# CS 635 Advanced Object-Oriented Design & Programming
**Spring Semester, 2002**
**Doc 17 Prototype, Flyweight, Facade**
**Contents**

## References

*Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 117-126, 185-206

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 63-89, 179-211

## Prototype
## Intent

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

## Applicability

Use the Prototype pattern when

- A system should be independent of how its products are created, composed, and represented; **and**
- when the classes to instantiate are specified at run-time; or
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
- when instances of a class can have one of only a few different combinations of state.
- It may be easier to have the proper number of prototypes and clone them rather than instantiating the class manually each time

## Insurance Example

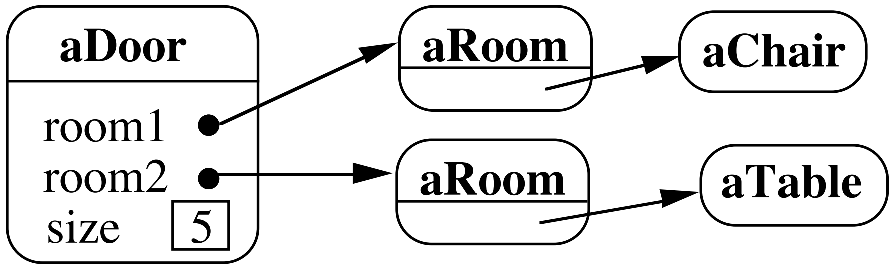
Insurance agents start with a standard policy and customize it

Two basic strategies:

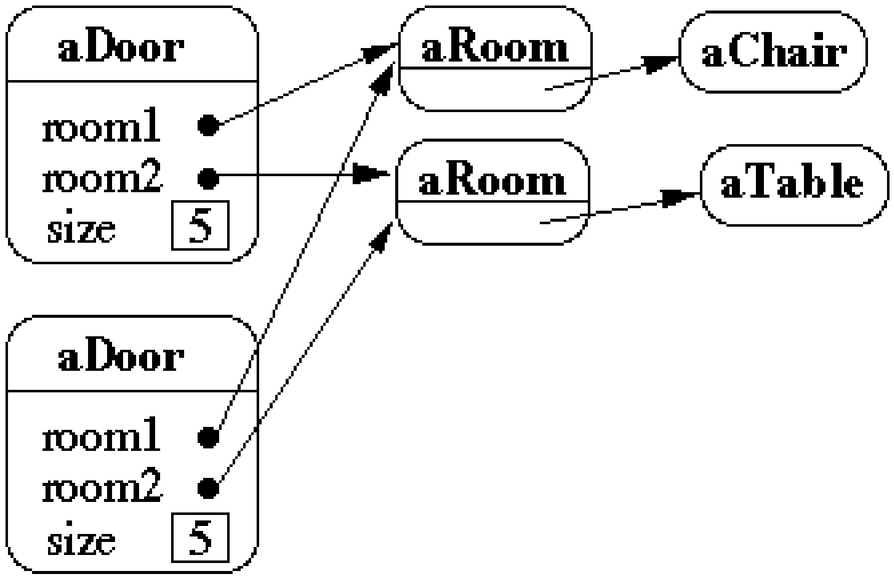
- Copy the original and edit the copy
- Store only the differences between original and the customize version in a decorator

