorPtr` that acts as a proxy for an iterator. It takes care of cleaning up the `Iterator` object when it goes out of scope.

`IteratorPtr` is always allocated on the stack.[5] C++ automatically takes care of calling its destructor, which deletes the real iterator. `IteratorPtr` overloads both `operator->` and `operator*` in such a way that an `IteratorPtr` can be treated just like a pointer to an iterator. The members of `IteratorPtr` are all implemented inline; thus they can incur no overhead.

```
template <class Item>
class IteratorPtr {
public:
    IteratorPtr(Iterator<Item>* i): _i(i) { }
    ~IteratorPtr() { delete _i; }

    Iterator<Item>* operator->() { return _i; }
    Iterator<Item>& operator*() { return *_i; }
private:
    // disallow copy and assignment to avoid
    // multiple deletions of _i:

    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=(const IteratorPtr&);
private:
    Iterator<Item>* _i;
};
```

`IteratorPtr` lets us simplify our printing code:

```
AbstractList<Employee*>* employees;
// ...

IteratorPtr<Employee*> iterator(employees->CreateIterator());
PrintEmployees(*iterator);
```

[5] You can ensure this at compile-time just by declaring private `new` and `delete` operators. An accompanying implementation isn't needed.

Iterator
Example Code 3

An internal ListIterator.

```
template <class Item>
class ListTraverser {
public:
    ListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

```
template <class Item>
ListTraverser<Item>::ListTraverser (
    List<Item>* aList
) : _iterator(aList) { }
```

```
template <class Item>
bool ListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        result = ProcessItem(_iterator.CurrentItem());

        if (result == false) {
            break;
        }
    }
    return result;
}
```

Let's use a `ListTraverser` to print the first 10 employees from our employee list. To do it we have to subclass `ListTraverser` and override `ProcessItem`. We count the number of printed employees in a `_count` instance variable.

```
class PrintNEmployees : public ListTraverser<Employee*> {
public:
    PrintNEmployees(List<Employee*>* aList, int n) :
        ListTraverser<Employee*>(aList),
        _total(n), _count(0) { }

protected:
    bool ProcessItem(Employee* const&);
private:
    int _total;
    int _count;
};

bool PrintNEmployees::ProcessItem (Employee* const& e) {
    _count++;
    e->Print();
    return _count < _total;
}
```

Here's how `PrintNEmployees` prints the first 10 employees on the list:

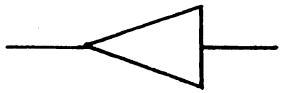
```
List<Employee*>* employees;
// ...

PrintNEmployees pa(employees, 10);
pa.Traverse();
```

Iterator

Sample Code 4

```
template <class Item>
class ListTraverser {
public:
    ListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```



```
template <class Item>
class FilteringListTraverser {
public:
    FilteringListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

```
template <class Item>
bool ListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        result = ProcessItem(_iterator.CurrentItem());

        if (result == false) {
            break;
        }
    }
    return result;
}
```

```
template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());

