CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2004
Doc 5 Template & Factory Method
Contents

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. 107-116, 325-330

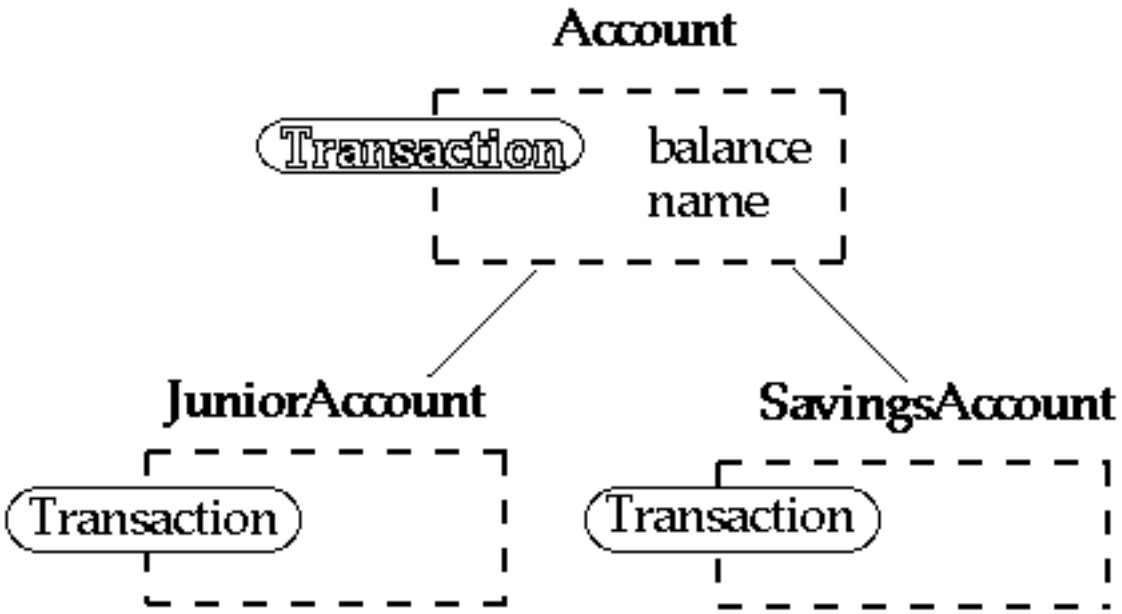
Pattern-Oriented Software Architecture: A System of Patterns (POSA 1), Buschman, Meunier, Rohnert, Sommerlad, Stal, 1996,

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998,pp. 63-76, 355-370

Template Method
Introduction
Polymorphism

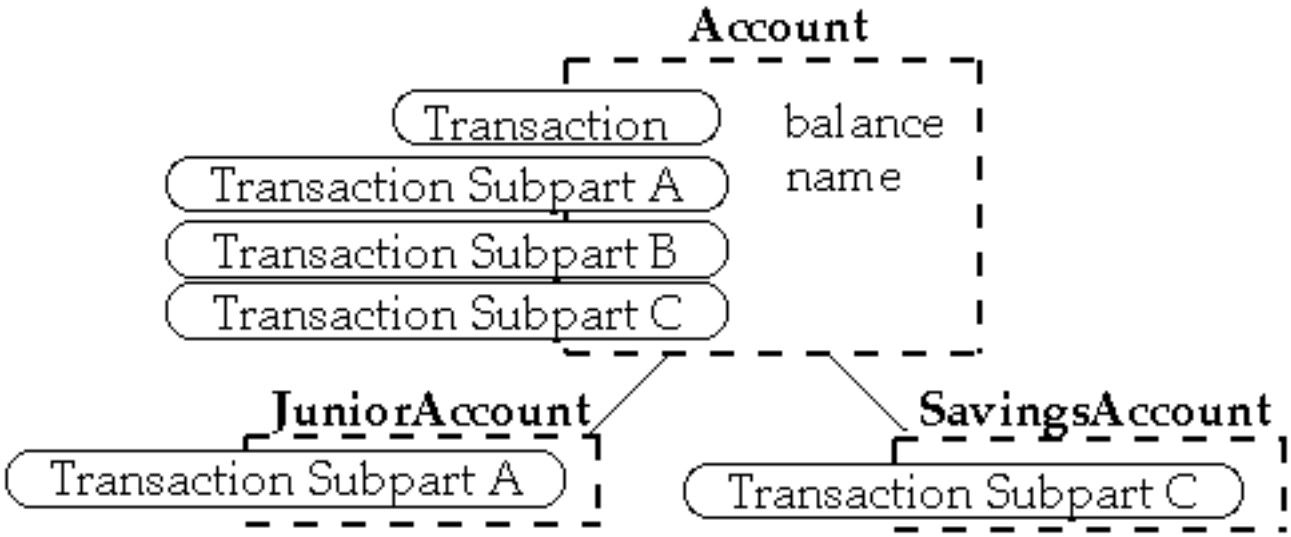
```
class Account {
   public:
      void virtual Transaction(float amount)
            { balance += amount;}
      Account(char* customerName, float InitialDeposit = 0);
   protected:
      char* name;
      float balance;
}

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

class SavingsAccount : public Account {
   public:   void Transaction(float amount) {//  put 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 Methods




