

Design Patterns 2

6.170 Lecture 18

Spring 2004

Last week we looked at creational and structural design patterns. It is easy enough for a single client to use a single abstraction. However, occasionally a client may need to use multiple abstractions; furthermore, the client may not know ahead of time how many or even which abstractions will be used. The observer, visitor, blackboard, and other *behavioral* patterns permit such communication.

1 Observer

Suppose that there is a database of all MIT student grades, and the 6.170 staff wishes to view the grades of 6.170 students. They could write a `SpreadsheetView` class that displays information from the database. (We will assume that the viewer caches information about 6.170 students—it needs this information in order to redraw, for example—but whether it does so is not an important part of this discussion.) The display might look something like this:

	PS1	PS2	PS3
G. Bates	45	85	80
A. Hacker	95	90	85
A. Turing	90	100	95

Suppose the code to communicate between the grade database and the view of the database uses the following interface:

```
interface GradeDBViewer {  
    void update(String course, String name, String assignment, int grade);  
}
```

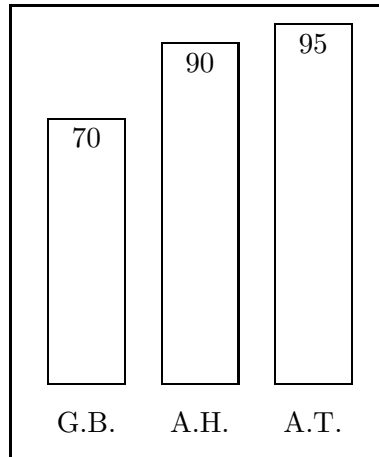
When new grade information is available (say, a new assignment is graded and entered, or an assignment is regraded and the old grade corrected), the grade database must communicate that information to the view. Let's suppose that Gill Bates has demanded a regrade on problem set 1, and that regrade did reveal grading errors: Gill's score should have been 30. The database code must somewhere make calls to `SpreadsheetView.update`. Suppose that it does so in the following way:

```
SpreadsheetView ssv = new SpreadsheetView();  
...  
ssv.update("6.170", "G. Bates", "PS1", 30);
```

(For brevity, this code shows literal values rather than variables for the `update` arguments.)
Then the spreadsheet view would redisplay itself in the following way:

	PS1	PS2	PS3
G. Bates	30	85	80
A. Hacker	95	90	85
A. Turing	90	100	95

The staff might later decide that they would like to also view grade averages as a bargraph, and implement such a viewer:



Maintaining such a view in addition to the spreadsheet view requires modifying the database code:

```
SpreadsheetView ssv = new SpreadsheetView();
BargraphView bgv = new BargraphView();
...
ssv.update("6.170", "G. Bates", "PS1", 30);
bgv.update("6.170", "G. Bates", "PS1", 30);
```

Likewise, adding a pie chart view, or removing some view, would require yet more modifications to the database code. Object-oriented programming (not to mention good programming practice) is supposed to provide relief from such hard-coded modifications: code should be reusable without editing and recompiling either the client or the implementation.

The observer pattern achieves the goal in this case. Rather than hard-coding which views to update, the database can maintain a list of observers which should be notified when its state changes.

```
Vector observers = new Vector();
...
for (int i=0; i<observers.size(); i++) {
    GradeDBViewer v = (GradeDBViewer) observers[i];
    v.update("6.170", "G. Bates", "PS1", 30);
}
```

In order to initialize the vector of observers, the database will provide two additional methods, `register` to add an observer and `remove` to remove an observer.

```

void register(GradeDBViewer observer) {
    observers.add(observer);
}

boolean remove(GradeDBViewer observer) {
    return observers.remove(observer);
}

```

The observer pattern permits client code (which manages the database and the viewers) to select which observers are active, and observers can even be added and removed at run-time.

