References

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, 163-174
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);
}
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

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

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
Code Refactoring

Any change to a computer program which improves its

readability or
simplifies its structure

without changing its results

Source Wikipedia
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.
Composite
Motivation

How does the window hold and deal with the different items it has to manage?

Widgets are different that WidgetContainers
class Window {
    Buttons[] myButtons;
    Menus[] myMenus;
    TextAreas[] myTextAreas;
    WidgetContainer[] myContainers;

    public void update() {
        if (myButtons != null)
            for (int k = 0; k < myButtons.length(); k++)
                myButtons[k].refresh();
        if (myMenus != null)
            for (int k = 0; k < myMenus.length(); k++)
                myMenus[k].display();
        if (myTextAreas != null)
            for (int k = 0; k < myTextAreas.length(); k++)
                myTextAreas[k].refresh();
        if (myContainers != null)
            for (int k = 0; k < myContainers.length(); k++)
                myContainers[k].updateElements();
        etc.
    }

    public void fooOperation() {
        if (myButtons != null)
            etc.
    }
}
class Window {
    GUIWidgets[] myWidgets;
    WidgetContainer[] myContainers;

    public void update() {
        if ( myWidgets != null )
            for ( int k = 0; k < myWidgets.length(); k++ )
                myWidgets[k].update();
        if ( myContainers != null )
            for ( int k = 0; k < myContainers.length(); k++ )
                myContainers[k].updateElements();
        etc.
    }
}
Component implements default behavior for widgets when possible

Button, Menu, etc overrides Component methods when needed

WidgetContainer will have to overrides all widgetOperations

class WidgetContainer {
    Component[] myComponents;

    public void update() {
        if (myComponents != null)
            for (int k = 0; k < myComponents.length(); k++)
                myComponents[k].update();
    }
}
Issue - WidgetContainer Operations

Should the WidgetContainer operations be declared in Component?

**Pro - Transparency**
Declaring them in the Component gives all subclasses the same interface

All subclasses can be treated alike. (?)

**Con - Safety**
Declaring them in WidgetContainer is safer

Adding or removing widgets to non-WidgetContainers is an error

One out is to check the type of the object before using a WidgetContainer operation
Issue - Parent References

class WidgetContainer
{
    Component[] myComponents;

    public void update() {
        if (myComponents != null)
            for (int k = 0; k < myComponents.length(); k++)
                myComponents[k].update();
    }

    public add(Component aComponent) {
        myComponents.append(aComponent);
        aComponent.setParent(this);
    }
}

class Button extends Component {
    private Component parent;
    public void setParent(Component myParent) {
        parent = myParent;
    }
    etc.
More Issues

Should Component implement a list of Components?

The button etc. will have a useless data member

Child ordering is important in some cases

Who should delete components?
Applicability

Use Composite pattern when you want

To represent part-whole hierarchies of objects

Clients to be able to ignore the difference between compositions of objects and individual objects