```
class Account {
   public:
      void Transaction(float amount);
      void virtual TransactionSubpartA();
      void virtual TransactionSubpartB();
      void virtual TransactionSubpartC();
}

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

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

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

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

Template Method- The Pattern
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

Motivation

An application framework with Application and Document classes

Abstract Application class defines the algorithm for opening and reading a document

```
void Application::OpenDocument (const char* name ) {
  if (!CanNotOpenDocument (name)) {
    return;
  }

  Document* doc = DoCreateDocument();

  if (doc) {
    _docs->AddDocument( doc);
    AboutToOpenDocument( doc);
    Doc->Open();
    Doc->DoRead();
  }
}
```

Smalltalk Examples
PrintString

```
Object>>printString
   | aStream |
   aStream := WriteStream on: (String new: 16).
   self printOn: aStream.
   ^aStream contents

Object>>printOn: aStream
   | title |
   title := self class printString.
   aStream nextPutAll:
         ((title at: 1) isVowel ifTrue: ['an '] ifFalse: ['a ']).
   aStream nextPutAll: title
```

Object provides a default implementation of printOn:

Subclasses just override printOn:

Collections & Enumeration

Standard collection iterators
collect:, detect:, do:, inject:into:, reject:, select:

```
Collection>>collect: aBlock
   | newCollection |
   newCollection := self species new.
   self do: [:each | newCollection add: (aBlock value: each)].
   ^newCollection

Collection>>do: aBlock
   self subclassResponsibility

Collection>>inject: thisValue into: binaryBlock
   | nextValue |
   nextValue := thisValue.
   self do: [:each | nextValue := binaryBlock value: nextValue value: each].
   ^nextValue

Collection>>reject: aBlock
   ^self select: [:element | (aBlock value: element) == false]

Collection>>select: aBlock
   | newCollection |
   newCollection := self species new.
   self do: [:each | (aBlock value: each) ifTrue: [newCollection add: each]].
   ^newCollection
```

Subclasses only have to implement:
species, do:, add:

Applicability

Template Method pattern should be used:

- To implement the invariant parts of an algorithm once.

  Subclasses implement behavior that can vary

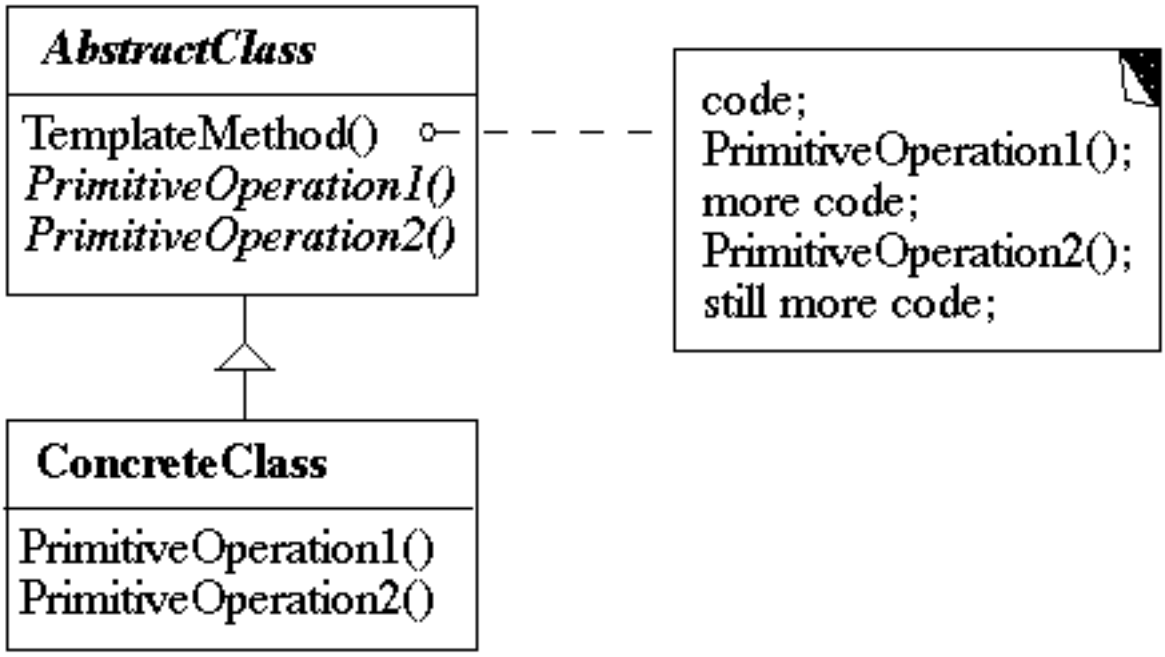
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication

