Foundations of Software Engineering
Behavioral Design Patterns

Fall 2017
Department of Computer Science Ben-Gurion university

Based on slides of:

Nurit Gal-oz, Department of Computer Science Ben-Gurion University
Mira Balaban Department of Computer Science Ben-Gurion university
F. Tip. IBM T J Watson Research Center.
Behavioral Patterns

- Concerned with algorithms and the assignment of responsibilities between objects
- Behavioral class patterns use inheritance to distribute behavior between classes
- Behavioral object patterns use composition to distribute behavior between objects

Chain of Responsibility  Observer
Command  State
Interpreter  Strategy
Iterator  Template Method
Mediator  Visitor
Memento
Iterator
Iterator: Motivation

- Accessing the elements of an aggregate object without exposing its internal structure.

- Traverse the aggregate in different ways, depending on needs.

- Do not want to bloat the aggregate interface with operations for different traversals, even if they can be anticipated.

- Need to have more than one traversal pending on the same aggregate.
Iterator: Solution

- **Key idea:** Take the responsibility for access and traversal out of the aggregate object and put it into an `Iterator` object.

- The list objects are responsible for creating their corresponding iterator.
Iterator: Class Diagram

```
return new ConcreteIterator(this);
```
Iterator: 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
Iterator: intent and context

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

• apply *Iterator* for the following purposes:
  • to access an aggregate object’s contents without exposing its internal representation
  • to support multiple traversals of aggregate objects
  • to provide a uniform interface for traversing different aggregate structures (support polymorphic iteration)
Iterator Example: Directory traversal

• use *Iterator* to allow clients to iterate through the *Files* in a *Directory*
  
• without exposing *Directory’s* internal structure to the client
Interface Iterator

interface Iterator {
    public void first(); // set to first
    public void next();  // advance
    public boolean isDone(); // is done
    public Object current(); // get current
}
class Directory (1)

class Directory extends Node {
    ...

    public Iterator iterator() {
        return new DirectoryIterator(this);
    }

    // use a private inner class because:
    // - it is not visible outside the class
    // - its methods have access to Directory's
    //     private field _children

    private class DirectoryIterator implements Iterator {
        private Vector _files;
        private int _fileCnt;

        DirectoryIterator(Directory d) {
            _files = d._children;
            _fileCnt = 0;
        }

        ...
    }
}
class Directory (2)

public void first()
{
    _fileCnt = 0;
}

public void next()
{
    _fileCnt++;
}

public boolean isDone()
{
    return _fileCnt == _files.size();
}

public Object current()
{
    return _files.elementAt(_fileCnt);
}
}
public class Main {
    public static void main(String[] args) {
        Directory root = new Directory("");
        File core = new File("core", root, "hello");
        Directory usr = new Directory("usr", root);
        File adm = new File("adm", usr, "there");
        Directory foo = new Directory("foo", usr);
        File bar1 = new File("bar1", foo, "abcdef");
        File bar2 = new File("xbar2", foo, "abcdef");
        File bar3 = new File("yybarzz3", foo, "abcdef");

        // use iterator to print contents of /usr
        Iterator it = usr.iterator();
        for (it.first(); !it.isDone(); it.next()) {
            Node n = (Node) it.current();
            System.out.println(n.getAbsoluteName());
        }
    }
}
Output

/usr/adm
/usr/foo/
Observer
(aka. Listener, or Publish-Subscribe)

The subject contains the state. So, whenever the state changes, observers need to be notified.

Register and UnRegister the observers

Once the notification is received from subject, observers call the subject and get data that is changed.
Observer: Motivation

- A spreadsheet object and bar chart are different presentations of the same application data object.
- The data object need not to know about them.
- The different presentations do not know about each other.
- The presentations should be **notified** about changes in the data object.
Observer: Solution

- Key objects: *subject* and *observer*.
- A subject may have any number of dependent observers.
- All observers are notified whenever the subject changes its state.
- Each observer can query the subject to synchronize their states.
Observer: Class Diagram

```
for all o in observers {
    o.update();
}
```

calls subject.getState() to retrieve state of the subject
Observer: Participants

- **Subject**
  - knows its observers. any number of observers may observe a subject
  - provides an interface for attaching/detaching observers
- **Observer**
  - defines an updating interface for objects that should be notified of changes
- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects
  - sends a notification to its observers when state changes
- **ConcreteObserver**
  - maintains reference to a ConcreteSubject object
  - stores state that should stay consistent with subject’s
  - implements the Observer updating interface to keep its state consistent with the subject’s
Observer: Sequence Diagram
Observer: intent and context

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

- apply *Observer* when
  - an abstraction has two aspects, one dependent on the other.
  - a change to one object requires changing others.
  - object should be able to notify other objects without making assumptions about the identity of these objects.
Observer Example

