References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 293-303, 117-126

Java API

VisualWorks Smalltalk API

Prototype-based Languages

http://en.wikipedia.org/wiki/Prototype-based_programming
JavaScript The Definite Guide 4'th Ed, Flanagan, O'Reilly Press, 2002

http://c2.com/cgi/wiki?TemplateMethodPattern WikiWiki comments on the Template Method

http://wiki.cs.uiuc.edu/PatternStories/TemplateMethodPattern Stories about the Template Method

Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1995, pp. 325-330, 107-116
Observer

One-to-many dependency between objects

When one object changes state,
    all its dependents are notified and updated automatically
Structure

Subject

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

Observer
- Update()

ConcreteSubject
- GetState()

ConcreteObserver
- GetState()
- Update()

Observers

Subject

Observer
- Update()

ConcreteObserver

Subject

observer A

observer B

subject

Update()

GetState()

SetState()

observer A

observer B

subject

SetState()

Notify()

Update()

GetState()

Update()
Pseudo Java Example

public class Subject {
    Window display;
    public void someMethod() {
        this.modifyMyStateSomeHow();
        display.addText( this.text() );
    }
}

public class Subject {
    ArrayList observers = new ArrayList();
    public void someMethod() {
        this.modifyMyStateSomeHow();
        changed();
    }
    private void changed() {
        Iterator needsUpdate = observers.iterator();
        while (needsUpdate.hasNext() )
            needsUpdate.next().update( this );
    }
}

public class SampleWindow {
    public void update(Object subject) {
        text = ((Subject) subject).getText();
        Thread.sleep(10000).
    }
}

Abstract coupling - Subject and Observer

Broadcast communication

Updates can take too long
Some Language Support

<table>
<thead>
<tr>
<th>Smalltalk</th>
<th>Java</th>
<th>Ruby</th>
<th>Observer Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Observer</td>
<td></td>
<td>Abstract Observer class</td>
</tr>
<tr>
<td>Object &amp; Model</td>
<td>Observable</td>
<td>Observable</td>
<td>Subject class</td>
</tr>
</tbody>
</table>

Smalltalk Implementation
Object implements methods for both Observer and Subject.

Actual Subjects should subclass Model
Java's Observer

Class java.util.Observable

<table>
<thead>
<tr>
<th>Java</th>
<th>Observer Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Abstract Observer class</td>
</tr>
<tr>
<td>Observer class</td>
<td>Subject class</td>
</tr>
</tbody>
</table>

Observable object may have any number of Observers

Whenever the Observable instance changes, it notifies all of its observers

Notification is done by calling the update() method on all observers.

**Interface java.util.Observer**

Allows all classes to be observable by instances of class Observer
class Counter extends Observable {
    public static final String INCREASE = "increase";
    public static final String DECREASE = "decrease";
    private int count = 0;
    private String label;

    public Counter( String label ) { this.label = label; }

    public String label() { return label; }
    public int value() { return count; }
    public String toString() { return String.valueOf( count );}

    public void increase() {
        count++;
        setChanged();
        notifyObservers( INCREASE );
    }

    public void decrease() {
        count--;
        setChanged();
        notifyObservers( DECREASE );
    }
}

class IncreaseDetector implements Observer {
    public void update(java.util.Observable whatChanged,
                        java.lang.Object message) {
        if (message.equals(Counter.INCREASE)) {
            Counter increased = (Counter) whatChanged;
            System.out.println(increased.label() + " changed to " +
                                increased.value());
        }
    }
}

public static void main(String[] args) {
    Counter test = new Counter();
    IncreaseDetector adding = new IncreaseDetector();
    test.addObserver(adding);
    test.increase();
}
require 'observer'

class Counter
  include Observable

  attr_reader :count

  def initialize
    @count = 0
  end

  def increase
    @count += 1
    changed
    notify_observers(:INCREASE)
  end

  def decrease
    @count -= 1
    changed
    notify_observers(:DECREASE)
  end
end

class IncreaseDetector
  def update(type)
    if type == :INCREASE
      puts('Increase')
    end
  end
end

count = Counter.new()
puts count.count
count.add_observer(IncreaseDetector.new)
count.increase
puts count.count
count.increase
puts count.count
Implementation Issues
Mapping subjects (Observables) to observers

Use list in subject
Use hash table

```java
public class Observable {
    private boolean changed = false;
    private Vector obs;

    public Observable() {
        obs = new Vector();
    }

    public synchronized void addObserver(Observer o) {
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }

    public synchronized void addObserver(Observer o) {
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }
}
```
Observing more than one subject

If an observer has more than one subject how does it know which one changed?