            if (result == false) {
                break;
            }
        }
    }
    return result;
}
```

Internal iterators can encapsulate different kinds of iteration. For example, `FilteringListTraverser` encapsulates an iteration that processes only items that satisfy a test:

Visitor

Behavioural Pattern

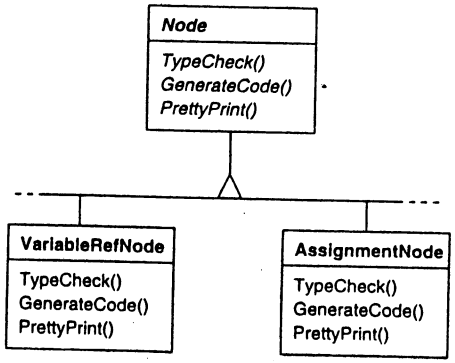
Intent

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.

Motivation

Consider a compiler that represents programs as abstract syntax trees. It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all variables are defined. It will also need to generate code. So it might define operations for type-checking, code optimization, flow analysis, checking for variables being assigned values before they're used, and so on. Moreover, we could use the abstract syntax trees for pretty-printing, program restructuring, code instrumentation, and computing various metrics of a program.

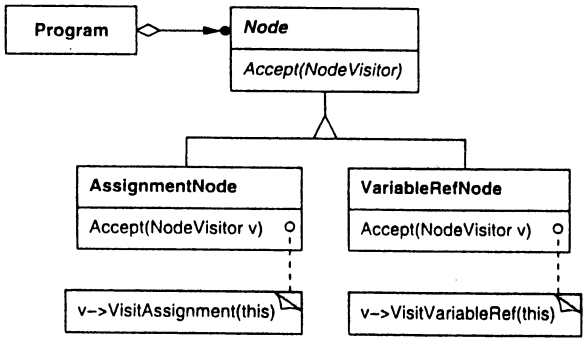
Most of these operations will need to treat nodes that represent assignment statements differently from nodes that represent variables or arithmetic expressions. Hence there will be one class for assignment statements, another for variable accesses, another for arithmetic expressions, and so on. The set of node classes depends on the language being compiled, of course, but it doesn't change much for a given language.

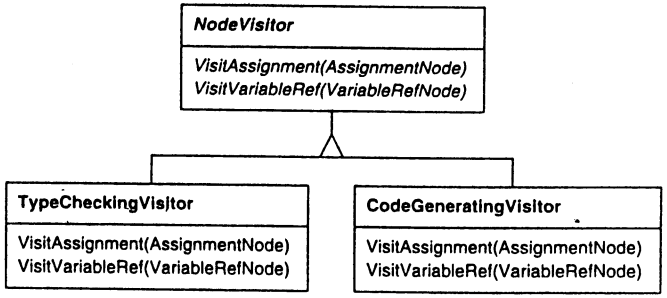


This diagram shows part of the Node class hierarchy. The problem here is that distributing all these operations across the various node classes leads to a system that's hard to understand, maintain, and change. It will be confusing to have type-checking code mixed with pretty-printing code or flow analysis code. Moreover, adding a new operation usually requires recompiling all of these classes. It would be better if each new operation could be added separately, and the node classes were independent of the operations that apply to them.

Visitor

Motivation





Solution by packaging related operations from each class in a separate object, called a **visitor**, and passing it to elements of the abstract syntax tree as it's traversed. When an element "accepts" the visitor, it sends a request to the visitor that encodes the element's class. It also includes the element as an argument. The visitor will then execute the operation for that element—the operation that used to be in the class of the element.

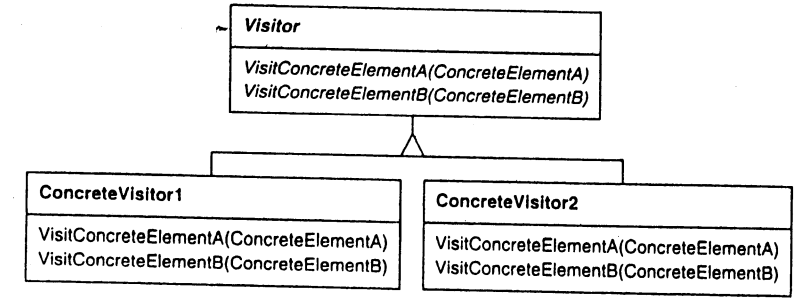
For example, a compiler that didn't use visitors might type-check a procedure by calling the TypeCheck operation on its abstract syntax tree. Each of the nodes would implement TypeCheck by calling TypeCheck on its components (see the preceding class diagram). If the compiler type-checked a procedure using visitors, then it would create a TypeCheckingVisitor object and call the Accept operation on the abstract syntax tree with that object as an argument. Each of the nodes would implement Accept by calling back on the visitor: an assignment node calls VisitAssignment operation on the visitor, while a variable reference calls VisitVariableReference. What used to be the TypeCheck operation in class AssignmentNode is now the VisitAssignment operation on TypeCheckingVisitor.

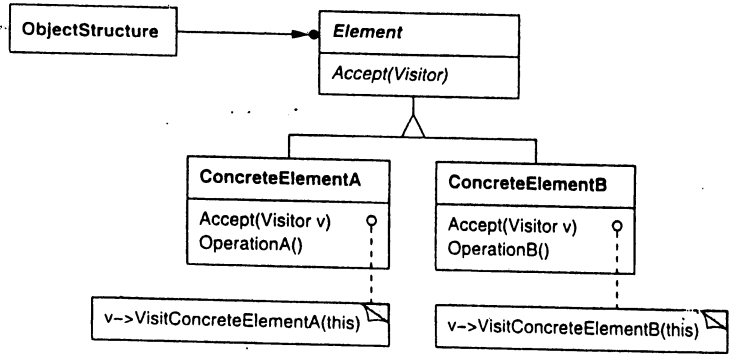
To make visitors work for more than just type-checking, we need an abstract parent class NodeVisitor for all visitors of an abstract syntax tree. NodeVisitor must declare an operation for each node class. An application that needs to compute program metrics will define new subclasses of NodeVisitor and will no longer need to add application-specific code to the node classes. The Visitor pattern encapsulates the operations for each compilation phase in a Visitor associated with that phase.