- add `FileObservers` to our `FileSystem` example.
  - add a method `write(String)` to class `File` to model operations that change a `File`’s contents
  - associate `FileObservers` with `Files`; notify these after each write
  - `FileObservers` print a warning message that the file has changed
interface Observer {
    public void update();
}

class FileObserver implements Observer {
    FileObserver(File f){
        f.attach(this);
        _subject = f;
    }
    public void update(){
        System.out.println("file "+
         _subject.getAbsolutePath() +
         " has changed.");
    }
    private File _subject;
}
class File extends Node {
    File(String n, Directory p, String c) {
        super(n, p);
        _contents = c;
    }

    public void attach(Observer o) {
        if (!_observers.contains(o)) {
            _observers.add(o);
        }
    }

    public void detach(Observer o) {
        _observers.remove(o);
    }

    ...
}
public void notifyObservers(){
    for (int t=0; t < _observers.size(); t++){
        ((Observer)_observers.elementAt(t)).update();
    }
}

public void write(String s){
    _contents = s;
    notifyObservers();
}

private String _contents;
private Vector _observers = new Vector();
public class Main {
    public static void main(String[] args) {
        Directory root = new Directory("\n");
        File core = new File("core", root, "hello");
        Directory usr = new Directory("usr", root);
        File bar1 = new File("bar1", usr, "abcdef");

        // create observer for file bar1
        FileObserver obs = new FileObserver(bar1);
        bar1.write("abracadabra");
        bar1.write("fffff");
        bar1.write("gggggg");
    }
}

Output

- `file /usr/barl` has changed.
- `file /usr/barl` has changed.
- `file /usr/barl` has changed.
Observer: Considerations (1)

- Sometimes observers need to observe more than one subject.
- who triggers the update?
  - state-changing subject methods call notify() method, or
  - make clients responsible for calling notify().
- avoid dangling references when deleting subjects
  -- Subject notifies its observers about its deletion.
- make sure Subject’s state is self-consistent before calling notify (the observers will query the state).
Observer: Considerations (2)

- avoiding observer-specific update protocols
  - **push model**: subject sends its observers detailed information about the changes
  - **pull model**: subject only informs observers that state has changed; observers need to query subject to find out what has changed

- specifying modifications of interest explicitly
  - of interest when *observer* are interested in only some of the state-changing events:
    - `Subject.attach(Observer, interest)`
    - `Observer.update(Subject, interest)`

- encapsulating complex update semantics
  - when there is a highly complex relationship between subject and observer, introduce a ChangeManager class to reduce the amount of work.
State
State: Motivation

- **TCPConnection** that represents a network connection.
- A **TCPConnection** object can be in one of several different states: *Established, Listening, Closed*.
- **Problem**: A **TCPConnection** object responds differently to requests, depending on its current state.
State: Solution

- Key idea: Introduce an abstract class `TCPState` to represent the states of the network connection.
State: Class Diagram
State: Participants

- **Context**
  - defines interface of interest to clients
  - maintains reference to a *ConcreteState* subclass that defines the current state

- **State**
  - defines an interface for encapsulating the behavior associated with a particular state of the *Context*

- **ConcreteState** subclasses
  - each subclass implements a behavior associate with a state of the *Context* (by overriding methods in *State*)
State: Intent and context

- Allow an object to change its behavior when its internal state changes
- use *State* when:
  - an object’s behavior depends on its state
  - operations have large conditional statements that depend on the object’s state (the state is usually represented by one or more enumerated constants)
State: Example

- example of a vending machine:
  - product price is $0.25
  - machine accepts any combination of nickels, dimes, and quarters
  - customer enters coins; when credit reaches $0.25 product is dispensed, and refund is given for the remaining credit.
  - machine has display that shows the current balance
Statechart diagram of Vending Machine

- State: Credit0
  - Transitions:
    - addQuarter /dispenseProduct
    - addDime /dispenseProduct /refund5
    - addNickel
    - addDime

- State: Credit5
  - Transitions:
    - addQuarter /dispenseProduct /refund5
    - addNickel
    - addDime
    - addNickel

- State: Credit10
  - Transitions:
    - addQuarter /dispenseProduct /refund10
    - addDime
    - addDime
    - addNickel
    - addNickel

- State: Credit20
  - Transitions:
    - addQuarter /dispenseProduct /refund20
    - addDime
    - addNickel
    - addNickel

- State: Credit15
  - Transitions:
    - addDime /DispenseProduct
    - addNickel()
“Traditional” implementation

