BEHAVIORAL DESIGN PATTERNS
BEHAVIORAL DESIGN PATTERNS

• Identifies
  • common communication patterns between objects that realize these patterns

• Describes
  • the objects and classes interact and divide responsibilities among themselves
    • how different objects and classes send messages to each other
    • how the various steps of a task are divided among different objects
DIFFERENCES BETWEEN THE “TYPES” OF PATTERNS

• Creational
  • describe a snapshot of time when objects are created

• Structural
  • describe the structure of a software
  • the structure is more-or-less unchanging

• Behavioral
  • describe the process flow
MAIN BEHAVIORAL PATTERNS

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
</table>
| Chain of Responsibility | - defines a linked list of handlers  
                        | - enables all of the information for a request to be contained within a single object  
                        | - useful when developing domain-specific languages or notations  
                        | - to sequentially access elements of an aggregate object, without exposing its structure  
                        | - removes the need for classes to communicate with each other directly  
                        | - permits the current internal state of an object to be stored  
                        | - allows objects to be linked; changes to one are automatically reflected in others  
                        | - allows an object to change its behavior depending on its current internal state  
                        | - similar algorithms are defined in their own classes; one is selected at run-time  
                        | - defines a group of interchangeable, similarly structured, multi-step algorithms  
                        | - separates structured data from the functionality that may be performed upon it |

CSE 219, Stony Brook University
THE “CHAIN OF RESPONSIBILITY”

• A design pattern that defines a linked list of handlers
  • each of which is able to process requests
  • When a request is submitted to the chain, it is passed to the first handler in the list that is able to process it

• Common problem
  • an event generated by an object needs to be handled by another object
  • the “chain of responsibility” pattern addresses this
THE "CHAIN OF RESPONSIBILITY"

- We have
  - a source of command objects, and
  - series of processing objects
- The command is passed to the first processing object
  - which can handle this command or send it to its successor
  - this chain continues until the command is processed or the end of the chain is reached
- The object sending a command doesn’t know which object will process it!
**Client** passes commands to the first object of the chain.

**HandlerBase**
- an interface or base class for all concrete handlers
- contains a member pointing to the next processing object

**ConcreteHandlers**
- concrete implementation of the HandlerBase class
A REAL-WORLD EXAMPLE

• A vending machine accepts coins
  • rather than having a slot for each type of coin, it has only one for all
  • the inserted coin is sent to the appropriate storage place determined by the receiver
• In this example we have a Coin class with two properties: Diameter and Weight.
  • The Coin class is in this example a command
• Next we have an abstract CoinHandlerBase class with a method setSuccessor()
  • sets the next processing object and an abstract method EvaluateCoin
• For each type of coin (say, 1c, 5c, 10c, 25c) we must implement a handler
• So now we have four handlers
  • each will work with the diameter and weight of the coin
THE COMMAND PATTERN

• A design pattern that
  • enables all of the information for a request to be contained within a single object
  • the command can then be invoked as required
    • often as part of a batch of queued commands with rollback capabilities
• All information needed to execute a method is encapsulated within an object
  • which could be used immediately or held for later use
  • this object doesn’t execute anything, it only includes information
THE COMMAND PATTERN

• Three key terms

  • **Client** created the command object
  • **Invoker** decides when the method which needs information encapsulated within the command object should be called
  • **Receiver** is an instance of the class which contains the method’s code
**Invoker** decides when to invoke the command

**Client** creates the command and links them to receivers

**Receiver** is the class which knows how to perform the operations associated with carrying out the request

CSE 219, Stony Brook University
**CommandBase** is the abstract class (or interface) for all command objects
- It holds information about the receiver responsible for executing an operation
- using information encapsulated within the command object
A REAL-WORLD EXAMPLE

• To control a robot’s movement
• In this example the client is an application
• The receiver of the commands is the robot itself
• The Robot class has four methods for controlling the movement: Move, RotateLeft, RotateRight, TakeSample
• RobotCommandBase is the abstract base class for all concrete command classes
  • It has a protected Robot field which points to the Robot object and abstract methods Move and Undo which must be overridden by the concrete command
• A class RobotController as the invoker
  • It contains two methods
  • ExecuteCommands to execute all commands in a queue
  • UndoCommands to reverse any number of commands as necessary
THE INTERPRETER PATTERN