## Copying Issues
## Shallow Copy Verse Deep Copy

### Original Objects

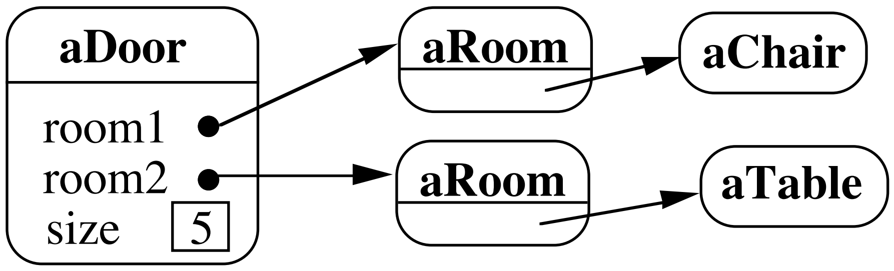


### Shallow Copy

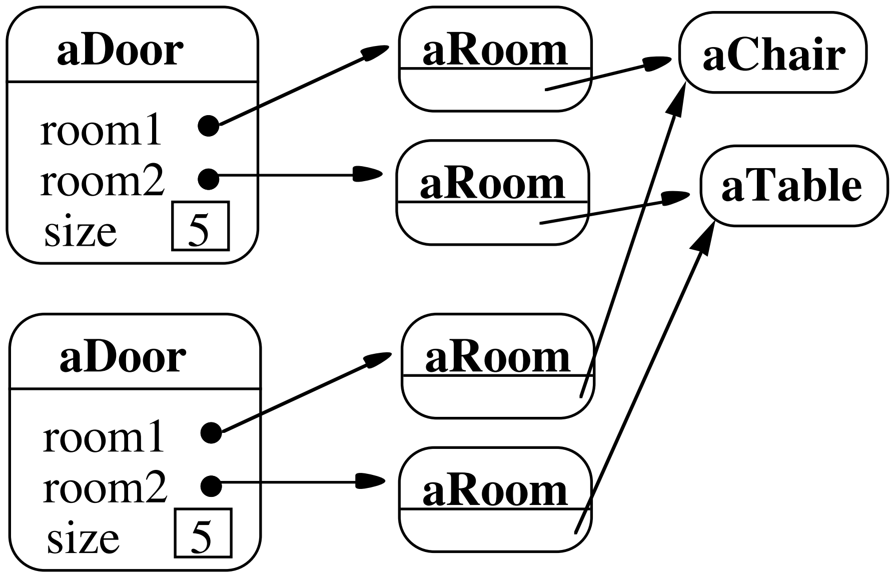


# Shallow Copy Verse Deep Copy

## Original Objects

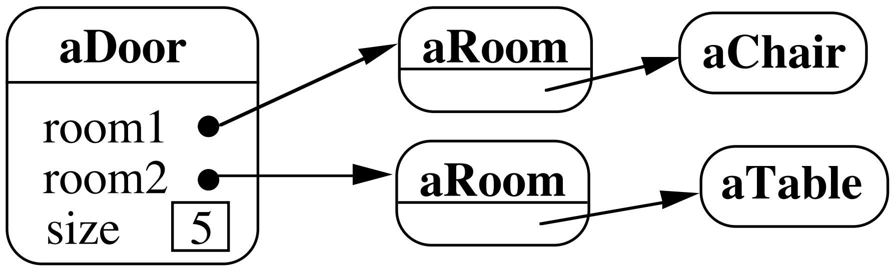


## Deep Copy

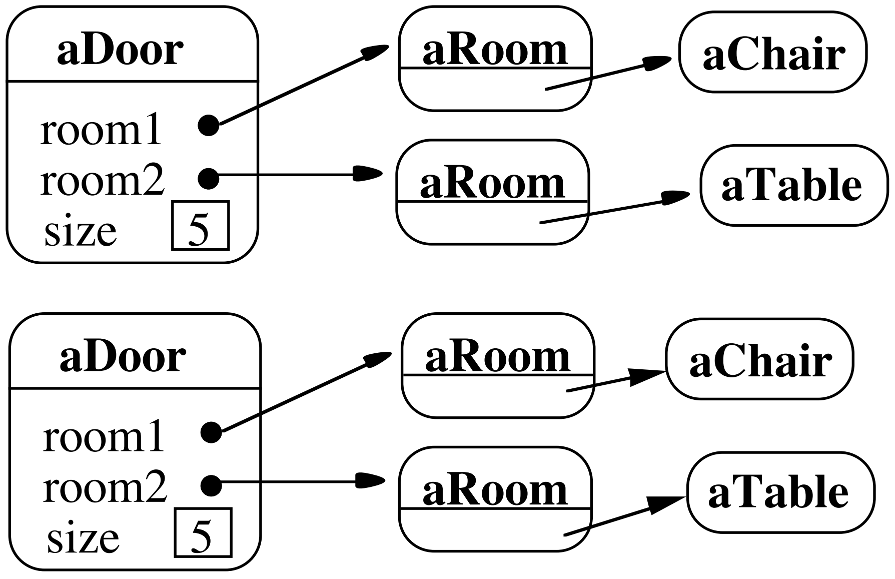


## Shallow Copy Verse Deep Copy

### Original Objects



### Deeper Copy



## Cloning Issues
## How to in C++ - Copy Constructors

```
class Door
   {
   public:
      Door();
      Door( const Door&);

      virtual Door* clone() const;

      virtual void Initialize( Room*, Room* );
      // stuff not shown
   private:
      Room* room1;
      Room* room2;
   }

Door::Door ( const Door& other ) //Copy constructor
   {
   room1 = other.room1;
   room2 = other.room2;
   }

Door* Door::clone()  const
   {
   return new Door( *this );
   }
```

## How to in Java - Object clone()

protected Object clone() throws CloneNotSupportedException

Default is shallow copy

Returns:
A clone of this Object.

Throws: OutOfMemoryError
If there is not enough memory.

Throws: CloneNotSupportedException
Object explicitly does not want to be cloned, or it does not support the Cloneable interface.

```
class Door implements Cloneable {
  public void Initialize( Room a, Room b)
    { room1 = a; room2 = b; }

  public Object clone() throws
            CloneNotSupportedException {
    // modify this method for deep copy
    // no need to implement this method for shallow copy
    return super.clone();
  }
  Room room1;
  Room room2;
}
```

## VisualWorks Smalltalk

Object>>shallowCopy
Does a shallowCopy of the receiver

```
Object>>copy
   ^self shallowCopy postCopy

   “Template method for copy”
```

Copy is the primary method for copying an object

Classes override postCopy to do more than shallow copyDoor

```
Smalltalk.CS635 defineClass: #Door
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'room1 room2 '

