CS 2340 Objects and Design

Behavioral Patterns

Christopher Simpkins
chris.simpkins@gatech.edu

Behavioral Design Patterns

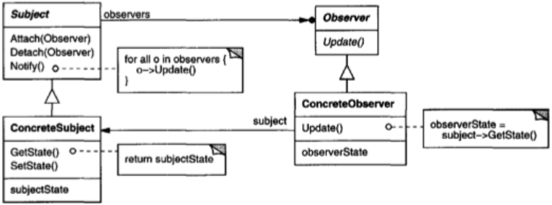
Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

- Behavioral class patterns use inheritance to distribute behavior between classes. (Template Method)
- Behavioral object patterns use object composition rather than inheritance. The Strategy (315) pattern encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses.

Observer (a.k.a. Dependents, Publish-Subscribe)

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



Participants

- **Subject** knows its observers.
- **Observer** defines a notification interface for objects that should be notified of changes in a subject.
- **ConcreteSubject** sends a notification to its observers when its state changes.
- **ConcreteObserver** implements Observer notification interface.

Observer Example: Swing Buttons

`javax.swing.AbstractButton` is a Subject, `javax.swing.JButton` is a ConcreteSubject. We set up an exit button like this:

```
JButton exitButton = new JButton("Exit");
exitButton.addActionListener(new ExitListener());
```

`JButton`'s `addActionListener` method takes an object that implements the `java.awt.event.ActionListener` interface:

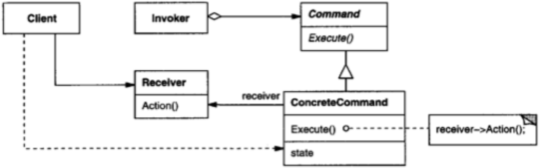
```
public interface ActionListener extends EventListener {

    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);

}
```

`java.awt.event.ActionListener` is an Observer, and `ExitListener` is a ConcreteObserver.

Command (a.k.a. Action, Transaction)

Intent: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Structure



Participants

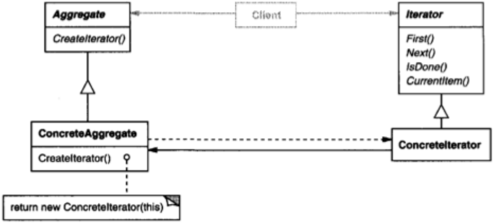
- **Command** declares an interface for executing an operation.
- **ConcreteCommand** defines a binding between a Receiver object and an action; implements Execute by invoking the corresponding operation(s) on Receiver.
- **Client** creates a ConcreteCommand object and sets its receiver.
- **Invoker** asks the command to carry out the request.
- **Receiver** knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

See colorbox for an example of an undoable command.

Iterator (a.k.a. Cursor)

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

Structure



Participants

- **Iterator** defines an interface for 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.

Iterator Example: BST Traversal (1 of 2)

Binary tree implemented as linked nodes:

```
public class BinaryTree<E extends Comparable<E>> implements
    Iterable<E> {

    private class Node<E> {
        E item;
        Node<E> left;
        Node<E> right;

        Node(E item, Node<E> left, Node<E> right) {
            this.item = item;
            this.left = left;
            this.right = right;
        }
    }
...
    private Node<E> root;
...
```

We'd like to allow clients to traverse a BST in a uniform way whether traversing in-order, pre-order, or post-order.

Iterator Example: BST Traversal (2 of 2)

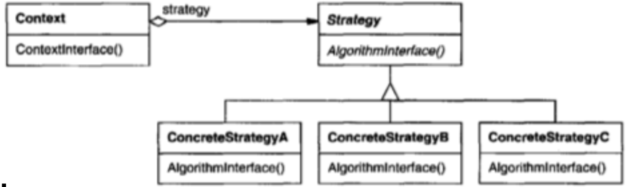
`java.util.Iterator` interface provides a uniform way to traverse all Java collections. Here's an implementation for BST:

```
private class InOrder<E> implements Iterator<E> {
  private Node<E> curNode;
  private Stack<Node<E>> fringe;

  public InOrder(Node<E> root) {
    curNode = root;
    fringe = new LinkedStack<>();
  }
  public boolean hasNext() { ... }

  public E next() {
    while (curNode != null) {
      fringe.push(curNode);
      curNode = curNode.left;
    }
    curNode = fringe.pop();
    E item = curNode.item;
    curNode = curNode.right;
    return item;
  }
  public void remove() { throw new UnsupportedOperationException(); }
```

Strategy (a.k.a. Policy)

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



Participants

- **Strategy** declares an interface common to all supported algorithms.
- **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.