• use integers to represent the states
  • more complex states may require objects and “enumerated types”
• methods `addNickel()`, `addDime()`, and `addQuarter()` to model user actions
• methods `refund()`, `displayBalance()`, and `dispenseProduct()` to model system’s actions
• conditional logic (with `if/switch` statements) depending on current state
Traditional implementation (1)

```java
class TraditionalVendingMachine {
    private int _balance;
    public TraditionalVendingMachine() {
        _balance = 0; welcome();
    }
    void welcome() {
        System.out.println("Welcome. Please enter $0.25 to buy product.");
    }
    void dispenseProduct() {
        System.out.println("dispensing product...");
    }
    void displayBalance() {
        System.out.println("balance is now: " + _balance);
    }
    void refund(int i) {
        System.out.println("refunding: " + i);
    }
    ...
}
```
public void addNickel() {
    switch (_balance) {
    case 0: {
        _balance = 5;
        displayBalance();
        break;
    }
    case 5: {
        _balance = 10;
        displayBalance();
        break;
    }
    case 10: {
        _balance = 15;
        displayBalance();
        break;
    }
    case 15: {
        _balance = 20;
        displayBalance();
        break;
    }
    case 20: {
        dispenseProduct();
        _balance = 0; welcome();
        break;
    }
    }
}
public void addDime() {
    switch (_balance) {
    case 0 : { _balance = 10;
                    displayBalance();
                    break; }
    case 5 : { _balance = 15;
                    displayBalance();
                    break; }
    case 10 : { _balance = 20;
                    displayBalance();
                    break; }
    case 15 : { dispenseProduct();
                    _balance = 0; welcome();
                    break; }
    case 20 : { dispenseProduct();
                    refund(5); _balance = 0; welcome();
                    break; }
    }
}
Traditional implementation: client code

```java
public class Client {

    public static void main(String[] args) {
        VendingMachine v = new VendingMachine();
        v.addNickel();
        v.addDime();
        v.addNickel();
        v.addQuarter();
    }
}
```
Observations

- state-specific behavior scattered over different conditionals
  - changing one state’s behavior requires visiting each of these
- inflexible: adding a state requires invasive change (editing each conditional)
- approach tends to lead to large classes
  - not clear how to partition functionality
Using the State pattern (1)

```java
interface VendingMachineState {
    public void addNickel(VendingMachine v);
    public void addDime(VendingMachine v);
    public void addQuarter(VendingMachine v);
    public int getBalance();
}
```
Example of a ConcreteState

class Credit0 implements VendingMachineState {
    private Credit0() { }
    private static Credit0 _theInstance;
    static Credit0 instance(VendingMachine v) {
        if (_theInstance == null) {
            _theInstance = new Credit0();
        }
        v.welcome(); return _theInstance;
    }
    public void addNickel(VendingMachine v) {
        v.changeState(Credit5.instance());
    }
    public void addDime(VendingMachine v) {
        v.changeState(Credit10.instance());
    }
    public void addQuarter(VendingMachine v) {
        v.dispenseProduct();
        v.changeState(Credit0.instance(v));
    }
    public int getBalance() { return 0; }
}
class Credit10 implements VendingMachineState {
    private Credit10(){ } 
    private static Credit10 _theInstance;
    static Credit10 instance(){
        if (_theInstance == null){
            _theInstance = new Credit10();
        }
        return _theInstance;
    }
    public void addNickel(VendingMachine v) {
        v.changeState(Credit15.instance());
    }
    public void addDime(VendingMachine v) {
        v.changeState(Credit20.instance());
    }
    public void addQuarter(VendingMachine v) {
        v.dispenseProduct();
        v.refund(10);
        v.changeState(Credit0.instance(v));
    }
    public int getBalance(){ return 10; }
}
public class VendingMachine {
    public VendingMachine() {
        _state = Credit0.instance(this);
    }
    // methods welcome(), dispenseProduct() etc.
    // same as before
    void changeState(VendingMachineState s) {
        _state = s; displayBalance();
    }
    public void addNickel() { _state.addNickel(this); }  
    public void addDime() { _state.addDime(this); }  
    public void addQuarter() { _state.addQuarter(this); }  
    private VendingMachineState _state;
}

State: Benefits

- localizes state-specific behavior, and partitions behavior for different states
  - leads to several small classes instead of one large class
  - natural way of partitioning the code
- avoids (long) if/switch statements with state-specific control flow
  - also more extensible---you don’t have to edit your switch statements after adding a new state
- makes state transitions explicit
  - simply create a new `ConcreteState` object, and assign it to the state field in `Context`
- `state`-objects can be shared
  - and common functionality can be placed in abstract class `State`
State: Implementation Issues

- who defines the state transitions?
  - not defined by the pattern
  - usually done by the various `ConcreteStates`
    - add an operation to `Context` for setting the state

- when to create `ConcreteStates`?
  - on demand or ahead-of-time
  - choice depends on how often `ConcreteStates` get created, and cost of creating them
  - can use `Singleton` or `Flyweight` if `ConcreteStates` don’t have any fields
Strategy
Strategy: Motivation

- Breaking a stream of text into lines.
- Many algorithms.
- Hard-wiring all such algorithms into the client classes isn't desirable:
  - Clients get more complex, harder to maintain.
  - No need to support multiple algorithms if not used.
  - Difficult to add algorithms and vary existing ones when they are an integral part of a client.