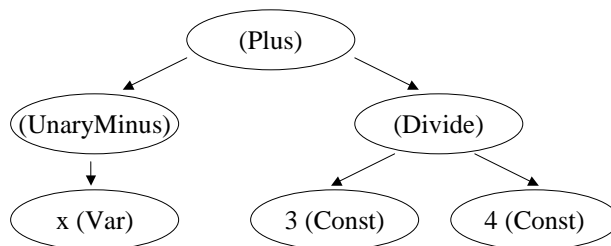
This discussion has glossed over a number of details. For instance, the client might store all the information of interest to it (which might be all the 6.170 grades, or just the grades for some students, or just the number of updates to the database for a `DatabaseActivityViewer`), duplicating parts of the database, or the client might read the database when needed. A related design decision is whether the database sends all potentially relevant information to the client when an update occurs (this is the *push* structure), or the database simply informs the client, “an update has occurred” (this is the *pull* structure). The pull structure forces the client to request information, which may result in more messages, but overall a smaller amount of data transferred.

2 Traversing hierarchical structures

Iterators are a familiar design pattern that facilitate traversal through a linear collection. What about hierarchical collections? Object oriented software systems are full of hierarchies:

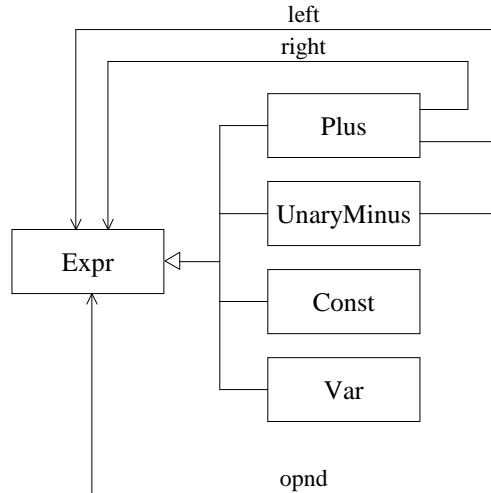
- ▷ a user interface is a hierarchical collection of UI objects: windows, panes, buttons, textboxes;
- ▷ a filesystem is a hierarchy of directories and files;
- ▷ an email client contains a hierarchy of mail accounts, folders, messages, and attachments.

Before we look at how to traverse these kinds of hierarchies, we should first consider how the hierarchy might be represented. Recall that the composite pattern permits a client to manipulate either an atomic unit or a collection of units in exactly the same way. The client need not create special code for the case of a higher-level object with structure as opposed to a basic object. The same operations work on both, because both extend the same type. Consider the problem of representing expressions in a programming language. Syntactically, the expression $-x + 3/4$ can be represented by a hierarchy of objects called an abstract syntax tree:



The nodes at the bottom of the tree, instances of `Const` and `Var`, are the atomic units, or leaves. The internal nodes of the tree, corresponding to operators like `Plus` and `Divide`, are composites.

The key idea of the composite pattern is that composites and leaves all share a common interface, which in this case we'll call `Expr`. Here's part of the object model showing the relevant classes:



The shape of this object model tells you it's a composite pattern: a base class (**Expr**) with several subclasses, some of which are terminal (the leaves **Const** and **Var**), and some of which have references to subexpressions (the composites **Plus** and **UnaryMinus**). The code for these classes might look like this, if we only pay attention to the fields and ignore the methods:

```

interface Expr {
    ...
}
class Plus implements Expr {
    Expr left;
    Expr right;
    ...
}
class UnaryMinus implements Expr {
    Expr opnd;
    ...
}
class Const implements Expr {
    double value;
    ...
}
class Var implements Expr {
    String name;
    ...
}
  
```

A complete representation would need other classes as well, to represent multiply, divide, subtract, etc.

2.1 Interpreter

Now we're ready to talk about patterns for traversing composite structures, using **Expr** as an example. There are many operations we may want to perform on an expression. Suppose we want to evaluate the expression, given some execution context that assigns values to variables. Evaluation might be provided as a method of **Expr**:

```

interface Expr {
    double eval (Context context);
    ...
}

```

Each type of expression then implements `eval` in the appropriate manner. In particular, composites like `Plus` have to recursively evaluate their subexpressions:

```

class Plus implements Expr {
    Expr left;
    Expr right;
    double eval (Context context) {
        return left.eval (context) + right.eval (context);
    }
    ...
}

```

```

class Const implements Expr {
    double value;
    double eval (Context context) {
        return value;
    }
    ...
}

```

```

class Var implements Expr {
    String name;
    double eval (Context context) {
        return context.getVarValue (name);
    }
    ...
}

```

This technique is called the interpreter pattern. An operation that needs to traverse a composite hierarchy is declared as a method on the composite base class, in this case `Expr`. Composite nodes recursively call the operation on their parts, and the leaves terminate the recursion. In the interpreter pattern, both the operation and the means of traversal are bound up in the composite classes themselves.