Pass information in the update method
Deleting Subjects

In C++ the subject may no longer exist

Java/Smalltalk observer may prevent subject from garbage collection
Who Triggers the update?

Have methods that change the state trigger update

class Counter extends Observable {
    // some code removed
    public void increase() {
        count++;
        setChanged();
        notifyObservers(INCREASE);
    }
}

Have clients call Notify at the right time

class Counter extends Observable {
    // some code removed
    public void increase() {
        count++;
    }
}

Counter pageHits = new Counter();
pageHits.increase();
pageHits.increase();
pageHits.increase();
pageHits.increase();
pageHits.notifyObservers();
class ComplexObservable extends Observable {
    Widget frontPart = new Widget();
    Gadget internalPart = new Gadget();

    public void trickyChange() {
        frontPart.widgetChange();
        internalPart.anotherChange();
        setChanged();
        notifyObservers();
    }
}

class MySubclass extends ComplexObservable {
    Gear backEnd = new Gear();

    public void trickyChange() {
        super.trickyChange();
        backEnd.yetAnotherChange();
        setChanged();
        notifyObservers();
    }
}
Adding information about the change

push models - add parameters in the update method

class IncreaseDetector extends Counter implements Observer { // stuff not shown

    public void update( Observable whatChanged, Object message) {
        if ( message.equals( INCREASE) )
            increase();
    }
}

class Counter extends Observable { // some code removed
    public void increase() {
        count++;
        setChanged();
        notifyObservers( INCREASE );
    }
}
Adding information about the change

pull model - observer asks Subject what happened

class IncreaseDetector extends Counter implements Observer {
    public void update( Observable whatChanged ) {
        if ( whatChanged.didYouIncrease() )
            increase();
    }
}

class Counter extends Observable {
    public void increase() {
        count++;
        setChanged();
        notifyObservers();
    }
}
Scaling the Pattern
Java Event Model

AWT/Swing components broadcast events to Listeners

JDK1.0 AWT components broadcast an event to all its listeners

A listener normally not interested all events

Broadcasting to all listeners was too slow with many listeners
Java 1.1+ Event Model

Each component supports different types of events:

Component supports
- ComponentEvent
- KeyEvent
- MouseEvent
- FocusEvent
- MouseEvent

Each event type supports one or more listener types:

- MouseEvent
  - MouseListener
  - MouseMotionListener

Each listener interface replaces update with multiple methods

- MouseListener
  - mouseClicked()
  - mouseEntered()
  - mousePressed()
  - mouseReleased()

Listeners
- Only register for events of interest
- Don't need case statements to determine what happened
Small Models

Often an object has a number of fields (aspects) of interest to observers

Rather than make the object a subject make the individual fields subjects
  Simplifies the main object
  Observers can register for only the data they are interested in

VisualWorks ValueHolder

Subject for one value

ValueHolder allows you to:

  Set/get the value
    Setting the value notifies the observers of the change

  Add/Remove dependents
Template Method
Polymorphism

class Account {
    public:
        void virtual Transaction(float amount) {
            balance += amount;
        }
        Account(char* customerName, float InitialDeposit = 0);
    protected:
        char* name;
        float balance;
}

class JuniorAccount : public Account {
    public: void Transaction(float amount) {//code here}
}

class SavingsAccount : public Account {
    public: void Transaction(float amount) {//code here}
}

Account* createNewAccount(){
    // code to query customer and determine what type of
    // account to create
};

main() {
    Account* customer;
    customer = createNewAccount();
    customer->Transaction(amount);
}
Deferred Methods

class Account {
    public:
        void virtual Transaction() = 0;
    
}

class JuniorAccount : public Account {
    public
        void Transaction() { put code here}
    