• A design pattern that is useful when developing domain-specific languages or notations
• Allows the grammar for such a notation to be represented in an object-oriented manner
  • that can easily be extended
THE INTERPRETER PATTERN

• Specifies how to evaluate sentences in a language
• This pattern is described in terms of formal grammars
• It performs some activities based upon a set of expressions
• We have two types of expressions: terminal and non-terminal
• The difference between these types is very simple:
  • terminal expressions represent structures that can be immediately evaluated
  • non-terminal expressions are composed of one or more expressions that in turn, could be terminal or non-terminal
• Helpful in designing compilers, etc.
MAIN BEHAVIORAL PATTERNS

**Chain of Responsibility**
- defines a linked list of handlers

**Command**
- enables all of the information for a request to be contained within a single object
- useful when developing domain-specific languages or notations

**Interpreter**
- to sequentially access elements of an aggregate object, without exposing its structure
- removes the need for classes to communicate with each other directly
- permits the current internal state of an object to be stored

**Observer**
- allows objects to be linked; changes to one are automatically reflected in others

**State**
- allows an object to change its behavior depending on its current internal state

**Strategy**
- similar algorithms are defined in their own classes; one is selected at run-time
- defines a group of interchangeable, similarly structured, multi-step algorithms

**Visitor**
- separates structured data from the functionality that may be performed upon it
THE ITERATOR PATTERN

• A design pattern that provides a means for the elements of an aggregate object to be accessed sequentially
  • without knowledge of its structure
  • allows traversing of lists, trees and other structures in a standard manner
• Aggregate defines the interface for the creation of the iterator object
• The Iterator defines the interface for access and traversal of elements
  • ConcreteIterator implements this interface
    • keeps track of the current position in the traversal
• ConcreteAggregate implements the Aggregate interface
  • returns an instance of the ConcreteIterator
THE ITERATOR PATTERN
WHEN TO USE THE ITERATOR PATTERN?

- Collection – one of the most common data structures
  - [https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)
  - A generic interface
    - A collection can be a list, set, queue, stack, vector, etc.

- Need to access elements without worrying about
  - the type of the elements
  - the internal representation of the collection
    - LinkedList, ArrayList, TreeSet, HashSet

- When you need a uniform traversal interface, and there may be multiple traversals across elements
THE ITERATOR PATTERN IN JAVA

• Java provides a generic interface called Iterator
  • [https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)
  • provides next() and hasNext() methods

• Creating an iterator
  • through a method named iterator() in the container class

```java
List<String> list = new ArrayList<String>();
Iterator it = list.iterator();
while (it.hasNext()) {
    String s = it.next();
}
```
<table>
<thead>
<tr>
<th>Main Behavioral Patterns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain of Responsibility</td>
<td>• defines a linked list of handlers</td>
</tr>
<tr>
<td>Command</td>
<td>• enables all of the information for a request to be contained within a single object</td>
</tr>
<tr>
<td>Interpreter</td>
<td>• useful when developing domain-specific languages or notations</td>
</tr>
<tr>
<td>Iterator</td>
<td>• to sequentially access elements of an aggregate object, without exposing its structure</td>
</tr>
<tr>
<td>Mediator</td>
<td>• removes the need for classes to communicate with each other directly</td>
</tr>
<tr>
<td>Memento</td>
<td>• permits the current <em>internal</em> state of an object to be stored</td>
</tr>
<tr>
<td>Observer</td>
<td>• allows objects to be linked; changes to one are automatically reflected in others</td>
</tr>
<tr>
<td>State</td>
<td>• allows an object to change its behavior depending on its current <em>internal</em> state</td>
</tr>
<tr>
<td>Strategy</td>
<td>• similar algorithms are defined in their own classes; one is selected at run-time</td>
</tr>
<tr>
<td>Template</td>
<td>• defines a group of interchangeable, similarly structured, multi-step algorithms</td>
</tr>
<tr>
<td>Visitor</td>
<td>• separates structured data from the functionality that may be performed upon it</td>
</tr>
</tbody>
</table>
THE MEDIATOR PATTERN

• To handle complex communications between related objects
  • Helps with decoupling of those objects

• A real-world example
  • Air traffic controller
  • One tower (the controller) takes care of who can take off and land, from where
  • *Can you imagine what would and could happen if all airplanes were directly talking to each other trying to coordinate this?*
    • It’s exactly what should never happen!