postCopy
   room1 := room1 copy.
   room2 := room2 copy.
```

## Consequences

- Adding and removing products at run-time
- Specifying new objects by varying values
- Specifying new objects by varying structure
- Reducing subclassing (from factory method)
- Configuring an application with classes dynamically

## Implementation Issues

- Using a prototype manager
- Implementing the Clone operation
- Initializing clones

## Sharable & Flyweight
## Nation Example

Each country, like India or China, has a lot of information

A Nation object for one country may have a lot of data

A program may only have a few references to an India object

Having all references share the same object

- Saves space
- Saves time creating/copying the object
- Keeps all references consistent

## Symbol & Interned Strings

### Smalltalk

Only one instance a symbol with a give character sequence in an image

```
| a b |

a := #cat.
b := ('ca' , 't') asSymbol.
a = b  "True"
a == b   "True – a & b point to same location"
```

Symbols

- Save space
- Make comparing symbols fast
- Make hashing symbols fast

### Java

Compiler tries to use only one instance of a string with a given character sequence

String>>intern() returns a reference to a unique instance of a string

## Text Example

Use objects to represent individual characters of the alphabet

Using objects allows the program to treat characters like any other part of the document - as an object

A character, like “G”, may require a fair amount of information:

- The character type - G not h or y
- Its font
- Its width, height, ascenders, descenders, etc.
- How to display/print the character
- Where it is in the document

Most of this information is the same for all instances of the same letter

So

- Have one G object and
- Have all places that need a G reference the same object

What if there is state information that is different between references?