- a.k.a Policy
Strategy: Solution

- Define classes that encapsulate different linebreaking algorithms -- a strategy.

  **SimpleCompositor**: implements a simple strategy that determines linebreaks one at a time.
  **TeXCompositor**: implements the TeX algorithm for finding linebreaks. This strategy tries to optimize linebreaks globally, that is, one paragraph at a time.
  **ArrayCompositor**: implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.

- Composition class is responsible for maintaining and updating the linebreaks of text displayed in a text viewer.
Strategy: Class diagram

```
Context
  +contextInterface()

Strategy
  +algorithmInterface()

ConcreteStrategy1
  +algorithmInterface()

ConcreteStrategy2
  +algorithmInterface()
```

- Context
- Strategy
- ConcreteStrategy1
- ConcreteStrategy2
Strategy: Participants

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

- **ConcreteStrategy**
  - implements the interface declared in `Strategy`

- **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: Intent and context

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

- Use *Strategy* when:
  - you need different variants of an algorithm (e.g. with different time/space tradeoffs)
  - you want to avoid exposing details/data structures of an algorithm that clients shouldn’t know about
Strategy: Example

• method `Warehouse.searchByAuthor()` from an implementation of a book-selling system
  • computes a `Vector` of `Books`
  • sorts this `Vector` by calling `BubbleSorter.sort()`, which implements bubble-sort
  • then returns an `Iterator` over this `Vector`

• This design hard-wires the choice of a specific sorting algorithm
public Iterator searchByAuthor(String name) {
  Vector results = new Vector();

  for (int i = 0; i < _theBooks.size(); i++) {
    BookInfo bookInfo = (BookInfo) _theBooks.elementAt(i);
    Book book = bookInfo.getBook();
    String authorLastName = book.getLastName();
    String otherAuthors = book.getOtherAuthors();
    if ((authorLastName.indexOf(name) != -1) ||
        (otherAuthors != null &&
        otherAuthors.indexOf(name) != -1)) {
      results.addElement(book);
    }
  }
  BubbleSorter.sort(results);
  return new SearchResultIterator(results);
}
public class BubbleSorter {
    public static void sort(Vector books){
        for (int i=0; i < books.size(); i++){
            for (int j=books.size()-1; j > i; j--){
                if (compare(books, j, j-1)){
                    swap(books, j, j-1);
                }
            }
        }
    }

    public static boolean compare(Vector books, int i, int j){
        Book b1 = (Book)books.elementAt(i);
        Book b2 = (Book)books.elementAt(j);
        if (b1.getTitle().compareTo(b2.getTitle()) < 0){
            return true;
        }
        return false;
    }

    public static void swap(Vector books, int i, int j){...
}
Applying the Strategy pattern

• Avoid hard-wiring a specific sorting algorithm in the *Warehouse* as follows:
  • define interface *Sorter*, with method *sort(Vector)*
  • make *BubbleSorter* a subclass of *Sorter*, and override method *sort(Vector)*
  • add parameter of type *Sorter* to method *Warehouse.searchByAuthor()*
  • choice of sorting algorithm can now be made elsewhere (e.g., in the *Driver* component) and varied at run-time
  • can now easily adopt another sorting routine by creating another class that implements the *Sorter* interface (e.g., *MergeSorter*)
Revised Example (1)

```java
public Iterator searchByAuthor(String name, Sorter sorter) {
    Vector results = new Vector();
    for (int i = 0; i < theBooks.size(); i++) {
        BookInfo bookInfo = (BookInfo) theBooks.elementAt(i);
        Book book = bookInfo.getBook();
        String authorLastName = book.getLastName();
        String otherAuthors = book.getOtherAuthors();
        if ((authorLastName.indexOf(name) != -1)
            || (otherAuthors != null &&
                otherAuthors.indexOf(name) != -1)) {
            results.addElement(book);
        }
    }
    sorter.sort(results);
    return new SearchResultIterator(results);
}
```
Revised Example (2)

```java
public interface Sorter {
    public void sort(Vector v);
}

public class BubbleSorter implements Sorter {
    public void sort(Vector books){
        for (int i=0; i < books.size(); i++){
            for (int j=books.size()-1; j > i; j--){
                if (compare(books, j, j-1)){
                    swap(books, j, j-1);
                }
            }
        }
    }
}

public class MergeSorter implements Sorter {
    ...
}
```
Strategy: Considerations

- suitable for families of algorithms with similar interfaces
- avoids subclassing and conditional statements of the Context hierarchy
- Clients must be aware of different strategies and select one of them
- performance penalty:
  - additional overhead of communication between Strategy and Context
  - increased number of objects
Visitor

Visitor
Visitor: Motivation

- A compiler that represents a program as abstract syntax tree.
- The set of node classes is fixed for a language.

