Software Engineering - Methods

- Overview
  - Design Patterns
  - Testing
  - Professional Issues
- Lecturers
  - Jeff Magee <jnm@doc.ic.ac.uk>
  - Michael Huth <mrh@doc.ic.ac.uk>

Design Patterns

Objective:

Increasing the flexibility, modularity and reusability of OO designs.

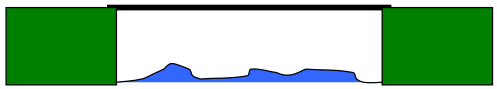
Jeff Magee
jnm@doc.ic.ac.uk
rm 572A

Bridges - (thanks to Sue Eisenbach for this)

- A bridge is a structure which is used for traversing a chasm.
- In its basic form it consists of a beam constructed from a rigid material.
- The two ends of the beam are fixed at opposite edges of the chasm.

Bridges

- The bridge will fulfill its function if the rigidity of the beam can support the loads that go over it.
- Heavier loads may tax the rigidity of the bridge.
- The rigidity depends on both the length and the material that the beam is made of.



Modifying the design

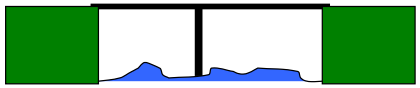
- If the bridge might fail
  - heaviness of the load
  - size of span
  - material of construction
- Then modify bridge design
  - increase the rigidity
  - decrease the span

Increase the rigidity

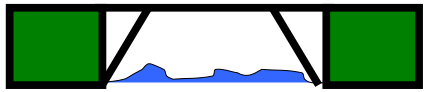
- Box girder
  - redistribute material
- The arch
- The suspension bridge

Decrease the span

- Divide the chasm



- Extend the edges of the chasm



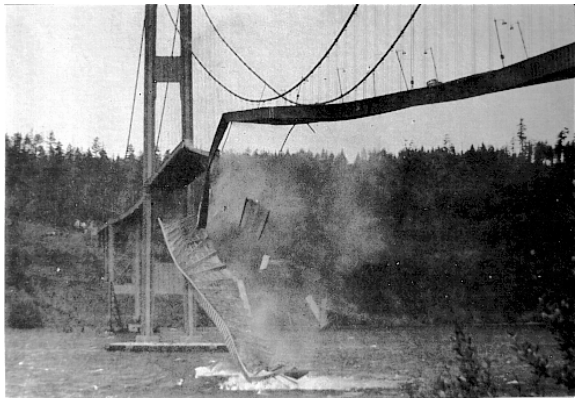
Civil engineering design patterns

- These are all the design patterns of bridge design.
- Civil engineers only build bridges following one of the designs shown.
- The idea of design patterns comes from architects who also follow a fixed number of designs.
- Why should software design be different from design in other engineering disciplines?

Tacoma Narrows Bridge

November 7, 1940, at approximately 11:00 AM,

the End

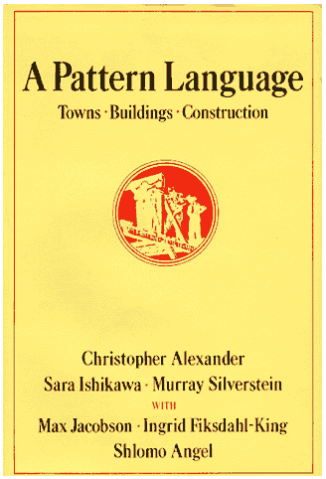


What is a design pattern?

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Christopher Alexander

From Architecture…

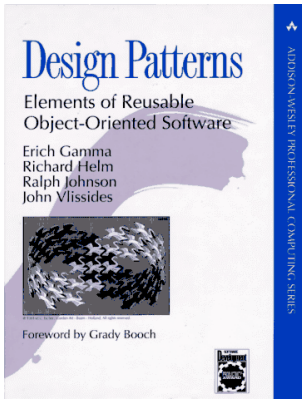


Christopher Alexander, Sara Ishikawa, Murray Silverstein, *with*
Max Jacobsen, Ingris Fiksdahl-King, and Shlomo Angel.

A Pattern Language: Towns,Buildings, Construction.

Oxford University Press, New York, 1977.