With the Visitor pattern, you define two class hierarchies: one for the elements being operated on (the Node hierarchy) and one for the visitors that define operations on the elements (the NodeVisitor hierarchy). You create a new operation by adding a new subclass to the visitor class hierarchy. As long as the grammar that the compiler accepts doesn't change (that is, we don't have to add new Node subclasses), we can add new functionality simply by defining new NodeVisitor subclasses.

Visitor

Structure





Applicability

Use the Visitor pattern when

- an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
- the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

Visitor

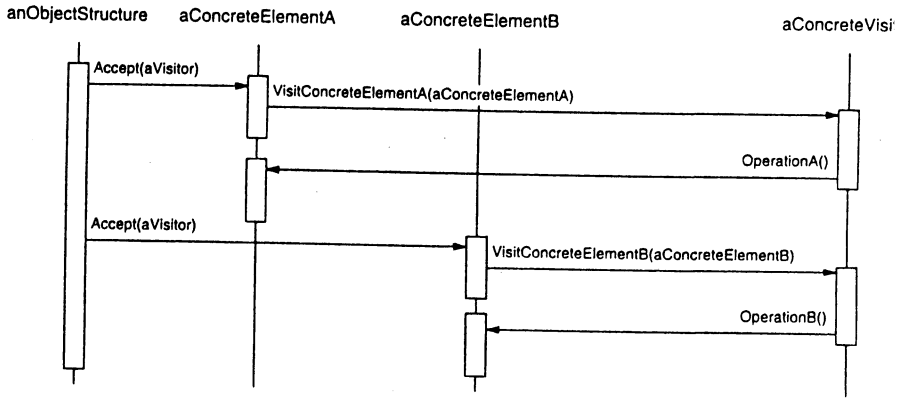
Participants

- **Visitor** (NodeVisitor)
  - 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** (TypeCheckingVisitor)
  - 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** (Node)
  - defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement** (AssignmentNode,VariableRefNode)
  - implements an Accept operation that takes a visitor as an argument.
- **ObjectStructure** (Program)
  - can enumerate its elements.
  - may provide a high-level interface to allow the visitor to visit its elements.
  - may either be a composite (see Composite (163)) or a collection such as a list or a set.

Collaborations

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

The following interaction diagram illustrates the collaborations between object structure, a visitor, and two elements:



Visitor

Consequences

Some of the benefits and liabilities of the Visitor pattern are as follows:

- *Visitor makes adding new operations easy.* Visitors make it easy to add operations that depend on the components of complex objects. You can define a new operation over an object structure simply by adding a new visitor. In contrast, if you spread functionality over many classes, then you must change each class to define a new operation.

- *A visitor gathers related operations and separates unrelated ones.* Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor subclasses. That simplifies both the classes defining the elements and the algorithms defined in the visitors. Any algorithm-specific data structures can be hidden in the visitor.

- *Adding new ConcreteElement classes is hard.* The Visitor pattern makes it hard to add new subclasses of Element. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class. Sometimes a default implementation can be provided in Visitor that can be inherited by most of the ConcreteVisitors, but this is the exception rather than the rule.

  So the key consideration in applying the Visitor pattern is whether you are mostly likely to change the algorithm applied over an object structure or the classes of objects that make up the structure. The Visitor class hierarchy can be difficult to maintain when new ConcreteElement classes are added frequently. In such cases, it's probably easier just to define operations on the classes that make up the structure. If the Element class hierarchy is stable, but you are continually adding operations or changing algorithms, then the Visitor pattern will help you manage the changes.

- *Visiting across class hierarchies.* An iterator (see Iterator (257)) can visit the objects in a structure as it traverses them by calling their operations. But an iterator can't work across object structures with different types of elements. For example, the Iterator interface defined on page 263 can access only objects of type `Item`:

```
template <class Item>
class Iterator {
    // ...
    Item CurrentItem() const;
};
```

  This implies that all elements the iterator can visit have a common parent class `Item`. → Composite/Component

  Visitor does not have this restriction. It can visit objects that don't have a common parent class. You can add any type of object to a Visitor interface. For example, in

```
class Visitor {
public:
    // ...
    void VisitMyType(MyType*);
    void VisitYourType(YourType*);
};
```

  `MyType` and `YourType` do not have to be related through inheritance at all.

- *Accumulating state.* Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would be passed as extra arguments to the operations that perform the traversal, or they might appear as global variables.

- *Breaking encapsulation.* Visitor's approach assumes that the ConcreteElement interface is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access an element's internal state, which may compromise its encapsulation.

Visitor

Implementation

Each object structure will have an associated Visitor class. This abstract visitor class declares a VisitConcreteElement operation for each class of ConcreteElement defining the object structure. Each Visit operation on the Visitor declares its argument to be a particular ConcreteElement, allowing the Visitor to access the interface of the ConcreteElement directly. ConcreteVisitor classes override each Visit operation to implement visitor-specific behavior for the corresponding ConcreteElement class.

The Visitor class would be declared like this in C++:

```
class Visitor {
public:
    virtual void VisitElementA(ElementA*);
    virtual void VisitElementB(ElementB*);

    // and so on for other concrete elements
protected:
    Visitor();
};
```

Each class of ConcreteElement implements an `Accept` operation that calls the matching `Visit...` operation on the visitor for that ConcreteElement. Thus the operation that ends up getting called depends on both the class of the element and the class of the visitor.[10]

[10] We could use function overloading to give these operations the same simple name, like `Visit`, since the operations are already differentiated by the parameter they're passed. There are pros and cons to such overloading.

```
class Element {
public:
    virtual ~Element();
    virtual void Accept(Visitor&) = 0;
protected:
    Element();
};
```

```
class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class ElementB : public Element {
public:
    ElementB();
    virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};


class CompositeElement : public Element {
public:
    virtual void Accept(Visitor&);
private:
    List<Element*>* _children;
};

