Object-Oriented Programming
Visitor Pattern
Observer Pattern

CSIE Department, NTUT
Woei-Kae Chen
Visitor Pattern
Behavioral pattern
Visitor: 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.

• Example 1
  object structure: abstract syntax tree of a compiler
  element: AssignmentNode, etc.
  operations: GenerateCode, etc.
• Example 2
  object structure: Petri Net
  element: place, transition, etc.
  operations: fire, etc.
Visitor: Motivation (1)

- Option 1: implement operations on each ConcreteNode class
  - distributing operations across the various node classes leads to a system that’s hard to understand, maintain, and change
  - confusing to have type-checking code mixed with pretty-printing code.
  - adding a new operation requires recompiling all of these classes
Visitor: Motivation (2)

- Option 2: Visitor pattern
  - Use a NodeVisitor hierarchy to define operations on the elements
  - Fixed grammar → fixed Node classes → easy to add new functionality (visitor)

- Perform operations on concrete class
- Integrating all code generation programs into one class
- GenerateCode() moved here
- GenerateCode() removed
- Must open interfaces for different Visitors
Visitor: 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.
  - avoid “polluting” classes with many distinct and unrelated operations
    - visitor lets you keep related operations together by defining them in one class
  - the object classes rarely change, but you often want to define new operations over the structure
    - changing object classes often ⇒ better to define operations in object classes
Visitor: Structure

Client

Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)

ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)

ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)

Element
Accept(Visitor)

ConcreteElementA
Accept(Visitor v)
OperationA()
v->VisitConcreteElementA(this)

ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementB(this)

ObjectStructure

May use Composite pattern

Visitor as an argument
Visitor: Participants

- **Visitor (NodeVisitor)**
  - declares a `VisitConcreteElement` operation for each `ConcreteElement` class in the object structure.

- **ConcreteVisitor (CodeGenerationVisitor, etc.)**
  - defines an operation to the object structure.
  - implements each operation declared by Visitor.
  - may accumulate results during the traversal of the object structure.

- **Element (Node)**
  - defines an `Accept` operation that takes a visitor as argument.

- **ConcreteElement (AssignmentNode, etc.)**
  - implements an `Accept` operation that takes a visitor as an argument.

- **Object structure (Program)**
  - 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.
Visitor: Collaboration

- A client creates a ConcreteVisitor object, then
  - traverse the object structure, visiting each element with the visitor.
- When an element is visited (accepts a visitor)
  - it calls the visitor operation that corresponds to its class.
  - passing itself as an argument if necessary.
Visitor: Consequences (1)

- **Benefits**
  - *Visitor makes adding new operations easy*
    - simply add a new concrete 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 is localized in a visitor.
  - *Accumulating state*
    - The Collecting Parameter pattern.
    - Visitors are arguments → Visitors can accumulate state as they visit each element in the object structure.
    - Without a Visitor → pass state as extra arguments to the operations that perform the traversal.
Visitor: Consequences (2)

- **Benefits**
  - *Visiting across class hierarchies*
    - An iterator can also visit the objects in a structure.
    - Iterator → all elements must have a common parent class (Item)
      ```cpp
template <class Item>
class Iterator {
  ...
  Item CurrentItem() const;
};
```
    - Visitor → can visit objects that do not have a common parent class
      ```cpp
class Visitor {
public:
  ...
  void VisitMyType(MyType *);
  void VisitYourType(YourType *);
};
```
Visitor: Consequences (3)

● Liabilities
  – *Adding new ConcreteElement classes is hard*
    ● a new ConcreteElement → a new abstract operation on Visitor
    → change every ConcreteVisitor Class.
  – *Breaking encapsulation*
    ● The ConcreteElement’s public interface must be powerful enough to let visitors do their job → often forces you to provide public operations that access an element’s internal state.
Visitor: Implementation (1)

- The Visitor class (C++)
  
  ```
  class Visitor {
  public:
      virtual void VisitElementA(ElementA *e);  
      virtual void VisitElementB(ElementB *e);
  ...
  protected:
      Visitor();  // protected constructor
  };
  ```

- The ConcreteVisitor class (C++)
  
  ```
  class ConcreteVisitor1 : public Visitor {
  public:
      ConcreteVisitor1();
      virtual void VisitElementA(ElementA *e) 
      {...; e->OperaionA(); ...;}
      virtual void VisitElementB(ElementB *e); 
      {...; e->OperaionB(); ...;}
  };
  ```
Visitor: Implementation (2)

- The Element class (C++)
  ```cpp
class Element {
public:
    virtual ~Element();
    virtual void Accept(Visitor &) = 0;
protected:
    Element();  // protected constructor
};
```

- The ConcreteElement class (C++)
  ```cpp
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);
    }
};
```
A CompositeElement class (C++)

class CompositeElement : public Element {
public:
    CompositeElement();
    virtual void Accept(Visitor &) {
        ListIterator<Element *> i(_children);
        for (i.First(); !i.IsDone(); i.Next()) {
            i.CurrentElement()->Accept(v);
        }
        v.VisitCompositeElement(this);
    }
private:
    List<Element *> *_children;
};
Visitor: Implementation (4)

- **Implementation issues**
  - *Double dispatch*
    - **Single dispatch**
      - The operation that gets executed depends both on the kind of request and the type of the *(one)* receiver.
      - C++ supports only single-dispatch.
    - **Double (multiple) dispatch**
      - The operation that gets executed depends on the kind of request and the types of *two* receivers.
      - Languages that support multiple-dispatch lessen the need for the Visitor pattern.
  - **Accept** is a double-dispatch operation
    - Its meaning depends on *two* types: the Visitor’s and the Element’s.
    - **Accept** do the binding at run-time.
Visitor: Implementation (5)

