References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 293-303, 325-330

Java API

VisualWorks Smalltalk API

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

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

One-to-many dependency between objects

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

observer A
observer B

subject

GetState()
Update()
SetState()

observer A
observer B

subject

GetState()
Update()
SetState()

Subject
Attach(Observer)
Detach(Observer)
Notify()

Observer
Update()

ConcreteSubject
GetState()

ConcreteObserver
Update()
observerState

observers
subject

SetState()
Pseudo Java Example

```java
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

void addObserver(Observer o)
void clearChanged()
int countObservers()
void deleteObserver(Observer o)
void deleteObservers()
boolean hasChanged()
void notifyObservers()
void notifyObservers(Object arg)
void setChanged()

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
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();
Subject is self-consistent before Notification

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

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

class Counter extends Observable {
    // some code removed
    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  FocusEvent
  KeyEvent        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) {
    }
}

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

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

main() {
    Account* customer;
    customer = createNewAccount();
    customer->Transaction(amount);
}
class Account {
    public:
        void virtual Transaction() = 0;
    }

class JuniorAccount : public Account {
    public:
        void Transaction() { put code here}
    }

class SavingsAccount : 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);
    }
}
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
java.util.AbstractCollection

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.