• The mediator pattern is similarly used
  • to manage algorithms, relationships and responsibilities between objects
THE MEDIATOR PATTERN

- The **Mediator** defines an interface for communication between **Colleague** objects.
- **ConcreteMediator** implements this interface
  - coordinates communication between **Colleagues**
  - it is aware of ALL the colleague objects
- Similarly, we have a **Colleague** interface
  - and its implementation, the **ConcreteColleague**
- The **ConcreteColleague** communicates with other colleagues through the **Mediator**
  - This is how the colleagues are decoupled
WHEN TO USE THE MEDIATOR PATTERN?

• When the communication between objects is complicated, but well defined
• When there are too many relationships between the objects in your code
  • It’s time to think about a central authority figure to control the communications
• A variation of the mediator pattern is used in Java Message Service (JMS) implementations
  • which allows applications to subscribe and publish data to other applications
  • actually, the implementation is a mix of the mediator and observer patterns
MEDIATOR PATTERN: A CHATROOM APPLICATION

//Mediator interface
public interface Mediator {
    public void send(String message, Colleague colleague);
}

//Colleague interface
public abstract Colleague{
    private Mediator mediator;
    public Colleague(Mediator m) { mediator = m; }

    //send a message via the mediator
    public void send(String message) { mediator.send(message, this); }

    //get access to the mediator
    public Mediator getMediator() {return mediator;}
    public abstract void receive(String message);
}
public class ApplicationMediator implements Mediator {
    private ArrayList<Colleague> colleagues;
    public ApplicationMediator() { colleagues = new ArrayList<Colleague>(); }

    public void addColleague(Colleague colleague) { colleagues.add(colleague); }

    public void send(String message, Colleague originator) {
        // tell other screens know that this screen has changed
        colleagues.forEach((Colleague c) -> {
            if (colleague != originator) // no need to tell ourselves
                colleague.receive(message);
        });
    }
}
public class ConcreteColleague extends Colleague {
    public void receive(String message) {
        System.out.println("Colleague Received: " + message);
    }
}

• What if we have different types of colleagues?
  • *One friend chatting from a desktop, while another is using a mobile platform*

public class MobileColleague extends Colleague {
    public void receive(String message) {
        System.out.println("Mobile Colleague Received: " + message);
    }
}
public class Client {
    public static void main(String[] args) {
        ApplicationMediator mediator = new ApplicationMediator();
        ConcreteColleague desktop = new ConcreteColleague(mediator);
        ConcreteColleague mobile = new MobileColleague(mediator);
        mediator.addColleague(desktop);
        mediator.addColleague(mobile);
        desktop.send("Hello world");
        mobile.send("Hello tiny world");
    }
}
MAIN BEHAVIORAL PATTERNS

- **Chain of Responsibility**: defines a linked list of handlers
- **Command**: enables all of the information for a request to be contained within a single object
- **Interpreter**: useful when developing domain-specific languages or notations
- **Iterator**: to sequentially access elements of an aggregate object, without exposing its structure
- **Mediator**: removes the need for classes to communicate with each other directly
- **Memento**: permits the current *internal* state of an object to be stored
- **Observer**: allows objects to be linked; changes to one are automatically reflected in others
- **State**: allows an object to change its behavior depending on its current *internal* state
- **Strategy**: similar algorithms are defined in their own classes; one is selected at run-time
- **Template**: defines a group of interchangeable, similarly structured, multi-step algorithms
- **Visitor**: separates structured data from the functionality that may be performed upon it
THE OBSERVER PATTERN

• Without proper design, the mediator can itself become very complicated
  • In such a scenario, the observer pattern can be useful
  • The colleague objects can deal with events from the mediator
  • Simplifies some of the orchestration (otherwise) required from the mediator

• Of course, there are other uses of the observer pattern
  • In fact, we have all probably already used it even without knowing!
  • The observer pattern is the gold standard for decoupling
    • i.e., to understand how good a pattern is at decoupling, we compare it against the observer pattern
THE OBSERVER PATTERN

```
<<interface>> Subject
+ attach(Observer): void
+ detach(Observer): void
+ notify(): void

Concrete Subject
- subjectState

<<interface>> Observer
+ update(): void

Concrete Observer
- observerState
+ update(): void
```
THE OBSERVER PATTERN

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

• The idea is simple:
  • One or more Observers are interested in the state of a Subject
  • They register their interest in the subject by attaching themselves to it!
  • When something changes in the subject, a notify() message is sent
    • which calls an update() method in each Observer
  • When an observer is no longer interested in the subject
    • it can simply detach itself
FLOW OF ACTION IN THE OBSERVER PATTERN

Subject

attach()

Observer1

attach()

Observer2

Change in state triggers notification

notify()

notify()
WHEN TO USE THE OBSERVER PATTERN?