## Intrinsic State

- Information that is independent from the objects context
- The information that can be shared among many objects

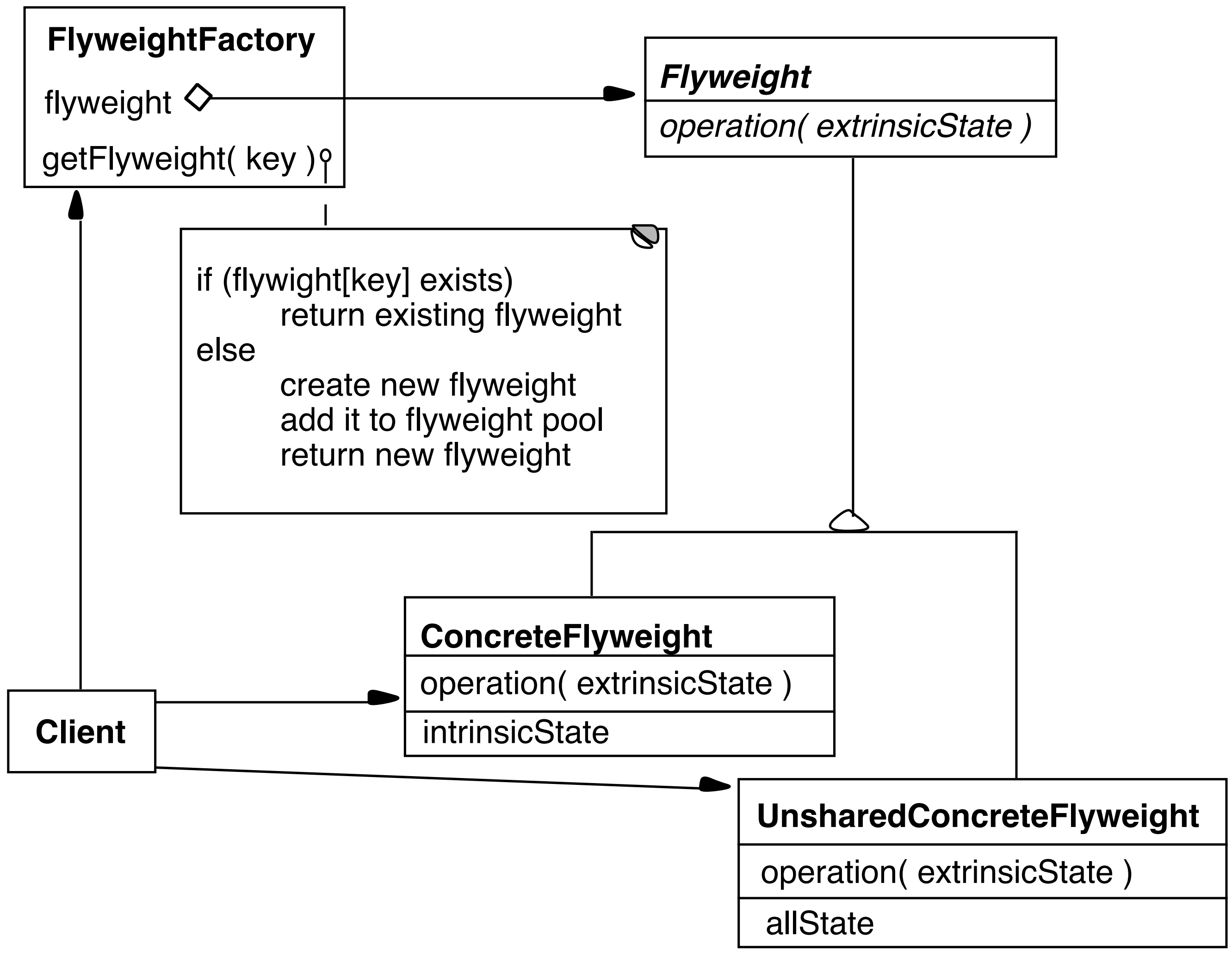
## Extrinsic State

- Information that is dependent on the objects context
- The information that can not be shared among objects

So create one instance of the object and use the same object wherever you need an instance

The one object can store the intrinsic state, but someone else needs to store the extrinsic state for each context of the object

