CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2009
Doc 17 Template & Factory Method
March 18, 2009

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

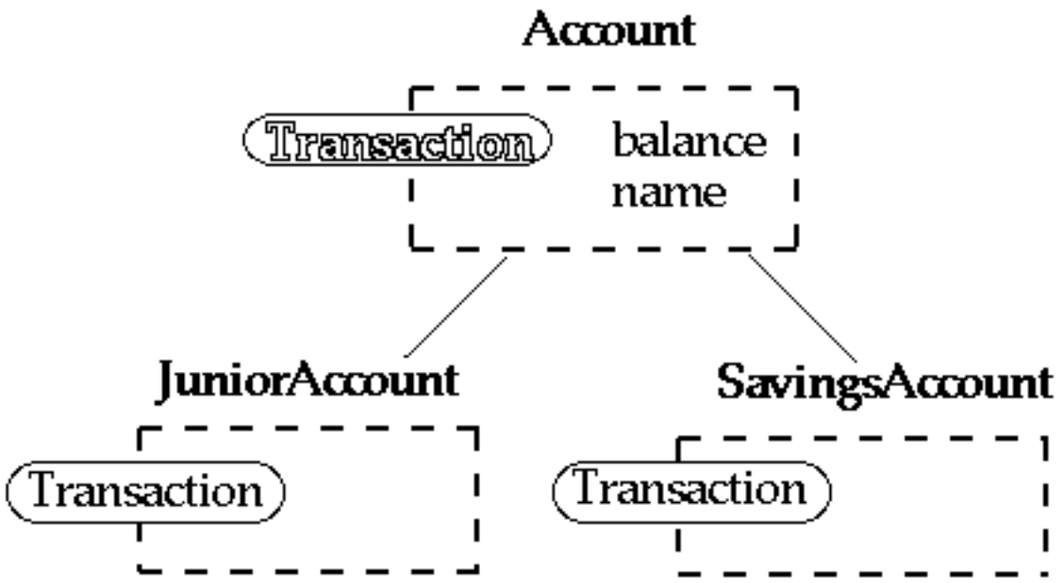
Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1995, pp. 325-330, 107-116

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);
}
```

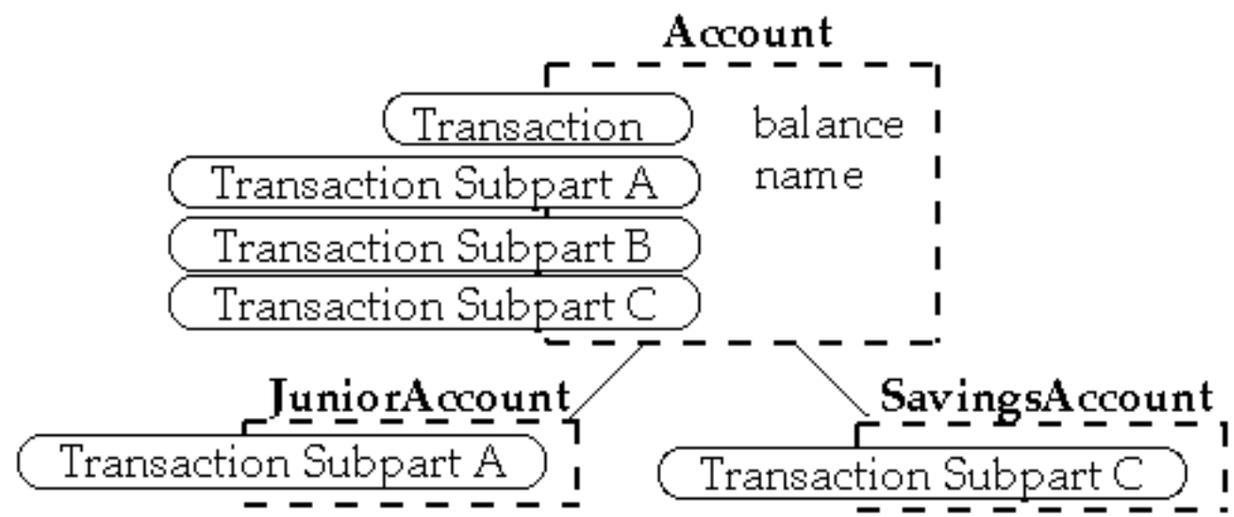
Deferred Methods




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

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

Template Method