- **Implementation issues**
  - *Who is responsible for traversing the object structure?*
    1. In the object structure
      - A collection simply iterate over its elements, calling the `Accept` operation on each.
      - A Composite implements the `Accept` operation by calling its children’s `Accept` operation recursively.
    2. Using a separate iterator object
      - Iterators can be used to traverse the object structure.
      - The key to the Visitor pattern is double-dispatching: the operation that gets executed depends on both the type of Visitor and the type of Element it visits.
    3. In the Visitor
      - When implementing complex traversal that depends on the results of the operations on the object structure.
      - Usually end up duplicating the traversal code in each `ConcreteVisitor` for each aggregate `ConcreteElement`.
Visitor: Related Patterns

- **Composite pattern**
  - Visitors can be used to apply an operation over an object structure defined by the Composite pattern.

- **Interpreter pattern**
  - Visitor can be applied to do the interpretation.
Observer Pattern
Behavioral pattern
Observer: Intent

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

*Also known as*
- Dependents
- Publish-Subscribe
Observer: Motivation

- **Observer pattern**
  - describes how to establish the relationship between subject (one) and observers (many).

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>60</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>50</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>z</td>
<td>80</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
```

- Subject
  - a = 50%
  - b = 30%
  - c = 20%

- Observer
  - change notification
  - requests, modifications
Observer: Applicability

- Use the Observer pattern in any of the following situations
  - When an abstraction has two aspects, one (object) dependent on the other (object).
    - Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you do not know how many objects need to be changed.
  - When an object should be able to notify other objects without knowing who these objects are.
    - Objects are loosely coupled
Observer: Structure

Subject
- Attach(Observer)
- Detach(Observer)
- Notify()

ConcreteSubject
- GetState()
- SetState()
- subjectState

Observers
- for all o in observers
  - o->Update();

Observer
- Update()

ConcreteObserver
- Update()
- observerState

observerState = subject->GetState()
Observer: Participants

- **Subject**
  - knows its observers. Any number of Observer objects may observe a subject.

- **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.
Observer: Collaborations

- A ConcreteSubject notifies its observers whenever a change occurs.
- A ConcreteObserver query the subject for information to reconcile its state.
Observer: Consequences

● Benefits
  - Vary subjects and observers independently.
  - Add observers without modifying the subject or other observers.
  - *Abstract coupling between Subject and Observer*
    ● The subject does not know the concrete class of any observer.
  - *Support for broadcast communication*
    ● The notification is broadcast automatically to all interested objects that subscribed to it.

● Liabilities
  - *Unexpected updates*
    ● A seemingly harmless operation on the subject may cause a cascade of updates to observers and their dependent objects.
Observer: Implementation (1)

- **Mapping subjects to their observers**
  - store references in the subject
    - such storage may be too expensive when there are many subjects and few observers.
  - associative lookup
    - maintains subject-to-observer mapping.
    - increases the cost of accessing observers.

- **Observing more than one subject**
  - extend the **Update** interface to let the observer know which subject is sending the notification.
  - the subject may pass itself as a parameter in the **Update** operation.

- **Dangling references to deleted subjects**
  - deleting the observers is not an option because other objects may reference them.
  - make the subject notify its observers as it is deleted so that they can reset their reference to it.
Observer: Implementation (2)

- **Who triggers the update? (calls notify)**
  - `SetState` call `Notify` after its state is changed
    - Advantage: client do not have to remember to call `Notify`
    - Disadvantage: several consecutive operations will cause several consecutive updates \(\rightarrow\) inefficient.
  - Client calls `Notify` at the right time
    - Advantage: client can trigger an update after a series of state changes \(\rightarrow\) more efficient.
    - Disadvantage: client might forget to call `notify` \(\rightarrow\) error prone.

- **Making sure Subject state is self-consistent before notification**
  - Use Template Method
    - define a primitive operation for subclasses to override and make `Notify` the last operation in the Template Method.
    ```cpp
    void Text::Cut(TextRange r) { // Template Method
        doReplaceRange(r); // redefined in subclasses
        Notify();
    }
    ```
Avoiding observer-specific update protocols: the push and pull models

- The subject may pass change information as an argument to `Update`
- Push model
  - The subject sends observers detailed information about the change, whether they want it or not.
  - The subject knows something about Observer classes
- Pull model
  - The subject sends nothing, and observers ask for details.
  - Observer must ascertain what changed without help from the Subject → inefficient.

Specifying modification of interest explicitly

- Improve update efficiency by extending the subject’s interface to allow registering observers only for specific events of interest.
  ```cpp
  void Subject::Attach(Observer*, Aspect &interest);
  void Observer::Update(Subject*, Aspect &interest);
  ```
Observer: Implementation (4)

- Encapsulating complex update semantics
  - Use a ChangeManager object (a Mediator) to minimize the work required to make observers reflect a change (eliminating unnecessary updates).

```plaintext
observer = ChangeManager()

observer.Register(subject, observer)
observer.UnRegister(subject, observer)
observer.Notify()
```

```plaintext
SimpleChangeManager
observer.Register(subject, observer)
observer.UnRegister(subject, observer)
observer.Notify()
```

```plaintext
DAGChangeManager
mark all observers to update
update all marked observers
```
**Observer: Implementation (5)**

- *Combining the Subject and Observer classes*
  - Combine the interface of Subject and Observer in one class → for an object to act as both a subject and an observer.
  - When multiple inheritance is not supported (e.g., Smalltalk).
Observer: Related Patterns

- **Mediator**
  - By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.

- **Singleton**
  - The ChangeManager may use the Singleton pattern to make it unique and globally accessible.