2.2 Procedural Traversal

A number of operations might be reasonably implemented as interpreter methods on `Expr`: pretty-printing, type checking, and optimization come to mind. But a problem arises when a client wants to define a new operation. For example, suppose I want to know all the variable names that are used by an expression, perhaps so I can introduce a new variable and avoid conflicts with existing names. With the interpreter pattern, I would have to add a new method to `Expr` for my new operation. Then I'd have to implement the new method in all the subclasses of the composite. I'd

need to change many classes to implement just one new operation. Furthermore, I'd need access to the source code of the expression classes, which isn't always possible. A first step towards fixing this problem is to represent the new operation as a class itself. (This is what the Liskov text calls the procedural approach to hierarchy traversal, although here we're using a class rather than a collection of static procedures.) The class has a separate method for each type of expression. Methods corresponding to composites take care of traversing the composite's children. Here's the code:

```
class FindVariables {
    Set vars = new HashSet ();
    void forExpr (Expr e) {
        ... // see below for this method's body
    }
    void forPlus (Plus e) {
        forExpr (e.left ());
        forExpr (e.right ());
    }
    void forConst (Const e) { }
    void forVar (Var e) {
        vars.add (e.name ());
    }
    ...
}
```

This class traverses the expression tree, and every time a `Var` node is encountered, its variable name is added to a set. For convenience of other clients, it's probably best to hide this operation class behind a simple method interface that instantiates it and invokes it correctly:

```
static Set variablesUsed (Expr e) {
    FindVariables op = new FindVariables ();
    op.forExpr (e);
    return op.vars;
}
```

This works well enough – at least we didn't need to change the expression classes – but it has one ugly aspect that was omitted from the code above. The method for a composite like `forPlus` doesn't know what method it should call for its subexpressions, since they could be any type of node. So it calls `forExpr`, which tests for and dispatches on all the possible expression types:

```
void forExpr (Expr e) {
    if (e instanceof Plus) forPlus ((Plus) e);
    else if (e instanceof Const) forConst ((Const) e);
    else if (e instanceof Var) forVar ((Var) e);
    ...
    else assert (false); // don't know e's type
}
```

Maintaining this code is tedious and error-prone. The long list of cascaded if tests are likely to run slowly. Furthermore, even though this code is bad enough appearing once, in fact it must

occur repeatedly in every operation class we define. Systematic repetition in code is usually a sign of a need to redesign, possibly using a pattern.

We already know a Java construct that automatically chooses code to execute based on the type of an object: method dispatch. Method dispatch does the same kind of comparison and selection as the cascaded if tests, but does not clutter the code, is likely to be more efficient, and gives us compile-time checking if we forget a case. The visitor pattern takes advantage of this.

2.3 Visitor

The visitor pattern encodes a traversal over a composite hierarchy. As in the procedural approach, an operation is represented by a class, called a visitor, with a method for each type of node in the composite. However, instead of putting type-testing code in the visitor to decide which method needs to be called, that responsibility is shifted to the composite classes. Furthermore, the composite classes will also assume responsibility for the traversal. Here's the basic scheme:

1. A composite node is asked to accept a visitor by a call to its accept method:

```
composite.accept (visitor)
```

2. The composite recursively passes the visitor on to its parts:

```
for each part in composite,  
    part.accept (visitor)
```

3. Depending on its type, the composite calls the appropriate method of the visitor, passing itself as an argument.

```
visitor.forNodeType (composite)
```

The accept and visit methods work together in such a way that `composite.accept(visitor)` performs a depth-first traversal of the structure rooted at `composite`. In this case, the traversal is called post-order because a node is visited after all its children have been visited. Other possibilities are pre-order, which visits the node first, and in-order, which visits the node between its two children and makes sense only for binary trees.