• To pass data onto the observers
  • the subject doesn't need to know who needs to know!

• Everything is done through a common interface
  • `notify()` just calls all the objects out there that have registered their interest
  • excellent decoupling
    • any object can simply implement the Observer interface and get updates from the Subject

• In general, you want to use this pattern to reduce coupling
  • If you have an object that needs to share it's state with others, without knowing who those objects are
THE OBSERVER PATTERN IN JAVA

• Java has already defined the interface for us:
  • [https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html](https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html)

• The “Subject” is also defined, as the Observable class:
  • [https://docs.oracle.com/javase/8/docs/api/java/util/Observable.html](https://docs.oracle.com/javase/8/docs/api/java/util/Observable.html)
Consider a situation where you have data that needs to be dynamically shown on screen

- The “subject” will be a class called DataStore
  - extends java.util.Observable
- The “observer” will be a Screen class
  - implements java.util.Observer
import java.util.Observable;

public class DataStore extends Observable {
    private String data;

    public String getData() { return data; }

    public void setData(String data) {
        this.data = data;
        setChanged(); // mark the observable as changed
    }
}

• Call the setChanged() method of the Observable
  • required in order for the call to notify observers to send out the update
  • without this the Observable will see no reason to send out the update
**OBSERVER PATTERN: A JAVA APPLICATION**

```java
public class Screen implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        ... // do stuff
    }
}
```

Create our observer

```java
Screen screen = new Screen();
DataStore dataStore = new DataStore();
dataStore.addObserver(screen); // attach observer
```

Add it to the Screen

```java
dataStore.notifyObservers(); // send a notification
```
CAVEATS FOR THE OBSERVER PATTERN

• Additional (optional) reading
  • [http://martinfowler.com/eaaDev/OrganizingPresentations.html#observer-gotchas](http://martinfowler.com/eaaDev/OrganizingPresentations.html#observer-gotchas)

• Main potential downsides
  • Don’t have chains of observers, i.e., observers that are subjects of other observers
  • Watch out for memory leaks!
    • Subjects will continue holding references to observers unless and until they are explicitly detached
# MAIN BEHAVIORAL PATTERNS

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain of Responsibility</td>
<td>defines a linked list of handlers</td>
</tr>
<tr>
<td>Command</td>
<td>enables all of the information for a request to be contained within a single object</td>
</tr>
<tr>
<td>Interpreter</td>
<td>useful when developing domain-specific languages or notations</td>
</tr>
<tr>
<td>Iterator</td>
<td>to sequentially access elements of an aggregate object, without exposing its structure</td>
</tr>
<tr>
<td>Mediator</td>
<td>removes the need for classes to communicate with each other directly</td>
</tr>
<tr>
<td>Memento</td>
<td>permits the current <em>internal</em> state of an object to be stored</td>
</tr>
<tr>
<td>Observer</td>
<td>allows objects to be linked; changes to one are automatically reflected in others</td>
</tr>
<tr>
<td>State</td>
<td>allows an object to change its behavior depending on its current <em>internal</em> state</td>
</tr>
<tr>
<td>Strategy</td>
<td>similar algorithms are defined in their own classes; one is selected at run-time</td>
</tr>
<tr>
<td>Template</td>
<td>defines a group of interchangeable, similarly structured, multi-step algorithms</td>
</tr>
<tr>
<td>Visitor</td>
<td>separates structured data from the functionality that may be performed upon it</td>
</tr>
</tbody>
</table>
MEMENTO PATTERN

• Used in undo frameworks to bring an object back to a previous state

1. The **Originator** is the object that knows how to save itself
   • this is the class with internal ‘states’
2. The **Caretaker** is that object that deals with the when (and why) of the originator saving/restoring itself
3. The **Memento** holds the information about the originator’s state
   • cannot be modified by the caretaker