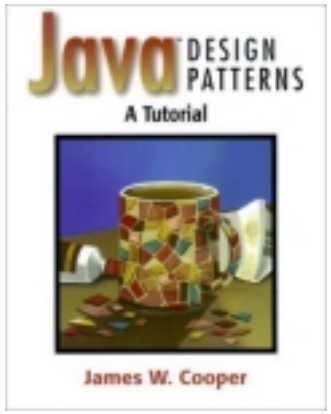
Gang of Four (GoF)



Design Patterns
Erich Gamma,
Richard Helm,
Ralph Johnson,
John Vlissides

Addison-Wesley 1995

Supplementary Text



Java Design Patterns
James W. Cooper

Addison-Wesley 2000

How do you describe a pattern?

- name
  - capture essence of pattern
- problem
  - intent, when to apply pattern, context
- solution
  - abstract, not concrete + examples of use
- consequences
  - results and trade-off of applying the pattern.

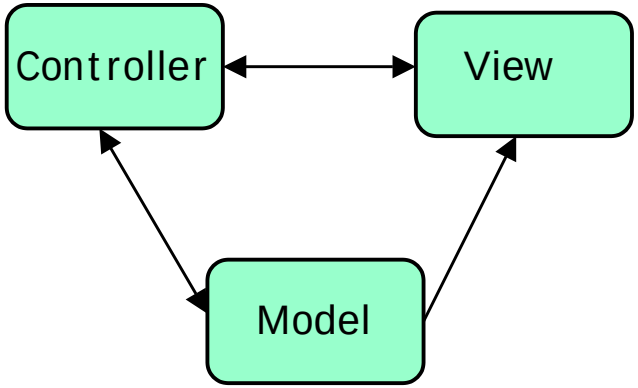
Rule of Three

- A pattern must have occurred in at least three existing systems.
  - Discovered rather than invented.
  - preferably reviewed by a third-party.

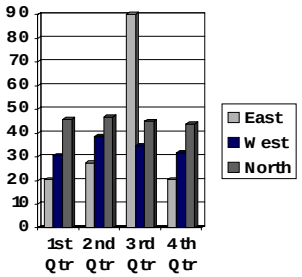
The first pattern (from Smalltalk)

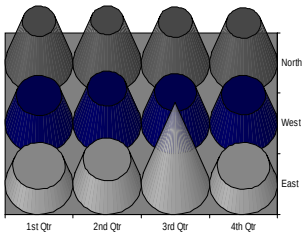
- Model-View-Controller group of classes from Smalltalk is used to build interfaces (Java Swing classes have a similar structure, EPOC uses MVC).
- Model -application object,
  View - presentation on screen,
  Controller - how user input controls user interface.
- Decoupling into 3 increases flexibility and reuse.
- A view must keep itself up-to-date with a subscribe/notify protocol.
- Model must tell views when they change.
- You can have several controllers eg for command keys, for pop-up menus and a do-nothing controller

MVC

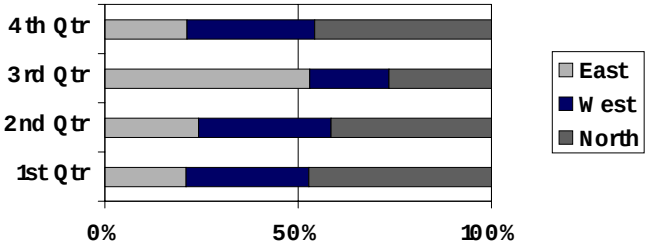


Two views on the same data





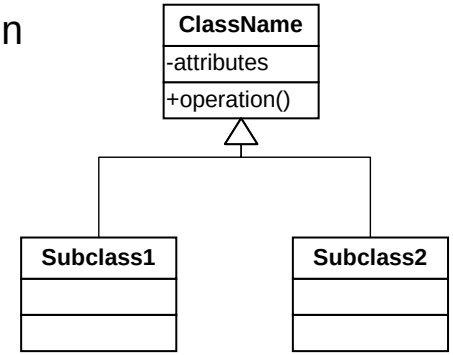
Another view + data from model



	1st Qtr	2nd Qtr	3rd Qtr	4th Qtr
North	45.9	46.9	45	43.9
West	30.6	38.6	34.6	31.6
East	20.4	27.4	90	20.4

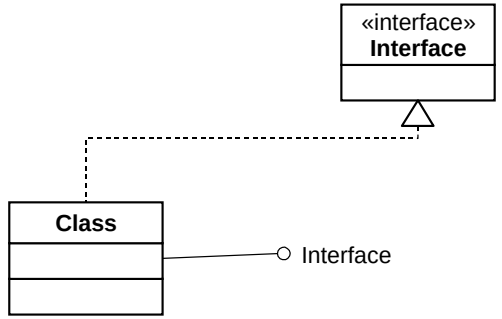
A digression - UML notation

Generalization
(which we consistently use to mean inheritance)