For our expression example, the code looks like this:

```
interface Visitor {  
    void forPlus (Plus e);  
    void forConst (Const e);  
    void forVar (Var e);  
    ...  
}  
interface Expr {  
    void accept (Visitor v);  
    ...  
}  
class Plus implements Expr {  
    Expr left;  
    Expr right;  
    void accept (Visitor v) {
```

```

        left.accept (v);
        right.accept (v);
        v.forPlus (this);
    }
    ...
}
class Const implements Expr {
    String name;
    void accept (Visitor v) {
        v.forConst (this);
    }
    ...
}
class Var implements Expr {
    String name;
    void accept (Visitor v) {
        v.forVar (this);
    }
    ...
}

```

Using the visitor pattern, the `FindVariables` operation could be implemented much more simply. There is no need for `instanceof` tests or downcasts, or even any traversal code. In fact, only the `forVar` method has a nontrivial body:

```

class FindVariables implements Visitor {
    Set vars = new HashSet ();
    void forPlus (Plus e) { }
    void forConst (Const e) { }
    void forVar (Var e) { vars.add (e.name ()); }
    ...
}

```

However, the method that uses `FindVariables` must reverse the way it invokes the operation. Instead of passing the expression to the operation, it must pass the operation to the expression:

```

static Set variablesUsed (Expr e) {
    FindVariables op = new FindVariables ();
    e.accept (op);
    return op.vars;
}

```

This validates the decision we made to hide `FindVariables` behind a method interface. Otherwise, changing it to a visitor would have entailed changing every place in the code where it was invoked.

A visitor is very much like an iterator. Each element of the hierarchy is presented to the visitor in turn, with the additional benefit (not provided by the iterator pattern) of a dispatch based on the type of the element. Further more, a visitor can accumulate state over the course of its traversal, e.g., the set of variable names encountered. Unfortunately, the visitor pattern shown

above does not give the visitor any control over the traversal. What if we wanted to implement `eval` as a visitor, rather than an interpreter? This would be hard as it stands. To implement `Plus`, for example, the visitor would need the results of evaluating its two subexpressions, but the design above only makes it easy to receive the result of the last node that was visited.

The Liskov text proposes one solution to this problem: saving results on a stack and popping them off at the appropriate times. This keeps the visitors and acceptors clean, but it can be hard to see how data flows between visitor calls.

Another solution is to move responsibility for the traversal back into the visitor. We'll call it a `SelfGuidedVisitor` to distinguish it from the more passive kind of visitor, but its methods are superficially the same:

```
interface SelfGuidedVisitor {
    void forPlus (Plus e);
    void forConst (Const e);
    void forVar (Var e);
    ...
}
```

When given a self-guided visitor, the `accept` methods do nothing but type-dispatching:

```
interface Expr {
    void accept (SelfGuidedVisitor v);
    ...
}
class Plus implements Expr {
    Expr left;
    Expr right;
    void accept (SelfGuidedVisitor v) {
        v.forPlus (this);
    }
    ...
}
class Var implements Expr {
    String name;
    void accept (SelfGuidedVisitor v) {
        v.forVar (this);
    }
    ...
}
```

The self-guided visitor must take over its own traversal, by asking the children of each visited node to accept it. Because it controls when children are visited, the visitor can also keep track of the results returned by those children:

```
class Evaluator implements SelfGuidedVisitor {
    Context context; // maps variables > values
    double result; // value of last subexpr evaluated
    void forPlus (Plus e) {
        // evaluate left subexpression
```

```

    e.left ().accept (this);
    // hang onto its result
    double left = result;
    // evaluate right subexpression
    e.right ().accept (this);
    double right = result;
    // combine the two results
    result = left + right;
}
void forConst (Const e) {
    result = e.value ();
}
void forVar (Var e) {
    result = context.getVarValue (e.name ());
}
...
}

```

Both variants of the visitor pattern are useful members of your toolbox, and a well designed composite hierarchy may need to support both kinds.

3 Blackboard

The blackboard pattern generalizes the observer pattern to permit multiple data sources as well as multiple viewers. It also has the effect of completely decoupling producers and consumers of information. A blackboard is a repository of messages which is readable and writable by all processes. Whenever an event occurs that might be of interest to another party, the process responsible for or knowledgeable about the event adds to the blackboard an announcement of the event. Other processes can read the blackboard. In the typical case, they will ignore most of its contents, which do not concern them, but they may take action on other events. A process which posts an announcement to the blackboard has no idea whether zero, one, or many other processes are paying attention to its announcements.