}
class Account {
    public:
        void Transaction(float amount);
    protected:
        void virtual TransactionSubpartA();
        void virtual TransactionSubpartB();
        void virtual TransactionSubpartC();
}

void Account::Transaction(float amount) {
    TransactionSubpartA();  TransactionSubpartB();
    TransactionSubpartC();  // EvenMoreCode;
}

class JuniorAccount : public Account {
    protected:  void virtual TransactionSubpartA(); }

class SavingsAccount : public Account {
    protected:  void virtual TransactionSubpartC(); }

Account* customer;
customer = createNewAccount();
customer->Transaction(amount);
Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses

Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure
import java.awt.*;
class HelloApplication extends Frame {
    public void paint( Graphics display ) {
        int startX = 30;
        int startY = 40;
        display.drawString( "Hello World", startX, startY );
    }
}
Ruby LinkedList Example

class LinkedList
    include Enumerable

    def [](index)
        Code not shown
    end

    def size
        Code not shown
    end

    def each
        Code not shown
    end

    def push(object)
        Code not shown
    end

end

def testSelect
    list = LinkedList.new
    list.push(3)
    list.push(2)
    list.push(1)

    a = list.select { |x| x.even? }
    assert(a == [2])
end

Where does list.select come from?
# Methods defined in Enumerable

<table>
<thead>
<tr>
<th>all?</th>
<th>any?</th>
<th>collect</th>
<th>detect</th>
</tr>
</thead>
<tbody>
<tr>
<td>each_cons</td>
<td>each_slice</td>
<td>each_with_index</td>
<td>entries</td>
</tr>
<tr>
<td>enum_cons</td>
<td>enum_slice</td>
<td>enum_with_index</td>
<td>find</td>
</tr>
<tr>
<td>find_all</td>
<td>grep</td>
<td>include?</td>
<td>inject</td>
</tr>
<tr>
<td>map</td>
<td>max</td>
<td>member?</td>
<td>min</td>
</tr>
<tr>
<td>partition</td>
<td>reject</td>
<td>select</td>
<td>sort</td>
</tr>
<tr>
<td>sort_by</td>
<td>to_a</td>
<td>to_set</td>
<td>zip</td>
</tr>
</tbody>
</table>

All use "each"

Implement "each" and the above will work
Subclass AbstractCollection

Implement
   iterator
   size
   add

Get
   addAll
   clear
   contains
   containsAll
   isEmpty
   remove
   removeAll
   retainAll
   size
   toArray
   toString
Consequences

This is the most commonly used of the 23 GoF patterns

Important in class libraries

Inverted control structure

Parent class calls subclass methods

Java's paint method is a primitive operation called by a parent method

Beginning Java programs don't understand how the following works:

```java
import java.awt.*;
class HelloApplication extends Frame
{
    public void paint( Graphics display )
    {
        int startX = 30;
        int startY = 40;
        display.drawString( "Hello World", startX, startY );
    }
}
```
Consequences

Template methods tend to call:
- Concrete operations
- Primitive (abstract) operations
- Factory methods
- Hook operations

Provide default behavior that subclasses can extend

It is important to denote which methods
- Must overridden
- Can be overridden
- Can not be overridden
Refactoring to Template Method

Simple implementation
   Implement all of the code in one method
   The large method you get will become the template method

Break into steps
   Use comments to break the method into logical steps
   One comment per step

Make step methods
   Implement separate methods for each of the steps

Call the step methods
   Rewrite the template method to call the step methods

Repeat above steps
   Repeat the above steps on each of the step methods
   Continue until:
      All steps in each method are at the same level of generality
      All constants are factored into their own methods

Design Patterns Smalltalk Companion pp. 363-364.
Prototype
Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Applicability

Use the Prototype pattern when

A system should be independent of how its products are created, composed, and represented; and

When the classes to instantiate are specified at run-time; or

To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

When instances of a class can have one of only a few different combinations of state.
Insurance Example

Insurance agents start with a standard policy and customize it

Two basic strategies:

Copy the original and edit the copy

Store only the differences between original and the customize version in a decorator
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

Shallow Copy
Shallow Copy Verse Deep Copy

Original Objects

Deep Copy

Deeper Copy
Cloning Issues - C++ Copy Constructors

class Door {
    public:
        Door();
        Door (const Door&);
        virtual Door* clone() const;

        virtual void Initialize(Room*, Room*);
        // stuff not shown
    private:
        Room* room1;
        Room* room2;
    }

Door::Door (const Door& other) //Copy constructor {
    room1 = other.room1;
    room2 = other.room2;
}