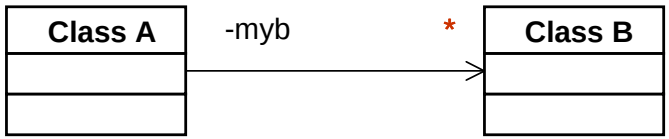

```
class Subclass1 extends ClassName {
…
}
```


UML – implementing interfaces




```
class Foo implements MyInterface {
}
```


UML - associations




```
class A {
   private B myb;
      …
}
```

```
class A {
   private B [] myb;
      …
}
```


GoF pattern description template

- Name & classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Pattern

Canonical Form

- Name
  - + classification
- Problem
- Context
  - preconditions for applicability
- Forces
  - example scenario sometimes used for motivation
- Solution
- Examples
- Resulting Context
- Rationale
- Related Patterns
- Known Uses

Classification (GoF)

- Creational Patterns
  - Abstract Object Instantiation
- Structural Patterns
  - Compose Objects into larger structures
- Behavioral Patterns
  - deal with algorithms and control flow between objects.

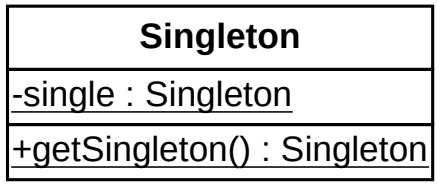
Creational patterns

- Deals with object creation
- Examples:
  - **singleton** - for creating classes which must have only a single instance (e.g. a printer spooler)
  - **factory method** - used when a class can't anticipate the class of objects it must create but it wants its subclasses to specify the objects it creates
  - **abstract factory** - provides an interface for creating families of related objects without the need to specify their concrete classes

Singleton

Ensure a class has one instance, and provide a global point of access to it.

Structure



Singleton - example

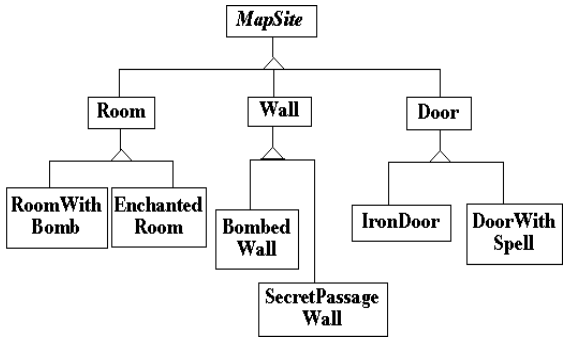
```
public class PrintSpooler {
    // a prototype for a spooler class,
    // such that only one instance can ever exist
    private static PrintSpooler spooler;
    private PrintSpooler() { //private constructor }
    public static synchronized
        PrintSpooler getSpooler() {
        if (spooler == null)
            spooler =  new PrintSpooler();
        return spooler;
    }

    public void print(String s) {
        System.out.println(s);
    }
}
```


Singleton usage

```
public class finalSpool {

    public finalSpool() {
        PrintSpooler spl = PrintSpooler.getSpooler ();
        spl.print ("Printing data");
    }

    static public void main(String argv[]) {
        new finalSpool();
    }

}
```

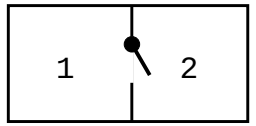

Factory Method Example - Maze Game

- Classes for Mazes



- Now a maze game has to make a maze, so we might have something like:

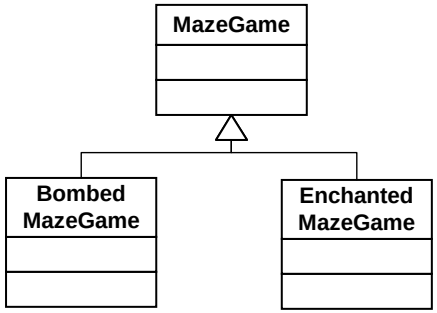
Maze Class - Version 1




```
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 );
          r1.setSide( North, new Wall() );
          r1.setSide( East, theDoor );
          r1.setSide( South, new Wall() );
          r1.setSide( West, new Wall() );
          r2.setSide( North, new Wall() );
          r2.setSide( East, new Wall() );
          r2.setSide( South, new Wall() );
          r2.setSide( West, theDoor );
          return aMaze;}}
```

How do we make Other Mazes?