Blackboards generally do not enforce a particular structure on their announcements, but a well-understood message format is required so that processes can interoperate. Some blackboards provide filtering services so that clients do not see all announcements, just those of a particular type; other blackboards automatically send announcements to clients which have registered interest (this is a pull structure). An ordinary bulletin board (either the physical or the electronic kind) is an example of a blackboard system. Another example of a blackboard at MIT is the zephyr messaging service.

The Liskov text calls this pattern “white board” rather than “blackboard.” The former name may be more modern-seeming, but the latter is standard computer science terminology which has been in use for decades and will be more quickly recognized outside 6.170. The first major blackboard system was the Hearsay-II speech recognition system, implemented between 1971 and 1976.

4 Creational patterns revisited

Now that we've seen samples from the three broad classes of patterns – creational, structural, and behavioral – let's return to some creational patterns that are more subtle than the ones we looked at in the previous lecture, and are important to know about.

4.1 Interning

The interning design pattern reuses existing objects rather than creating new ones. If a client requests an object that is equal to one that already exists, then the pre-existing one is returned instead. Interning is generally useful only for immutable objects. For example, an image might be represented as an array of pixels, each of which is a `Color`. A large image may have many pixels in it, but only a few distinct `Colors`. Representing every pixel with a unique `Color` object is unnecessarily wasteful of space. A better way would be to share the `Colors`.

Interning arranges for objects that are immutable to be reused. Interning requires a table of all the objects (of the given type) that have ever been created. If the table contains an object that is equivalent to the desired object, the table's object (the interned object) is returned instead. Otherwise, the desired object is created and stored in the table for future reference.

Here is a method that returns an interned `Color`, given its red, green, and blue values:

```
static Map map = new HashMap ();
public static Color fromRGB (float r, float g, float b) {
    Color color = new Color (r, g, b);
    if (map.containsKey (color)) {
        return (Color) map.get (color);
    } else {
        map.put (color, color);
        return color;
    }
}
```

The interning pattern necessarily uses a static method. We wouldn't be able to return an interned object from a Java constructor, because a constructor always returns a freshly minted object. In this case, the table maps each interned `Color` to itself, so the method creates a trial `Color` object before checking whether the color has already been interned. When the interned objects are expensive to create, however, it may make more sense for the interning table to map the content of the object (in this case, its RGB value) to the interned object, in order to save the cost of construction when the object already exists. Note that this code uses a `Map` rather than a `Set`, even though all we're really doing is storing a set of interned `Colors`. The reason is that sets do not have a `get` operation, only `contains`. In other words, we'd be able to test whether the set of interned objects contains the color we want, but we'd be unable to actually obtain a reference to the interned object. Interning for strings is so important that it is built into Java. `String.intern` returns an interned version of a string.

The Liskov text discusses interning in section 15.2.1, but calls the pattern 'flyweight,' which is different from current terminology in the field.

4.2 Flyweight

The flyweight pattern is a generalization of interning. Interning is applicable only when an object is completely immutable and all of its state can be shared by many instances of the object. The

more general flyweight pattern can be used when most (but not necessarily all) of an object is immutable, or when instances of the object share most but not all of their state. Suppose you are modelling a tree, with a trunk, branches, twigs, and leaves. We'll focus on the twigs and the leaves:

```
class Twig {
    Leaf[] leaves;
    ...
}
class Leaf {
    Color color;
    Shape shape;
    Twig parent;
    int distance; // along twig
    int angle; // around twig
}
```

The leaves in a tree are mostly identical. They differ only in their context, i.e., the twig to which the leaf is attached, and where it grows out of that twig. Properties inherent to the leaf make up its intrinsic state, in this case color and shape. Properties of the leaf's context are its extrinsic state; here, its parent, distance, and angle fields. Although leaves seem like natural candidates for the interning pattern, because there are many of them and they are intrinsically identical, the presence of the extrinsic state in `Leaf` blocks us from using interning, because every `Leaf` object must have a different location. The flyweight pattern solves this problem by removing the extrinsic state from the leaf object itself and relying on the leaf's context to supply it when needed. We'll define a new class `Leaflet` to represent just the intrinsic state of a leaf:

```
class Leaflet {
    Color color;
    Shape shape;
}
```