**Flow of events**
• The **Caretaker** asks the **Originator** for the **Memento** object
  • and performs any actions that it needs to
• To rollback the state before these actions, it returns the memento object to the originator
• Memento needs to save editor contents
  • say, just plain text

```java
public class EditorMemento {

    private final String editorState;

    public EditorMemento(String state) {
        editorState = state;
    }

    public String getSavedState() {
        return editorState;
    }
}
```
The Memento Pattern in Java: An Editor

- The Editor is our “originator” class
  - it can use the memento

```java
public class Editor {
    public String editorContents; // this is the ‘state’ of the editor

    public void setState(String contents) {
        this.editorContents = contents;
    }

    public EditorMemento save() { return new EditorMemento(editorContents); }

    public void restoreToState(EditorMemento memento) {
        editorContents = memento.getSavedState();
    }
}
```

CSE 219, Stony Brook University
### MAIN BEHAVIORAL PATTERNS

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Chain of Responsibility</strong></td>
<td>defines a linked list of handlers</td>
</tr>
<tr>
<td><strong>Command</strong></td>
<td>enables all of the information for a request to be contained within a single object</td>
</tr>
<tr>
<td><strong>Interpreter</strong></td>
<td>useful when developing domain-specific languages or notations</td>
</tr>
<tr>
<td><strong>Iterator</strong></td>
<td>to sequentially access elements of an aggregate object, without exposing its structure</td>
</tr>
<tr>
<td><strong>Mediator</strong></td>
<td>removes the need for classes to communicate with each other directly</td>
</tr>
<tr>
<td><strong>Memento</strong></td>
<td>permits the current <em>internal</em> state of an object to be stored</td>
</tr>
<tr>
<td><strong>Observer</strong></td>
<td>allows objects to be linked; changes to one are automatically reflected in others</td>
</tr>
<tr>
<td><strong>State</strong></td>
<td>allows an object to change its behavior depending on its current <em>internal</em> state</td>
</tr>
<tr>
<td><strong>Strategy</strong></td>
<td>similar algorithms are defined in their own classes; one is selected at run-time</td>
</tr>
<tr>
<td><strong>Template</strong></td>
<td>defines a group of interchangeable, similarly structured, multi-step algorithms</td>
</tr>
<tr>
<td><strong>Visitor</strong></td>
<td>separates structured data from the functionality that may be performed upon it</td>
</tr>
</tbody>
</table>

CSE 219, Stony Brook University
THE STATE PATTERN

- Allows objects to behave in different ways depending on internal state
- Used when you need a class to behave differently some times
- E.g., performing slightly different computations, based on some arguments passed through to the class
• The **Context** can have a number of internal **States**
  • whenever the `request()` method is called on the context
  • the message is delegated to the State to handle

• The **State** interface defines a common interface for all concrete states
  • encapsulating all behavior associated with a particular state
• When a **Context** changes state
  • what really happens is that we have a different “concrete state” associated with it
THE STATE PATTERN: A MUSIC PLAYER

//The context
public class MP3PlayerContext {
    private State state;
    private MP3PlayerContext(State state) { this.state = state; }
    public void play() { state.pressPlay(this); }
    public void setState(State state) { this.state = state; }
    public State getState() { return state; }
}

//The state interface
private interface State {
    public void pressPlay(MP3PlayerContext context);
}
THE STATE PATTERN: A MUSIC PLAYER

// Different “concrete” state implementations

public class StandbyState implements State {
    public void pressPlay(MP3PlayerContext context) {
        context.setState(new PlayingState());
    }
}

public class PlayingState implements State {
    public void pressPlay(MP3PlayerContext context) {
        context.setState(new StandbyState());
    }
}
THE STRATEGY PATTERN

• Useful in changing algorithm implementations at runtime
  • without causing tight coupling
    • it’s all about dynamically changing behavior

• A Context is composed of a Strategy
  • The context could be anything that would require changing behaviors (e.g., a class that provides sorting functionality)

• The Strategy is simply implemented as an interface, so that we can swap ConcreteStrategy objects in and out without effecting our Context
WHERE TO USE THE STRATEGY PATTERN?