void CompositeElement::Accept (Visitor& v) {
    ListIterator<Element*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}
```

Visitor

Implementation

Double dispatch. Effectively, the Visitor pattern lets you add operations to classes without changing them. Visitor achieves this by using a technique called **double-dispatch**. It's a well-known technique. In fact, some programming languages support it directly (CLOS, for example). Languages like C++ and Smalltalk support **single-dispatch**.

In single-dispatch languages, two criteria determine which operation will fulfill a request: the name of the request and the type of receiver. For example, the operation that a GenerateCode request will call depends on the type of node object you ask. In C++, calling `GenerateCode` on an instance of `VariableRefNode` will call `VariableRefNode::GenerateCode` (which generates code for a variable reference). Calling `GenerateCode` on an `AssignmentNode` will call `AssignmentNode::GenerateCode` (which will generate code for an assignment). The operation that gets executed depends both on the kind of request and the type of the receiver.

"Double-dispatch" simply means the operation that gets executed depends on the kind of request and the types of *two* receivers. `Accept` is a double-dispatch operation. Its meaning depends on two types: the Visitor's and the Element's. Double-dispatching lets visitors request different operations on each class of element.[11]

This is the key to the Visitor pattern: The operation that gets executed depends on both the type of Visitor and the type of Element it visits. Instead of binding operations statically into the Element interface, you can consolidate the operations in a Visitor and use `Accept` to do the binding at run-time. Extending the Element interface amounts to defining one new Visitor subclass rather than many new Element subclasses.

Who is responsible for traversing the object structure? A visitor must visit each element of the object structure. The question is, how does it get there? We can put responsibility for traversal in any of three places: in the object structure, in the visitor, or in a separate iterator object (see Iterator (257)).

Often the object structure is responsible for iteration. A collection will simply iterate over its elements, calling the Accept operation on each. A composite will commonly traverse itself by having each Accept operation traverse the element's children and call Accept on each of them recursively.

Another solution is to use an iterator to visit the elements. In C++, you could use either an internal or external iterator, depending on what is available and what is most efficient. In Smalltalk, you usually use an internal iterator using `do:` and a block. Since internal iterators are implemented by the object structure, using an internal iterator is a lot like making the object structure responsible for iteration. The main difference is that an internal iterator will not cause double-dispatching—it will call an operation on the *visitor* with an *element* as an argument as opposed to calling an operation on the *element* with the *visitor* as an argument. But it's easy to use the Visitor pattern with an internal iterator if the operation on the visitor simply calls the operation on the element without recursing.

You could even put the traversal algorithm in the visitor, although you'll end up duplicating the traversal code in each ConcreteVisitor for each aggregate ConcreteElement. The main reason to put the traversal strategy in the visitor is to implement a particularly complex traversal, one that depends on the results of the operations on the object structure. We'll give an example of such a case in the Sample Code.

Visitor

Sample Code

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Accept(EquipmentVisitor&);
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

class FloppyDisk : public Equipment {...}
class Chassis : public Equipment {...}

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {
    visitor.VisitFloppyDisk(this);
}

void Chassis::Accept (EquipmentVisitor& visitor) {
    for (
        ListIterator<Equipment*> i(_parts);
        !i.IsDone();
        i.Next()
    ) {
        i.CurrentItem()->Accept(visitor);
    }
    visitor.VisitChassis(this);
}
```

```
class EquipmentVisitor {
public:
    virtual ~EquipmentVisitor();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);

    // and so on for other concrete subclasses of Equipment
protected:
    EquipmentVisitor();
};
```

```
class PricingVisitor : public EquipmentVisitor {
public:
    PricingVisitor();

    Currency& GetTotalPrice();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    // ...
private:
    Currency _total;
};

void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _total += e->NetPrice();
}

void PricingVisitor::VisitChassis (Chassis* e) {
    _total += e->DiscountPrice();
}
```

Visitor

Sample Code

Erweiterung:

class EquipmentVisitor {...};

```
class InventoryVisitor : public EquipmentVisitor {
public:
    InventoryVisitor();

    Inventory& GetInventory();


    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    // ...

private:
    Inventory _inventory;
};
```

```
void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _inventory.Accumulate(e);
}

void InventoryVisitor::VisitChassis (Chassis* e) {
    _inventory.Accumulate(e);
}
```

Anwendung:

```
Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Inventory "
     << component->Name()
     << visitor.GetInventory();
```

Visitor

Mögliche Orte für die Übergabe, Speicherung und Verknüpfung der Parameter/Rückgabewerte

```
class Xpr {
  public:
    virtual ① Accept (Visitor&, ②);
};
```

```
class S_Xpr {
  int value;
  public:
    ① Accept (Visitor&, ②);
};
```

```
class C_Xpr {
  Xpr* op1;
  Xpr* op2;
  ③
  public:
    ① Accept (Visitor&, ②);
};
```

```
class Visitor {
  public:
    virtual ④ visitS (S_Xpr*, ⑤);
    virtual ④ visitC (C_Xpr*, ⑤);
};
```

```
class Interpret :: public Visitor {
  ⑥
  public:
    ④ visitS (S_Xpr*, ⑤);
    ④ visitC (C_Xpr*, ⑤);
};
```

Legend:

① Return value of Accept()
② I/O Parameter of Accept()
③ Attribute of C_Xpr node
④ Return value of visitX()
⑤ I/O Parameter of visitX()
⑥ Attribute of Visitor (one instance only)
⑦ Call-stack of Accept()

```
① S_Xpr::Accept(Visitor& v, ②)
{
  ④ = v.visitS(this, ⑤);
  return ①(②, ④, ⑤);
}
```

```
① C_Xpr::Accept(Visitor& v, ②)
{
  ⑦
  ① = op1->Accept(v, ①);
  ① = op2->Accept(v, ①);
  ④ = v.visitC(this, ⑤);
  return ①(②, ④, ⑤, ⑦);
}
```

```
④ Interpret::visitS(S_Xpr* s, ⑤)
{
  return ④(s->value);
}
```

```
④ Interpret::visitC(C_Xpr* c, ⑤)
{
  return ④(*c, ⑤);
}
```