We can readily apply interning to `Leaflet`; all the leaves of one tree, even a whole forest of a single species, can be represented by a single object. A richer model of a tree might have different `Leaflets` in different stages of growth or under attack by caterpillars or disease, but this can be easily accommodated while still obtaining substantial sharing from interning. When a `Leaflet` needs to access its extrinsic state, for instance to draw itself on the screen, the state must be passed in. For example, the twig might tell its leaves to draw themselves by computing their locations on the fly:

```
class Twig {
    Leaflet[] leaves;
    void draw () {
        for (int i = 0; i < leaves.length; ++i) {
            int distance = i * distanceIncrement;
            int angle = (i * angleIncrement) % 360;
            leaves[i].draw (this, distance, angle);
        }
    }
}
```

```

class Leaflet {
    Color color;
    Shape shape;
    void draw (Twig parent, int distance, int angle) {
        ...
    }
}

```

The main disadvantage of the flyweight pattern is that a reference to a `Leaflet` is no longer sufficient to identify a specific leaf and tell you everything you need to know about it. You have to know its context as well, namely the twig and the leaf's index on that twig, in order to access its extrinsic state. Often, however, flyweight can be used “under the covers” to improve the performance of a system, while from a client's point of view objects are still heavyweight. For example, if a client requests a reference to a specific leaf on a twig, `Twig` might return a full-fledged (but temporary) `Leaf` that wraps both the `Leaflet` and its context.

Flyweight should only be considered after profiling has determined that space usage is a critical bottleneck in the program. Introducing such constructs into programs complicates them and presents many opportunities for error. It should be undertaken only in limited circumstances.

4.3 Prototype pattern

In the prototype pattern, new objects are created by punching out copies of the prototype, using either `clone()` or a class-specific copy method. The latter technique is used below:

```

public class Race {
    public Race (Bicycle prototype) {
        bike1 = prototype.copy ();
        bike2 = prototype.copy ();
        ...
    }
}

public class Bicycle {
    public Bicycle copy () { ... }
    ...
}

```

Effectively, each object is itself a factory specialized to making objects like itself. Prototypes are particularly useful for avoiding subclassing, because you can create new “classes” at runtime by configuring different prototypes:

```

Bicycle prototype1 = new RacingBicycle ();
prototype1.replaceWheels (new FancyWheel ());
Bicycle prototype2 = new MountainBicycle ();
prototype2.repaint (Color.GREEN);

```

This customized objects can be used to create many others like themselves, which can be more convenient than creating subclasses for all the important permutations of features.

There is no free lunch: the code to create objects of particular classes must go somewhere. Factory methods put the code in methods in the client; factory objects put the code in methods in a factory object; and prototypes put the code in `copy/clone` methods.

design patterns

Creational patterns

Minimize coupling introduced by object creation

Structural patterns

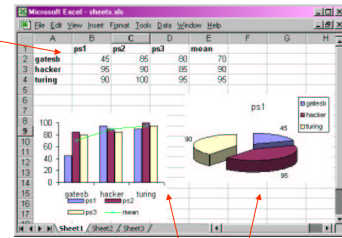
Minimize coupling introduced when classes or objects are assembled to form larger structures

Behavioral patterns

Minimize coupling introduced when classes or objects cooperate to perform distributed tasks

problem

Have some data in a spreadsheet or database

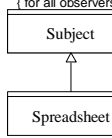


Have several ways to view that data simultaneously

How can multiple views of the same data be kept synchronized when the original data changes, especially if new views can be added at whim?

solution: Observer pattern

Subject Methods:
attach(observer);
detach(observer);
notify()
{ for all observers o, o.update(); }



Observer Methods:
update();

"Pull" style – observers have to query subject to see what changed.
"Push" style includes hints in call to update()

Spreadsheet Methods:
getCell(row, column)

Observer pattern leaves observed object ignorant of its viewers.

example for Observer pattern

```
class SignupSheet extends Observable {
    private List students = new ArrayList();
    void addStudent(String student) {
        students.add(student);
        setChanged();
        notifyObservers();
    }
    int size() {
        return students.size();
    }
}

class SignupObserver implements Observer {
    public void update(Observable o, Object arg) {
        System.out.println("Signup count: " + ((SignupSheet)o).size());
    }
}
```