Strategy Example: Repetitive Dives (1 of 4)

When we breath air at depth the increased pressure causes nitrogen to dissolve into body tissues. In SCUBA diving one must be mindful of resudual nitrogen in the body absorbed during a dive.

- On repetitive dives residual nitrogen limits the depth and time allowed on subsequent dives before decompression is required.
- The residual nitrogen in a diver's body is represented by a "pressure group" named by a single letter.
- There are many different ways to calcuate this pressure group: PADI's dive tables, NAUI's dive tables, the U.S. Navy dive tables, and so on.

These tables different *strategies* for calculating pressure groups.

Strategy Example: Repetitive Dives (2 of 3)

We can represent the general **Strategy** for calculating pressure group ofr repetitive dives as an interface:

```
public interface DiveTable {

    public void addDives(SortedSet<Dive> dives);

    public String calculatePressureGroup();
}
```

The PADI table is an example of a **ConcreteStrategy**:

```
public class PadiDiveTable implements DiveTable {

    private SortedSet<Dive> dives;

    public void addDives(SortedSet<Dive> dives) {
        this.dives = dives;
    }

    public String calculatePressureGroup() {
      // calculate using PADI's dive table.
    }
}
```

Strategy Example: Repetitive Dives (3 of 3)

The **Context** in which a `DiveTable` strategy is used is `RepetitiveDives`:

```
public class RepetitiveDives {
    private TreeSet<Dive> dives = new TreeSet<Dive>();

    public void add(Dive dive) {
        dives.add(dive);
    }
    public String calculatePressureGroup(DiveTable diveTable) {
        diveTable.addDives(dives);
        return diveTable.calculatePressureGroup();
    }
}
```

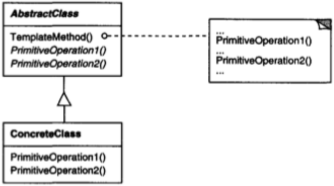
And if we have an instance of `RepetitiveDives` we can calucate the ending pressure group with any concrete strategy:

```
repetitiveDives.calculatePressureGroup(new PadiDiveTable());
// or
repetitiveDives.calculatePressureGroup(new NauiDiveTable());
// and so on ...
```

Template Method

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



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.
- **ConcreteClass** implements the primitive operations to carry out subclass-specific steps of the algorithm.

Template Method Example: Q Learning Agent (1 of 2)

```
class TabularQLearner[WS, MS, A] ... {

  override def getAction(worldState: WS) = { ... }

  override def observe(worldState: WS, action: A, worldNextState: WS)
    = {
    super.observe(worldState, action, worldNextState)
    val state: MS = moduleState(worldState)
    val nextState: MS = moduleState(worldNextState)
    fillInMissingQs(state)
    fillInMissingQs(nextState)
    val r = reward(nextState)
    val maxAction = calcMaxAction(nextState)
    val newVal = q((state, action)) + alpha *
      (r + gamma * q((nextState, maxAction)) - q((state, action)))
    q += ((state, action) -> newVal)
    r
  }
}
```

`observe` is a template method, calling `moduleState` and `reward` methods defined in a subclass

Template Method Example: Q Learning Agent (2 of 2)

```
class FindGoal extends TabularQLearner[ ... ] {

  def moduleState(ws: WumpusState) = FindGoalState(ws.wumpus, ws.goal)

  def actions(ms: FindGoalState) = WumpusAction.values.toIndexedSeq

  def reward(ms: FindGoalState) =
    if (ms.wumpus == ms.goal) 1.0 else -0.4
}
```

`moduleState` and `reward` are “primitive” operations used by the template method defined in the superclass.