- Applies operations like **type-checking, code optimization, flow analysis, checking for variables being assigned values before they're used** to all nodes.
- Operations might change – depend on static semantic analysis.
Visitor: Problem

- **Distributing** all operations across node classes leads to a system that's hard to understand, maintain, and change.
  - It is confusing to have *type-checking* code mixed with pretty-printing code or flow analysis code.
- Adding a new operation requires recompiling all of these classes.
- It would be better if:
  - each new operation could be added separately
  - the node classes were independent of the operations that apply to them.
Visitor: Solution

- Package related operations in a separate object, called a *visitor*.
- Passing it to elements of the abstract syntax tree as it's traversed.
- When an element "accepts *visitor*", it sends a request to that *visitor*, that includes the element as an argument.
- The *visitor* executes the operation for that element—the operation that used to be in the class of the element.
Visitor: Solution

![Diagram showing the Visitor pattern with classes AssignmentNode and VariableRefNode with their respective visit methods.]
Visitor: Class Diagram

Client

Visitor

ConcreteVisitor1

ConcreteVisitor2

Element

ConcreteElementA

ConcreteElementB

ObjectStructure

Visitor

ConcreteVisitor1

ConcreteVisitor2

ConcreteElementA

ConcreteElementB

Accept(v : Visitor)

Accept(v : Visitor)

OperationA()

OperationB()

v.VisitConcreteVisitorA(this)

v.VisitConcreteVisitorB(this)

v.VisitConcreteElementA(this)

v.VisitConcreteElementB(this)
Visitor: Participants

- **Visitor**
  - declares a `visit()` operation for each class of `ConcreteElement` in the object structure

- **ConcreteVisitor**
  - implements each operation declared by **Visitor**

- **Element**
  - defines an operation `accept(Visitor)`

- **ConcreteElement**
  - implements operation `accept(Visitor)`
Visitor: Sequence Diagram

```
: client

: c1 : ConcreteElementA

: c2 : ConcreteElementB

: ConcreteVisitor

accept(aVisitor)  visitA(c1)

operationA()

accept(aVisitor)

visitB(c2)

operationB()
```
Visitor: Intent and context

• represent an operation to be performed on a set of “related classes” without changing the classes.

• apply *Visitor* when:
  • a hierarchy contains many classes 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, and you want to avoid polluting their classes with these operations.
  • the classes in the object structure rarely change, but you frequently want to add new operations on the structure.
Visitor: Example

- a final variation on the *FileSystem* example
- goal: implement the Unix “du” (disk usage) command using a *Visitor* (Summarize disk usage size of a directory and its subdirectories)

- create interface *Visitor* with methods `visit(File)`, `visit(Directory)`, `visit(Link)`
  - create class `DuVisitor` that implements *Visitor*
  - declare `accept(Visitor)` method in class *Node*, implement in *File*, *Directory*, *Link*
Interface Visitor

```
interface Visitor {
    public void visit(File f);
    public void visit(Directory d);
    public void visit(Link l);
}
```
Class DuVisitor (1)

class DuVisitor implements Visitor {
    DuVisitor()
    {
        _nrFiles = 0;
        _nrDirectories = 0;
        _nrLinks = 0;
        _totalSize = 0;
    }

    // visit a file
    public void visit(File f)
    {
        _nrFiles++;
        _totalSize += f.size();
    }

    ...
}
Class DuVisitor (2)

...// when visiting a directory, visit all its children

public void visit(Directory d) {
    _nrDirectories++;
    Iterator it = d.iterator();
    for (it.first(); !it.isDone(); it.next()) {
        Node n = (Node) it.current();
        if (n instanceof File) {
            visit((File) n);
        } else if (n instanceof Directory) {
            visit((Directory) n);
        } else if (n instanceof Link) {
            visit((Link) n);
        }
    }
    n.accept(this);
}
Class DuVisitor (3)

...  

// Does not follow links. Some work would be involved to
// avoid counting the same file twice.

public void visit(Link l) {
    _nrLinks++;
}

public void report() {
    System.out.println("number of files: " + _nrFiles);
    System.out.println("number of directories: " + _nrDirectories);
    System.out.println("number of links: " + _nrLinks);
    System.out.println("total size of files: " + _totalSize);
}

int _totalSize; int _nrFiles; int _nrLinks; int _nrDirectories;
Adding accept methods

class File extends Node {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
    ...
}

class Directory extends Node {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
    ...
}

class Link extends Node {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
    ...
}
Client code

```java
public class Main {
    public static void main(String[] args) {
        Directory root = new Directory(""");
        File core = new File("core", root, "hello");
        Directory usr = new Directory("usr", root);
        File adm = new File("adm", usr, "there");
        Directory foo = new Directory("foo", usr);
        File bar1 = new File("bar1", usr, "abcdef");
        File bar2 = new File("xbar2", usr, "abcdef");
        File bar3 = new File("yybarzz3", usr, "abcdef");
        Link link = new Link("link-to-usr", usr, root);
        Link linkToLink = new Link("link-to-link", link, root);

        DuVisitor visitor = new DuVisitor();
        root.accept(visitor);
        visitor.report();
    }
}
```
Output

number of files: 5
number of directories: 3
number of links: 2
total size of files: 28
Visitor: Considerations