Java support for Observer

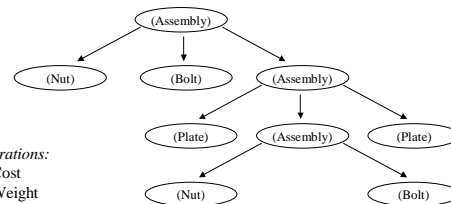
example for Observer pattern

```
class SignupSheet extends Observable {
    private List students = new ArrayList();
    void addStudent(String student) {
        students.add(student);
        setChanged();
        notifyObservers();
    }
    int size() {
        return students.size();
    }
}
```

```
SignupSheet s = new SignupSheet();
s.addStudent("turing");
s.addObserver(new SignupObserver());
s.addStudent("gatesb");
// outputs: "Signup count: 2"
```

```
class SignupObserver implements Observer {
    public void update(Observable o, Object arg) {
        System.out.println("Signup count: " + ((SignupSheet)o).size());
    }
}
```

problem



Operations:
getCost
getWeight
getSize

Want to apply several operations to a composite structure. Expect that may want other operations in future, but don't expect new node types.

problem

we have to add methods for this to every part – reasonable if we expect the set of parts to change more than the operations performed on them, but a pain otherwise

```

Part Methods:
float getWeight();
float getCost();
float getSize();
    
```

Want to apply several operations to a composite structure. Expect that may want other operations in future, but don't expect new node types.

solution: Visitor pattern

```

Visitor Methods:
visitNut(nut);
visitBolt(bolt);
visitPlate(plate);
visitAssembly(assembly);

Part Methods:
accept(visitor);

Bolt Overrides:
accept(visitor)
{ visitor.visitBolt(this); }
    
```

(Dependencies between Visitors and Parts have not been shown)

Encapsulate each operation to be performed in a separate Visitor object. Each Part calls a different method on the Visitor, independent of op.

Visitor pattern

```

class Foo {
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visitFoo(this); // or v.visit(this) with overloading
    }
}

class Visitor {
    void visitFoo(Foo n) {
        // perform work on a Foo
    }
    void visitBar(Bar n) { ... }
}
    
```

n.accept(v) performs a depth-first traversal of the structure rooted at *n*, performing *v*'s operation on each element of the structure

Visitor can carry along state (e.g. total cost so far)

Visitor pattern

```

a.accept(v)
b.accept(v)
v.visitPlate(b)
c.accept(v)
e.accept(v)
v.visitNut(e)
f.accept(v)
v.visitBolt(f)
v.visitAssembly(c)
d.accept(v)
v.visitPlate(d)
v.visitAssembly(a)
    
```

```

// Foo = Nut, Bolt, Plate, or Assembly
class Foo {
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visitFoo(this);
    }
}
    
```

Visitor pattern

```

class Foo {
    void accept(Visitor v) {
        v.visitFoo(this);
    }
}

class Visitor {
    void visitFoo(Foo n) {
        for each child of this node {
            child.accept(v);
        }
        // perform work on a Foo
    }
}
    
```

Alternative form: move traversal code into Visitor

Now need to repeat traversal code in all Visitors

Procedural and Interpreter style

Procedural: collects code for a type of operation, spreads apart code for a type of object
 Makes it easy to add operations, hard to add new object classes
 Example – the Visitor pattern

Interpreter: collects code for a type of object, spreads apart code for a type of operation
 Makes it easy to add new object classes, hard to add operations

For both, code for operations is quite similar
 Question is where to place that code
 Languages with multi-methods have an easier time here

Interpreter style

```
class AtomicPart {
    int getCost();
    int getWeight();
}
class Nut extends AtomicPart {
    int getCost() { ... }
    int getWeight() { ... }
}
class Bolt extends AtomicPart {
    int getCost() { ... }
    int getWeight() { ... }
}
```

Blackboard pattern

Generalizes Observer pattern
Multiple data sources
Multiple viewers
Completely decouples producers and consumers of info

Any source can post announcements on blackboard using agreed format; source has no idea who is paying attention

Viewers watch out for postings relevant to them, ignore irrelevant postings; may implement filtering

Think zephyr

categories of patterns

Creational patterns (some more!)