To control subclass extensions

  Template method defines hook operations

  Subclasses can only extend these hook operations

Structure



Participants

- AbstractClass

  Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm

  Implements a template method defining the skeleton of an algorithm

- ConcreteClass

  Implements the primitive operations

  Different subclasses can implement algorithm details differently

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 operations - must be overridden
- Factory methods
- Hook operations

  Methods called in Template method and have default implementation in AbstractClass

  Provide default behavior that subclasses can extend

  Smalltalk's printOn: aStream is a hook operation

It is important to denote which methods

- Must overridden
- Can be overridden
- Can not be overridden

Implementation

Using C++ access control

- Primitive operations can be made protected so can only be called by subclasses
- Template methods should not be overridden - make nonvirtual

Minimize primitive operations

Naming conventions

- Some frameworks indicate primitive methods with special prefixes
- MacApp use the prefix "Do"

Implementing a Template Method[1]

- 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 method 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

[1] See Design Patterns Smalltalk Companion pp. 363-364. Also see Reusability Through Self-Encapsulation, Ken Auer, Pattern Languages of Programming Design, 1995, pp. 505-516

Constant Methods

Template method is common in lazy initialization[2]

```
public class Foo {
   Bar field;

   public Bar getField() {
      if (field == null)
         field = new Bar( 10);
      return field;
   }
}
```

What happens when subclass needs to change the default field value?

```
   public Bar getField() {
      if (field == null)
         field = defaultField();
      return field;
   }
   protected Bar defaultField() {
      return new Bar( 10);
   }
```

Now a subclass can just override defaultField()

[2] See http://www.eli.sdsu.edu/courses/spring01/cs683/notes/coding/coding.html#Heading19 or Smalltalk Best Practice Patterns, Kent Beck, Prentice Hall, 1997 pp. 85-86

The same idea works in constructors

```
public Foo() {
   field := defaultField();
}
```

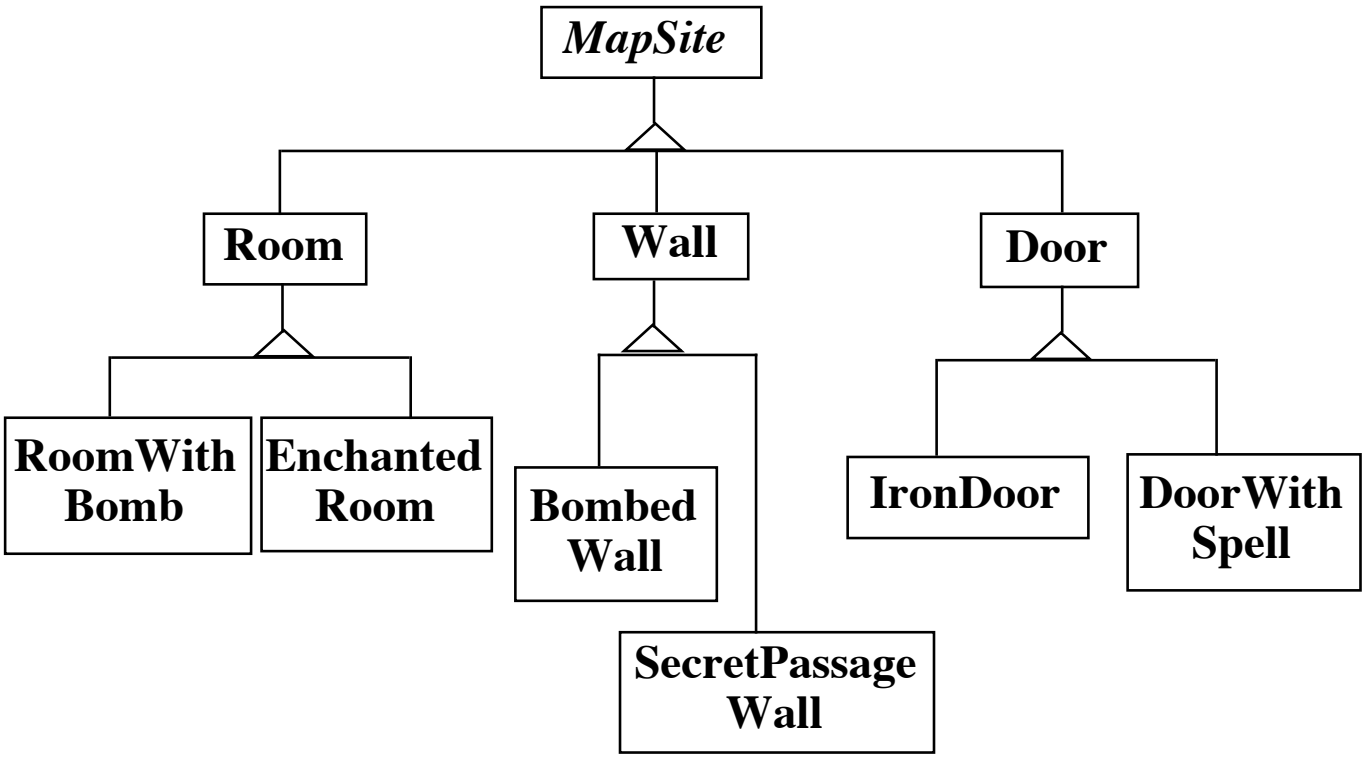
Now a subclass can change the default value of a field by overriding the default value method for that field