- Idea 1 - Subclass MazeGame, override createMaze



Note the amount of cut and paste!

```
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 );
          r1.setSide( North, new BombedWall() );
          r1.setSide( East, theDoor );
          r1.setSide( South, new BombedWall() );
          r1.setSide( West, new BombedWall() );
etc.
```


Factories: encapsulating object creation

- When you discover that you need to add new types to a system, the most sensible first step is to use polymorphism to create a common interface to those new types.
- This separates the rest of the code in your system from the knowledge of the specific types that you are adding. New types may be added without disturbing existing code ... or so it seems.
- At first it would appear that the only place you need to change the code in such a design is the place where you inherit a new type, but this is not quite true.
- You must still create an object of your new type, and at the point of creation you must specify the exact constructor to use.

- Thus, if the code that creates objects is distributed throughout your application, you have the same problem when adding new types – you must still chase down all the points of your code where type matters.
- It happens to be the creation of the type that matters in this case rather than the use of the type (which is taken care of by polymorphism).
- The solution is to force the creation of objects to occur through a common factory rather than to allow the creational code to be spread throughout your system.
- If all the code in your program must go through this factory whenever it needs to create one of your objects, then all you must do when you add a new object is to modify the factory.

How do we make Other Mazes? Idea 2 - 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();
         // next slide →
      return aMaze;
  }
}
```



```
Room r1 = makeRoom( 1 );
Room r2 = makeRoom( 2 );
Door theDoor = makeDoor(r1,r2);
aMaze.addRoom( r1 );
aMaze.addRoom( r2 );
r1.setSide( North, makeWall() );
r1.setSide( East, theDoor );
r1.setSide( South, makeWall() );
r1.setSide( West, makeWall() );
r2.setSide( North, makeWall() );
r2.setSide( East, makeWall() );
r2.setSide( South, makeWall() );
r2.setSide( West, theDoor );
```


Now subclasses of 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();
    }
```


Factory Method - summary

- "Create objects in a separate operation so that subclasses can override the way they're created"

Abstract factory

- Provides an interface for creating families of related objects without the need to specify their concrete classes
  - create a window and an attached scrollbar in a consistent visual style (Gnome vs NT)
  - no specification of the individual styles
- Different subclasses of **AbstractFactory** class are responsible for creating objects appropriate to particular families.

Example code

```
interface AbstractMazeFactory {
     public Maze makeMaze();
     public Room makeRoom(int n );
     public Wall makeWall();
     public Door makeDoor();
}

class MazeFactory implements AbstractMazeFactory {
  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(AbstractMazeFactory factory){
  Maze aMaze = factory.makeMaze();
  Room r1 = factory.makeRoom( 1 );
  Room r2 = factory.makeRoom( 2 );
  Door theDoor = factory.makeDoor(r1,r2);
  aMaze.addRoom( r1 );
  aMaze.addRoom( r2 );
  r1.setSide( North, factory.makeWall() );
  r1.setSide( East, theDoor );
  r1.setSide( South, factory.makeWall() );
  r1.setSide( West, factory.makeWall() );
  r2.setSide( North, factory.makeWall() );
  r2.setSide( East, factory.makeWall() );
  r2.setSide( South, factory.makeWall() );
  r2.setSide( West, theDoor );
  return aMaze;
}
```

```
class EnchantedMazeFactory implements
                          AbstractMazeFactory {
  public Maze makeMaze(){
    return new EnchantedMaze();
  }
  public Room makeRoom(int n){
    return new EnchantedRoom(n);
  }
  public Wall makeWall(){
    return new EnchantedWall();
  }
  public Door makeDoor(){
    return new EnchantedDoor();
  }
}
```