• Where you want to choose the algorithm to use at runtime
  • saving files in different formats
  • running various sorting algorithms
  • file compression

• Provides a way to define a “family” of algorithms
  • encapsulate each one as an object, and
  • make them interchangeable
THE STRATEGY PATTERN: FILE COMPRESSION

//Strategy Interface
public interface CompressionStrategy {
    public void compressFiles(ArrayList<File> files);
}

class ZipCompressionStrategy implements CompressionStrategy {
    public void compressFiles(ArrayList<File> files) {
        //using ZIP approach
    }
}

class RarCompressionStrategy implements CompressionStrategy {
    public void compressFiles(ArrayList<File> files) {
        //using RAR approach
    }
}
public class CompressionContext {

    private CompressionStrategy strategy;
    // can be set at runtime by the application preferences
    public void setCompressionStrategy(CompressionStrategy strategy) {
        this.strategy = strategy;
    }

    // use the strategy
    public void createArchive(ArrayList<File> files) {
        strategy.compressFiles(files);
    }
}

THE STRATEGY PATTERN: FILE COMPRESSION
**MAIN BEHAVIORAL PATTERNS**

- **Chain of Responsibility**
  - defines a linked list of handlers

- **Command**
  - enables all of the information for a request to be contained within a single object
  - useful when developing domain-specific languages or notations
  - to sequentially access elements of an aggregate object, without exposing its structure
  - removes the need for classes to communicate with each other directly
  - permits the current *internal* state of an object to be stored
  - allows objects to be linked; changes to one are automatically reflected in others
  - allows an object to change its behavior depending on its current *internal* state
  - similar algorithms are defined in their own classes; one is selected at run-time
  - defines a group of interchangeable, similarly structured, multi-step algorithms
  - separates structured data from the functionality that may be performed upon it
TEMPLATE PATTERN

• Used when two or more implementations of a similar algorithm exist
• In the real world, this pattern gets used all the time
  • architecture, engineering, etc.
• The UML class diagram is simple:
  • The AbstractClass contains the templateMethod() method
    • should be made final so that it cannot be overridden
    • this method makes use of other operations available in order to run the algorithm
    • but is decoupled for the actual implementation of these methods
    • all operations used by this template method are made abstract, so their implementation is deferred to subclasses
• Used when two or more implementations of a similar algorithm exist
• In the real world, this pattern gets used all the time
  • architecture, engineering, etc.
• The UML class diagram is simple:
  • The **ConcreteClass** implements all the operations required by the **templateMethod**
    • these were defined as abstract in the parent class
    • there can be many different concrete classes
The template method in the parent class controls the overall process

- “calling” subclass methods when necessary
- avoids low level components depending on high level components
- instead give these low level classes (ConcreteClass) a way of hooking into the parent class (AbstractClass)

There are four different types of methods used in the parent class:

1. **Concrete methods**: standard complete methods that are useful to the subclasses. These methods are usually utility methods.
2. **Abstract methods**: Methods containing no implementation that must be implemented in subclasses
3. **Hook methods**: Methods containing a default implementation that may be overridden in some classes.
   - Hook methods are intended to be overridden, concrete methods are not.
4. **Template methods**: A method that calls any of the methods listed above in order to describe the algorithm without needing to implement the details.
**MAIN BEHAVIORAL PATTERNS**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain of Responsibility</td>
<td>defines a linked list of handlers</td>
</tr>
<tr>
<td>Command</td>
<td>enables all of the information for a request to be contained within a single object</td>
</tr>
<tr>
<td>Interpreter</td>
<td>useful when developing domain-specific languages or notations</td>
</tr>
<tr>
<td>Iterator</td>
<td>to sequentially access elements of an aggregate object, without exposing its structure</td>
</tr>
<tr>
<td>Mediator</td>
<td>removes the need for classes to communicate with each other directly</td>
</tr>
<tr>
<td>Memento</td>
<td>permits the current <em>internal</em> state of an object to be stored</td>
</tr>
<tr>
<td>Observer</td>
<td>allows objects to be linked; changes to one are automatically reflected in others</td>
</tr>
<tr>
<td>State</td>
<td>allows an object to change its behavior depending on its current <em>internal</em> state</td>
</tr>
<tr>
<td>Strategy</td>
<td>similar algorithms are defined in their own classes; one is selected at run-time</td>
</tr>
<tr>
<td>Template</td>
<td>defines a group of interchangeable, similarly structured, multi-step algorithms</td>
</tr>
<tr>
<td>Visitor</td>
<td>separates structured data from the functionality that may be performed upon it</td>
</tr>
</tbody>
</table>
THE VISITOR PATTERN

• Of all the patterns we have seen, this is probably the most powerful
  • upside: it’s also rather convenient!