Factory Method

A template method for creating objects

Example - Maze Game

Classes for Mazes



Now a maze game has to make a maze

Maze Class Version 1

```
class MazeGame
  {

  public Maze createMaze()
    {
    Maze aMaze = new Maze();

    Room r1 = new Room( 1 );
    Room r2 = new Room( 2 );
    Door theDoor = new Door( r1, r2);

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

    etc.

    return aMaze;
    }
  }
```

How do we make other Mazes?

Subclass MazeGame, override createMaze

```
class BombedMazeGame extends MazeGame
  {

  public Maze createMaze()
    {
    Maze aMaze = new Maze();

    Room r1 = new RoomWithABomb( 1 );
    Room r2 = new RoomWithABomb( 2 );
    Door theDoor = new Door( r1, r2);

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

    etc.
```

Note the amount of cut and paste!

How do we make other Mazes?

Use Factory Method

```
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;
    }
  }
```

Now subclass MazeGame override make methods

CreateMaze method stays the same

```
class BombedMazeGame extends MazeGame
  {

  public Room makeRoom(int n )
    {
    return new RoomWithABomb( n );
    }

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

Applicability

Use 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

Consequences

- Eliminates need to hard code specific classes in code
- Requires subclassing to vary types used
- Provides hooks for subclasses
- Connects Parallel class hierarchies

Implementation
Two Major Varieties

- Top level Factory method is in an abstract class

```
abstract class MazeGame
   {
   public Maze makeMaze();
   public Room makeRoom(int n );
   public Wall makeWall();
   public Door makeDoor();
   etc.
   }

class MazeGame
   {
   public:
      virtual Maze* makeMaze() = 0;
      virtual Room* makeRoom(int n ) = 0;
      virtual Wall* makeWall() = 0;
      virtual Door* makeDoor() = 0;
```

- Top level Factory method is in a concrete class

  See examples on previous slides

Implementation - Continued
Parameterized Factory Methods

Let the factory method return multiple products

```
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();
    }
  }
```

C++ Templates to Avoid Subclassing

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

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

Hershey<SpecialRich> theBest;
```

Java forName and Factory methods

With Java's reflection you can use a Class or a String to specify which type of object to create

Using a string replaces compile checks with runtime errors

```
class Hershey
  {
  private String chocolateType;

  public Hershey( String chocolate )
    {
    chocolateType = chocolate;
    }

  public Candy makeChocolateStuff( )
    {
    Class candyClass =  Class.forName( chocolateType );
    return (Candy) candyClass.newInstance();
    }


Hershey theBest = new Heshsey( "SpecialRich" );
```

Clients Can Use Factory Methods

```
class CandyStore
  {
  Hershey supplier;
  public restock()
    {
    blah

    if ( chocolateStock.amount() < 10 )
      {
      chocolateStock.add(
        supplier.makeChocolateStuff() );
      }

  blah
```

Exercises

1. Find the template method in the Java class hierarchy of Frame that calls the paint(Graphics display) method.

3. Find other examples of the template method in Java or Smalltalk.

4. When I did problem one, my IDE did not help much. How useful was your IDE/tools? Does this mean imply that the use of the template method should be a function of tools available in a language?

5. Much of the presentation in this document follows very closely to the presentation in *Design Patterns: Elements of Reusable Object-Oriented Software.* This seems like a waste of lecture time (and perhaps a violation of copyright laws). How would you suggest covering patterns in class?