Minimize coupling introduced by object creation

Structural patterns

Minimize coupling introduced when classes or objects are assembled to form larger structures

Behavioral patterns

Minimize coupling introduced when classes or objects cooperate to perform distributed tasks

Prototype pattern

```
interface Duplicable {
    // return duplicate copy of self
    Object duplicate();
}
```



Create dynamic collections of instantiable objects, without having to make a special subclass for each

```
class Character implements Duplicable { ... }
class SpeechBubble implements Duplicable { ... }
```

```
cartoonToolbar.add(new Character(BOSS,DUMB));
cartoonToolbar.add(new Character(ENGINEER,EARNEST));
cartoonToolbar.add(new Character(DOG,EVIL));
cartoonToolbar.add(new SpeechBubble(THOUGHT));
cartoonToolbar.add(new SpeechBubble(TALK));
```

Prototype with clone()

```
class Character implements Cloneable {
    public Object clone() { ... }
}
class TalkBubble implements Cloneable {
    public Object clone() { ... }
}
```

```
cartoonToolbox.add(new Character(BOSS,DUMB));
cartoonToolbox.add(new Character(ENGINEER,EARNEST));
cartoonToolbox.add(new Character(DOG,EVIL));
cartoonToolbox.add(new TalkBubble(NORMAL));
cartoonToolbox.add(new TalkBubble(THOUGHT));
```

Java clone and Cloneable

Cloneable interface has no methods

There is a protected clone() method in Object

If a class implements Cloneable interface, Object.clone does field-by-field copy of Object; otherwise throws exception

Field-by-field copy is a shallow copy – member objects will be shared with clone

Will often need to override to do a deep copy – e.g. to make sure rep doesn't get exposed by shared objects

warnings and caveats

No guarantee a Cloneable object has a public clone() method

Are you getting a shallow or deep copy?

"In order for implementing the Cloneable interface to have any effect on a class, it and all of its superclasses must obey a fairly complex, unenforceable, and largely undocumented protocol" – Bloch

Flyweight pattern

If storage space for objects is a serious issue (e.g. many many objects), may wish to share as much state as possible

A flyweight is a shared object that can be used in multiple contexts simultaneously (e.g. characters in a document)

Flyweight idea:

- Separate the intrinsic (same across all contexts) and extrinsic (different for different contexts) state
- Share the intrinsic state (e.g. character code)
- Good when most of the object is immutable

interning

For immutable objects, fine to reuse existing objects instead of creating new ones (e.g. Java does this for String objects)

```
HashMap interns = new HashMap();
Object getInternedVersion(Object n) {
    if (interns.containsKey(n)) {
        return interns.get(n);
    } else {
        interns.put(n,n);
        return n;
    }
}
```

Two approaches:

1. Create the object, but perhaps then discard and return another
2. Check first before creating the object in the first place

Flyweight pattern

```
class Spoke {
    int length; // same for all spokes on a wheel
    Metal material; // same for all spokes on a wheel
    boolean crimped; // same for all spokes on a wheel
    int location; // index of slot in wheel that the spoke inhabits
    void tighten(int turns) {...} // tighten spoke by number of turns
}

class Wheel {
    Spoke[] spokes; // array of spokes, ordered by slot index
    void align() { ... spokes[i].tighten(numturns); ... } // align wheel
}
```

Typically many spokes, but few varieties of spoke

Flyweight pattern

```
class SpokeCore {
    int length; // same for all spokes on a wheel
    Metal material; // same for all spokes on a wheel
    boolean crimped; // same for all spokes on a wheel
    // location information omitted
}

class SpokeFull {
    SpokeCore spoke;
    int location; // index of slot in wheel that the spoke inhabits
    void tighten(int turns) {...} // tighten spoke by number of turns
}
```

Can share SpokeCore instances. This works, but can do better.

Flyweight pattern

```
class SpokeCore {
    int length; // same for all spokes on a wheel
    Metal material; // same for all spokes on a wheel
    boolean crimped; // same for all spokes on a wheel
    // location information omitted
    void tighten(int turns, int location) {...}
}

class Wheel {
    SpokeCore[] spokes; // array of spokes, ordered by slot index
    void align() { ... spokes[i].tighten(numturns,i); ... } // align wheel
}
```

Don't need SpokeFull anymore! Extrinsic state supplied by context. But code is a bit more complicated.