Usage:

```
   Maze m = createMaze(new EnchantedMazefactory());
```

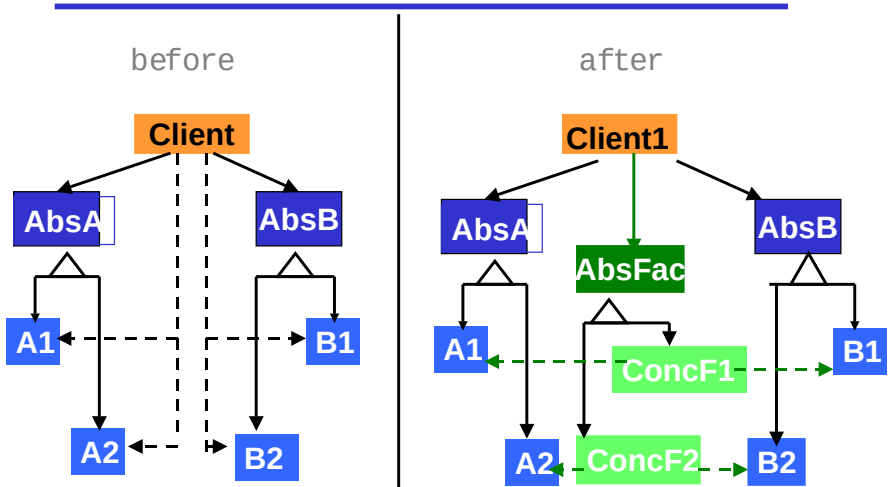
Use **AbstractFactory** when:

- A system should be independent of how its products are created, composed and represented
- a system needs to be configured with one of a number of families of products
- a family of related objects is designed to be used together, and this constraint needs to be enforced

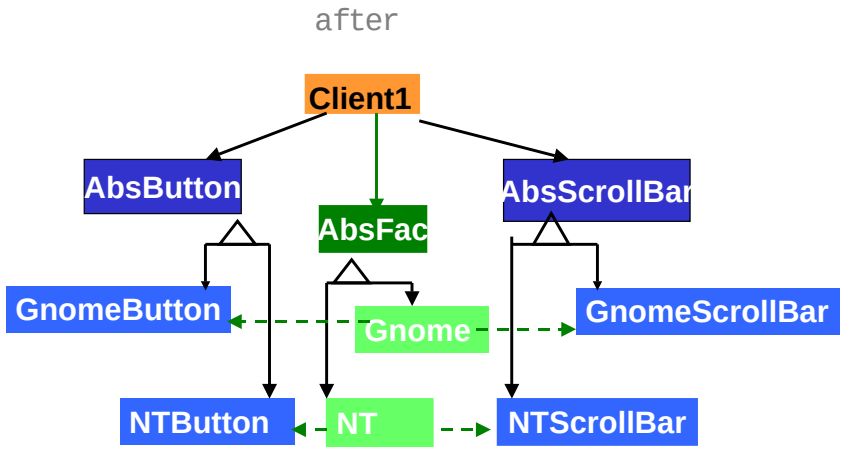
AbstractFactory classes

- **AbstractFactory**-declares operations which create abstract product objects
- **ConcreteFamily** subclasses - implements operations for particular families
- **AbstractProduct** - declares interface for one type of products
- **ConcreteProduct**- implements **AbstractProduct** interface and defines a product type to be created by corresponding concrete factory
- **Client** - uses only interfaces - so independent of particular family in use.

Application of **AbstractFactory**

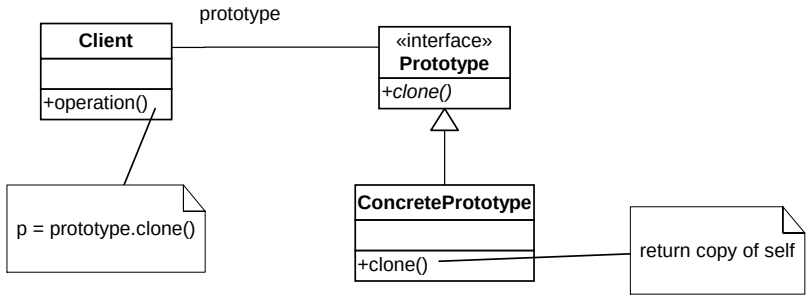


Windows Application of **AbstractFactory**



Prototype - creational

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



Prototype - implementation in Java

`public interface Cloneable`

A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of that class.

Attempts to clone instances that do not implement the `Cloneable` interface result in the exception `CloneNotSupportedException` being thrown.

The interface `Cloneable` declares no methods.

Java Hack - deep cloning

```
public Object deepClone() {
 try {
  ByteArrayOutputStream b
    = new ByteArrayOutputStream();
  ObjectOutputStream out
    = new ObjectOutputStream(b);
  out.writeObject(this);
  ByteArrayInputStream bIn
    = new ByteArrayInputStream(b.toByteArray());
  ObjectInputStream oi = new ObjectInputStream(bIn);
  return oi.readObject());
 } catch (Exception e) {
   System.out.println("exception:"+e.getMessage());
   e.printStackTrace();
   return null;
 }
}
```


Questions

- What Java standard interface must a class implement for `deepClone` to work.
- How does deep clone work?