Door* Door::clone() const {
    return new Door(*this);
}
Cloning Issues - Java Clone

Shallow Copy

class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

Deep Copy

public class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws CloneNotSupportedException {
        Door thisCloned = (Door) super.clone();
        thisCloned.room1 = (Room) room1.clone();
        thisCloned.room2 = (Room) room2.clone();
        return thisCloned;
    }
}
Prototype-based Languages

No classes

Behaviour reuse (inheritance)
   Cloning existing objects which serve as prototypes

Some Prototype-based languages

  Self
  JavaScript
  Squeak (eToys)
  Perl with Class::Prototyped module
Factory Method
Factory Method

A template method for creating objects

public class Example {
    protected Bar bar() { return new Bar(); }

    public void foo() {
        blah
        Bar soap = bar();
        blah;
    }
}
Maze Game Example

```
MapSite

   Room
   Wall
   Door

   RoomWithBomb
   EnchantedRoom

   BombedWall

   DoorWithSpell
   IronDoor

SecretPassageWall
```
Maze Game Example

class MazeGame{
    public Maze makeMaze() { return new Maze(); }
    public Room makeRoom(int n ) { return new Room( n ); }
    public Wall makeWall() { return new Wall(); }
    public Door makeDoor() { return new Door(); }

    public Maze CreateMaze(){
        Maze aMaze = makeMaze();
        Room r1 = makeRoom( 1 );
        Room r2 = makeRoom( 2 );
        Door theDoor = makeDoor( r1, r2);

        aMaze.addRoom( r1 );
        aMaze.addRoom( r2 );
        etc

        return aMaze;
    }
}

class BombedMazeGame extends MazeGame {
    public Room makeRoom(int n ) { 
        return new RoomWithABomb( n );
    }

    public Wall makeWall() { 
        return new BombedWall();
    }
}

Implementation Variation

class Hershey {

    public Candy makeChocolateStuff( CandyType id ) {
        if ( id == MarsBars ) return new MarsBars();
        if ( id == M&Ms ) return new M&Ms();
        if ( id == SpecialRich ) return new SpecialRich();

        return new PureChocolate();
    }
}

class GenericBrand extends Hershey {

    public Candy makeChocolateStuff( CandyType id ) {
        if ( id == M&Ms ) return new Flupps();
        if ( id == Milk ) return new MilkChocolate();
        return super.makeChocolateStuff(id);
    }
}
Using C++ Templates

template <class ChocolateType>
class Hershey
{
    public:
        virtual Candy* makeChocolateStuff();
}

template <class ChocolateType>
Candy* Hershey<ChocolateType>::makeChocolateStuff()
{
    return new ChocolateType;
}

Hershey<SpecialRich> theBest;
Smalltalk Variant

Return the class, caller creates an object

```smalltalk
chocolateStuff
^SpecialRich
```

```smalltalk
some code

candy := (self chocolateStuff) new

mode code
```
Use Factory Method When

A class can't anticipate the class of objects it must create

A class wants its subclasses to specify the objects it creates

You want to localize the knowledge of which help classes is used in a class

But when is this?
public class SDWitterServer {
    public void run(int port) throws IOException {
        ServerSocket input = new ServerSocket(port);

        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
    }

    void processRequest(InputStream in, OutputStream out) {
        do a bunch of stuff
    }

    etc.
}

This code requires us to send/receive data over the network, which at times is not convenient in unit tests.
Using Factory Method

public class SDWitterServer {
    public void run(int port) throws IOException {
        ServerSocket input = this.serverSocket(port);

        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
    }

    ServerSocket serverSocket(int port) {
        return new ServerSocket(port);
    }
}

etc.
public class TestServer extends SDWitterServer {
    MockServerSocket testSocket;

    ServerSocket serverSocket(int port) {
        return testSocket;
    }
}

Other than using a different type of socket it performs the operations as the parent class

public class Tests extends Testcase {
    public void testLogin() {
        TestServer server = new TestServer();
        server.testSocket = new MockServerSocket("client command to login");
        server.run();
        assertTrue(server.testSocket.serverResponse() = "the correct response here");
    }
}
MockServerSocket

Returns a fake (Mock) client connection

Fakes client connection
  Does not use network
  Contains fixed requests
  Records server responses