• Real-world analogies
  1. Calling a taxi
     • Customer calls a taxi, which arrives at his/her door
     • Once the person enters the vehicle that taxi is in control of the transport of that person
  2. Shopping in a supermarket
     • A cart full of items
     • At checkout, the cashier is a ‘visitor’ responsible for going through the items and giving you a total
THE VISITOR PATTERN

• Allows for one or more operation to be applied to a set of objects
  • at runtime
  • decoupling the operations from the object structure

• This pattern actually creates an external class
  • This external class then uses the data in the other classes
  • Helpful if you want to perform a set of operations on a disparate collection of objects
  • Can also provide additional functionality to a class without actually changing it!
THE VISITOR PATTERN

- The core is the **Visitor** interface
  - defines a “visit” operation for each type of element
- The **ConcreteVisitor**
  - implements the operations defined in the **Visitor** interface
  - stores local state as it traverses the elements
- The **Element** interface
  - simply defines an **accept** method to allow the visitor to run some action over that element
WHEN TO USE THE VISITOR PATTERN?

• When you have operations to perform across a variety of objects
  • these operations could be distinct and unrelated
  • the objects could also be very different
    • e.g., objects in a shopping cart, each implementing a different interface

• Advantages
  • avoids adding code throughout your object structure
    • so the object structure has cleaner code!
  • decouples the code for logical operations from the structure of the elements used in those operations
THE VISITOR PATTERN IN JAVA: A MAILING SYSTEM

• Set of elements
  • items in a shopping cart in an online market like Amazon

• Operation
  • to determine the cost of mailing each item
  • where the cost depends on
    1. type of item
    2. weight of item
    3. shipping destination
• We will create a separate visitor for each postal region
  • this way, the logic of calculating the cost is decoupled from the item itself
  • … which makes sense
    • a “book” should remain just a book, and not have any code in itself that pertains to the shipping cost!

    // The Element interface
    public interface Visitable {

        public void accept(Visitor visitor);
    }

// A concrete element class
public class Book implements Visitable {
    private double price;
    private double weight;

    // accept the visitor
    public void accept(Visitor visitor) { visitor.visit(this); } 

    public double getPrice() { return price; } 

    public double getWeight() { return weight; } 
}

THE VISITOR PATTERN IN JAVA: A MAILING SYSTEM
//A concrete element class
public class Book implements Visitable {
    private double price;
    private double weight;

    //accept the visitor
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    public double getPrice() { return price; }

    public double getWeight() { return weight; }
}
THE VISITOR PATTERN IN JAVA: A MAILING SYSTEM

• Now for the visitor interface
  • for each concrete element, we need a visit method

```java
public interface Visitor {
    public void visit(Book book);
    public void visit(CD cd);
    public void visit(DVD dvd);
}
```
The implementation specifies the details of the actual operation.

```java
public class ShippingVisitor implements Visitor {
    private double totalShippingCost;

    public void visit(Book book) {
        if (book.getPrice() < 10.0) // free shipping for a book over $10
            totalShippingCost += book.getWeight() * 2;
    }

    public void visit(CD cd) {...} // similarly for any other concrete element

    // return the internal state
    public double getTotalShipping() { return totalShippingCost; }
}
```
THE VISITOR PATTERN IN JAVA: A MAILING SYSTEM

• The point is to have all the calculations *away* from the actual item definitions, and in one central place.

• Finally, we need a way to iterate through the shopping cart:

```java
public class ShoppingCart {
    private ArrayList<Visitable> items; // shopping cart with items
    public double calculateShipping() {
        ShippingVisitor visitor = new ShippingVisitor();
        for(Visitable item: items)
            item.accept(visitor);

        return getTotalShipping();
    }
}
```
The arguments and return types for the visiting methods need to be known in advance.

so the Visitor pattern is not good for situations where these visited classes can change.

Every time a new type of Element is added, every Visitor derived class must be modified!

It can be difficult to refactor the Visitor pattern into code that wasn't already designed with the pattern in mind.