```
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

Java Example

```
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 note 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

all?	any?	collect	detect
each_cons	each_slice	each_with_index	entries
enum_cons	enum_slice	enum_with_index	find
find_all	grep	include?	inject
map	max	member?	min
partition	reject	select	sort
sort_by	to_a	to_set	zip

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

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

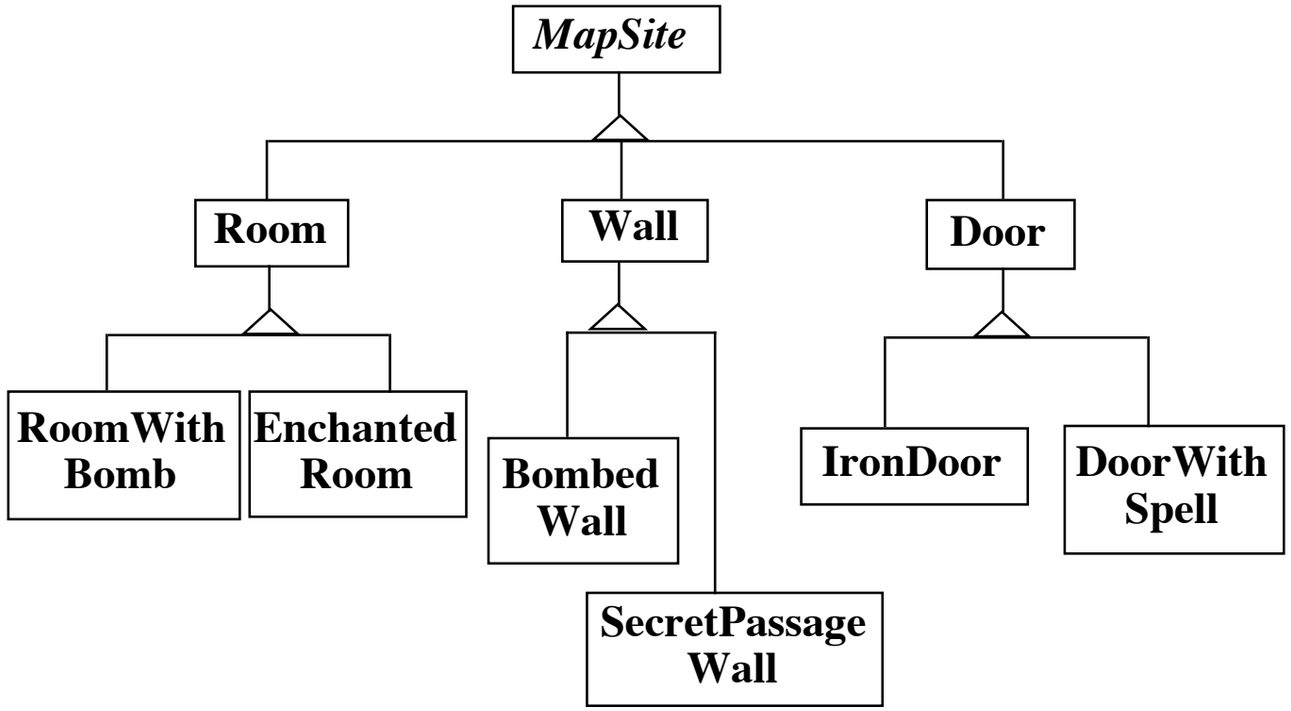
Design Patterns Smalltalk Companion pp. 363-364.

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



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

```
chocolateStuff
        ^SpecialRich
```

```
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?

CS 580 Example - Testing a Server

```
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.
```

TestServer

```
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");
    }
}
```


The situation is more complex than shown here. I simplified it to avoid details that might confuse people.

MockServerSocket

Returns a fake (Mock) client connection

Fakes client connection

- Does not use network
- Contains fixed requests
- Records server responses