- adding new operations is easy
- a *visitor* gathers related operations and separates unrelated ones
- adding new *ConcreteElement* classes is hard
  - requires new abstract operation on *Visitor*
  - requires implementation in every *ConcreteVisitor*
- *Visitor* not limited to classes in a (sub)hierarchy, can be applied to any collection of classes
  - provided they define *accept()* methods
- *Visitor* requires that *ConcreteElement* classes expose enough state so *Visitor* can do its job
  - breaks encapsulation
Template Method

- Basic structure
  - Add front stairs
  - Add flags
  - Add front stairs, flags and tower
Example
The Application class is responsible for opening existing documents stored in an external format, such as a file.

A Document object represents the information in a document once it's read from the file.

Applications built with the framework can subclass Application and Document to suit specific needs.

We want to specify the order of operations that a method uses, but allow subclasses to provide their own implementations of some of these operations.
Template Method

OpenDocument = template method:

```java
public void OpenDocument (String name) {
if (!CanOpenDocument(name)) {
    return;
}
Document doc = DoCreateDocument();
if (doc != null) {
    docs.AddDocument(doc);
    AboutToOpenDocument(doc);
    doc.Open();
    doc.DoRead();
}
}
```

The template method fixes the order of operations, but allows Application subclasses to vary those steps as needed.
Template Method: Class Diagram
Template Method: Intent and context

- 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.

Use the Template Method pattern:

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- to localize common behavior among subclasses and place it in a common class (in this case, a superclass) to avoid code duplication.
- to control how subclasses extend superclass operations. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.
Mediator: Motivation
Mediator: Solution
Mediator

- FontDialogDirector abstraction
Mediator: Structure
Mediator: Intent and Context

- Define an object that **encapsulates how a set of objects interact**. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Use the Mediator pattern when

- a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- reusing an object is difficult because it refers to and communicates with many other objects.
- a behavior that's distributed between several classes should be customizable without a lot of subclassing.
Design Patterns
Design patterns in practice

- examples where design patterns are used in the Java libraries
  - **Iterator**
    - java.util.Iterator
    - java.utilEnumeration
    - collection classes such as java.util.HashSet have iterator() methods
  - **Observer**
    - java.util.Observable
    - various kinds of Listeners
Principles of Behavioral patterns

- **Encapsulating variation:** encapsulate a frequently changing aspect (*Strategy, State, Mediator, Iterator*).
- **Objects as arguments:** *Visitor, Command, Memento*.
- **Communication:** Distributed in *Observer*; encapsulated in *Mediator*.
- **Decouple senders and receivers:** *Command, Observer, Mediator, and Chain of Responsibility*.
Design Patterns principles (Erich Gamma)

- Two principles of reusable object-oriented design (GOF).
  - Program to an interface, not an implementation
  - Favor object composition over class inheritance

- Program to an interface, not an implementation
  - This principle addresses dependency relationships
  - Dependency relationships have to be carefully managed in a large app.
  - It's easy to add a dependency on a class.
  - Getting rid of an unwanted dependency can block you from reusing the code in another context.
Design Patterns principles
(Erich Gamma)

- The value of interfaces
  - Once you depend on interfaces only, you're decoupled from the implementation.
  - The implementation can vary.
  - Separates the design, from the implementation, which allows clients to be decoupled from the implementation.
  - Use of Java interfaces?
    - An abstract class is good as well. In fact, an abstract class gives you more flexibility when it comes to evolution. You can add new behavior without breaking clients.
  - An interface refines the collaboration between objects.
  - An interface is free from implementation details, and it defines the vocabulary of the collaboration
Design Patterns principles (Erich Gamma)

- **Favor object composition over class inheritance**
  - Inheritance: There's a **tight coupling** between the **base class** and the **subclass**, because of the implicit context in which the subclass code plug in will be called.
  - Composition: the **coupling is reduced** by just having some smaller things you plug into something bigger, and the bigger object just calls the smaller object back.
  - From an API point of view defining that a method can be overridden is a stronger commitment than defining that a method can be called
Design Patterns principles

- Dynamic Binding (late-binding, run-time binding)
  - Issuing a request doesn't commit you to a particular implementation until run-time.
  - You can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request.
  - lets you substitute objects that have identical interfaces for each other at run-time.
    - **Polymorphism** - a key concept in object-oriented systems.
      - lets a client object make few assumptions about other objects beyond supporting a particular interface.
      - Polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time.
Design Patterns: Final Words

• **beware of pattern hype**: design patterns are not the solution to all problems!

• in general, **don’t try to apply as many patterns as possible**. Instead, try to:
  • recognize situations where patterns are useful
  • use key patterns to define global system architecture

• **document** your use of patterns, use names that reflect participants in patterns

• in practice, reusable software often has to be **refactored**
  • design patterns are often the “target” of refactorings that aim at making the system more reusable
Recommended Papers

Cheng Zhang and David Budgen, What Do We Know about the Effectiveness of Software Design Patterns?, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,

• Conclusions:
  • “We could not identify firm support for any of the claims made for patterns in general, although there was some support for the usefulness of patterns in providing a framework for maintenance, and some qualitative indication that they do not help novices learn about design.
  • For future studies we recommend that researchers use case studies that focus upon some key patterns, and seek to identify the impact that their use can have upon maintenance”


Our Catalogue: http://www.cs.bgu.ac.il/~cd-patterns/
Appendix
Command: Motivation

- A user interface toolkit include objects like buttons and menus that carry out a request in response to user input.
- The operation triggered by a request is application dependent: Performed by Domain Layer objects.
- The toolkit does not know the receiver of the request or the operations that will carry it out.
Command: Solution

- make requests of unspecified application objects by turning the request itself into an object.

- Key abstraction: an abstract \textit{Command} class.
  - Declares an interface for executing operations.
  - The \textit{Command} interface includes an abstract \textit{execute} operation.

- \textit{Concrete Command} subclasses specify a receiver-action pair by storing the receiver, and by implementing \textit{execute}.
  - The receiver has the knowledge required to carry out the request.
Command: Solution

![Diagram of the Command pattern solution]

- **Application**
  - Add(Document)

- **Menu**
  - Add(ButtonItem)

- **ButtonItem**
  - Clicked()

- **Command**
  - Execute()

- **Document**
  - Open()
  - Close()
  - Cut()
  - Copy()
  - Paste()

- **Application**
  - Add(Document)

- **OpenCommand**
  - Execute()
  - AskUser()

```java
name = AskUser()
doc = new Document(name)
analysis->Add(doc)
doc->Open()
```
Command: Solution

**Diagram:**
- **Command** class with `Execute()` method.
- **MacroCommand** class with `Execute()` method and method for iterating over commands.
- **Document** class with `Open()`, `Close()`, `Cut()`, `Copy()`, and `Paste()` methods.
- **PasteCommand** class with `Execute()` method.
- Arrows and relationships between classes indicating aggregation and association.
Command: Participants

- **Command**
  - declares an interface for executing an operation.

- **ConcreteCommand** (PasteCommand, OpenCommand)
  - defines a binding between a *Receiver* object and an action.
  - implements *execute* by invoking the corresponding operation(s) on *Receiver*.

- **Client** (Application)
  - creates a *ConcreteCommand* object and sets its receiver.

- **Invoker** (MenuItem)
  - asks the command to carry out the request.

- **Receiver** (Document, Application)
  - knows how to perform the operations associated with carrying out a request.
  - Any class may serve as a *Receiver*. 
Command: Class diagram

- When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
Command: Sequence diagram
Command: Intent and context

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

- Apply **Command** for the following purposes:
  - parameterize objects by an action to perform.
  - specify, queue, and execute requests at different times.
    - A **Command** object can have a lifetime independent of the original request.
  - support undo and logging changes.
    - The *execute* operation can store state for reversing its effects.
    - can keep a persistent log of changes.
  - Support transactions:
    - structure a system around high-level operations built on primitive operations.
Command: Consequences

• Decouples the object that invokes the operation from the one that knows how to perform it.

• *Commands* are first-class objects. They can be manipulated and extended like any other object.

• *Commands* can be assemble into a composite command.
  • *Composite commands* are an instance of the *Composite* pattern.

• Easy to add new *Commands*: no need to change existing classes.
Other Behavioral Patterns

- Chain of Responsibility
  - avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request

- Interpreter
  - given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

- Memento
  - without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.
AntiPatterns

- *AntiPatterns* are *negative* “solutions” that present more problems than they address.

- We know:
  - a third of software projects are canceled.
  - The remaining projects delivered software that was typically twice the expected budget and took twice as long to developed as originally planned [Johnson 95].

- *AntiPatterns* are a natural extension to *design patterns*:
  - focused on repeated software failures in an attempt to understand, prevent, and recover from them.
AntiPatterns Research

“The study of AntiPatterns is an important research activity. The presence of ‘good’ patterns in a successful system is not enough; you also must show that those patterns are absent in unsuccessful systems. Likewise, it is useful to show the presence of certain patterns (AntiPatterns) in unsuccessful systems, and their absence in successful systems.”

—Jim Coplien
How to Kill a Software Project

- Show the same demo twice to the same audience.
- Focus on the technologies, not the problems and scenarios.
- Fail to maximize return on investments; for example, developing proof-of-concept prototypes is more effective than adding additional content to an existing prototype.
- Change project focus from the larger scale to the smaller scale.
- Don’t maintain consistency between releases.
- Isolate team efforts from other groups within an organization.
- Rewrite existing clients, servers, and applications.
- Change the purpose of the system, so that the models describe the wrong focus and objects.
AntiPatterns – key concepts:

- *Root causes:* The fundamental context.

- *Primal forces:* Motivators for decision making.

- *Software design-level model (SDLM):* Software level.
Root Causes

- Mistakes in software development that result in failed projects, cost overruns, schedule slips, and unfulfilled business needs [Mowbray 97].

- based upon the “seven deadly sins” [Bates 96]:
  - Haste.
  - Apathy.
  - Narrow-mindedness.
  - Sloth.
  - Avarice.
  - Ignorance (intellectual sloth).
  - Pride.
Primal forces

- *Management of functionality*: meeting the requirements.
- *Management of performance*: meeting required speed of operation.
- *Management of change*: controlling evolution of software.
- *Management of IT resources*: controlling use and implementation of people and IT artifacts.
- *Management of technology transfer*: controlling technology change.
Design patterns and AntiPatterns

Design Patterns

Problem + Solution Pairs

Problem

Context & Forces

Solution

Benefits

Consequences

Related Patterns

AntiPatterns

Solution + Solution Pairs

Contextual Causes

AntiPattern Solution

Symptoms & Consequences

Refactored Solution

Benefits

Consequences

Related Patterns & AntiPatterns

MITRE
Software development AntiPatterns

- A key goal of development *AntiPatterns* is to describe useful forms of *software refactoring*.

- *Software refactoring* is a form of code modification, used to improve the software structure in support of subsequent extension and long-term maintenance.

- In most cases, the goal is to transform code without impacting behavior correctness.
Example: The *Blob* AntiPattern

- **General Form**
  - The Blob is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data.
  - This AntiPattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes.
  - Key problem: Major of the responsibilities are allocated to a single class.
Example: The *Blob* AntiPattern

- **Refactored solution**
  - Identify or categorize related attributes and operations according to contracts.
  - Look for “natural homes” for these contract-based collections of functionality and then migrate them there.
  - Remove all “far-coupled,” or redundant, indirect associations.
  - Migrate associates to derived classes to a common base class.
  - Remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments.
Example: The *Spaghetti Code* AntiPattern

**General Form**

- *Spaghetti Code* appears as a program or system that contains very little software structure.
- Coding and progressive extensions compromise the software structure to such an extent that the structure lacks clarity.
- **If developed using an object-oriented language:**
  - the software may include a small number of objects that contain methods with very large implementations that invoke a single, multistage process flow.
  - The object methods are invoked in a predictable manner, and there is a negligible degree of dynamic interaction between them.
  - The system is very difficult to maintain and extend, and there is no opportunity to reuse the objects and modules in other similar systems.
Example: The *Spaghetti Code* AntiPattern

- **Refactored solution**
  
  1. Use accessor functions.

  2. Convert a code segment into a function that can be reused in future maintenance and refactoring efforts. Resist implementing the *Cut-and-Paste AntiPattern*.

  3. Reorder function arguments to achieve greater consistency throughout the code base. Consistent bad *Spaghetti Code* is easier to maintain than inconsistent *Spaghetti Code*.

  4. Remove portions of the code that may become, or are already, inaccessible. Failure to remove obsolete portions of code causes the *Lava Flow AntiPattern*.

  5. Rename classes, functions, or data types to conform to an enterprise or industry standard and/or maintainable entity.
Example: The *Spaghetti Code* AntiPattern

**Development AntiPattern:**

**Spaghetti Code - Refactored Solution**

- **Refactor to generalize:** *Create an abstract superclass*
  1. Make subclass function signatures compatible
  2. Add function signatures to the superclass
  3. Make function bodies and variables compatible
  4. Migrate common code to the superclass

- **Refactor to specialize:** *Simplify conditionals*
  1. For each condition, create a subclass with matching invariant
  2. Copy the code into the subclass
  3. Simplify code based upon invariant
  4. Specialize the superclass constructor

- **Refactor to combine:** *Capture aggregations and components*
  Type A. Move members from an aggregate class to a components class
  Type B. Move members from component classes to aggregate class
  Type C. Convert inheritance into an aggregation

*Source: [Opdyke 92]*
Patterns:

Applications of Patterns
Patterns have a myriad of uses - many yet to be discovered.

- **Analysis Patterns**
  - Create robust analysis models

- **Design Patterns**
  - Design and write new software
  - Analyze and document software

- **AntiPatterns**
  - Refactor designs and software
  - Migrate/upgrade systems
  - Reengineer systems and organization

MITRE
AntiPatterns sources

- [http://www.antipatterns.com/briefing](http://www.antipatterns.com/briefing):
  - AntiPatterns, a Brief Tutorial
- [http://c2.com/cgi/wiki?AntiPatternsCatalog](http://c2.com/cgi/wiki?AntiPatternsCatalog)
  - Architecture AntiPatterns
  - Development AntiPatterns
  - Project Management AntiPatterns