## Structure


FlyweightFactory
flyweight
getFlyweight( key )
Flyweight
operation( extrinsicState )
if (flywight[key] exists)
return existing flyweight
else
create new flyweight
add it to flyweight pool
return new flyweight
ConcreteFlyweight
operation( extrinsicState )
intrinsicState
Client
UnsharedConcreteFlyweight
operation( extrinsicState )
allState

## Applicability

The pattern can be used when all the following are true

- The program uses a large number of objects
- Storage cost are high due to the sheer quantity of objects
- The program does not use object identity

```
MyClass* objectPtrA;
MyClass* objectPtrB;

if ( objectPtrA == objectPtrB ) //testing object identity
```

- Most object state can be made **extrinsic**

  Extrinsic state is data that is stored outside the object

  The extrinsic state is passed to the object as an argument in each method

- Many objects can be replaced by a relatively few shared objects, once the extrinsic state is removed

## Implementation

### Separating state from the flyweight

This is the hard part

Must remove the extrinsic state from the object

Store the extrinsic state elsewhere

  This needs to take up less space for the pattern to work

Each time you use the flyweight you must give it the proper extrinsic state

### Managing Flyweights

Cannot use object identity on flyweights

Need factory to create flyweights, cannot create directly

How do we know when we are done with a flyweight?

## Façade

### Compiler Example

The VisualWorks Smalltalk compiler system has 75 classes

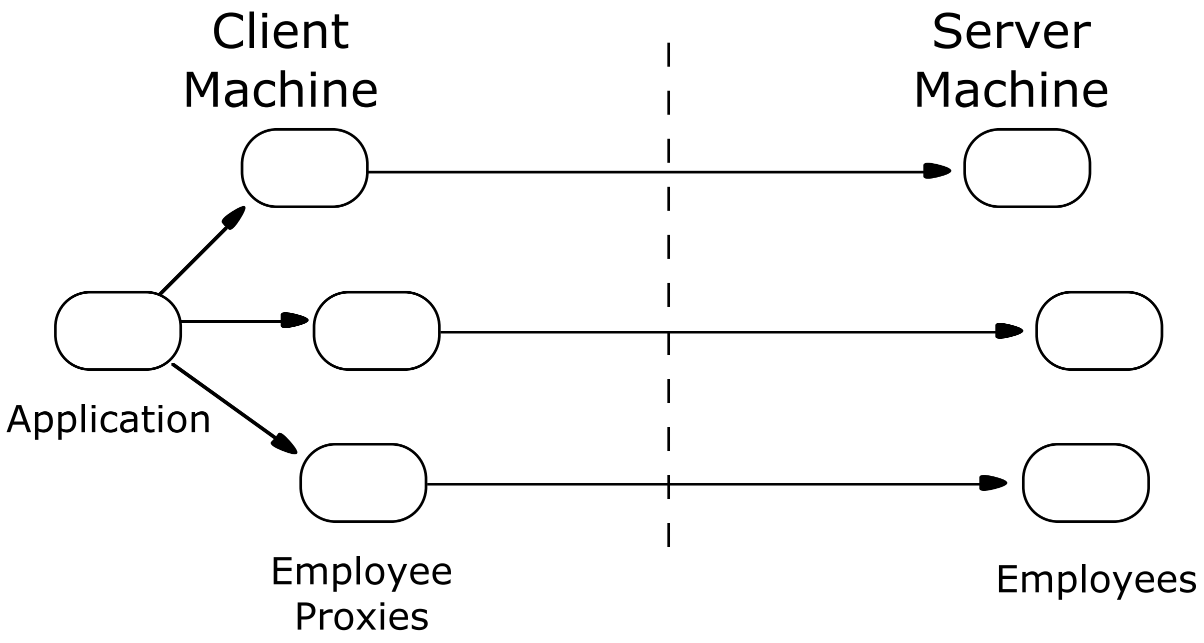
Programmers only use Compiler, which uses the other classes

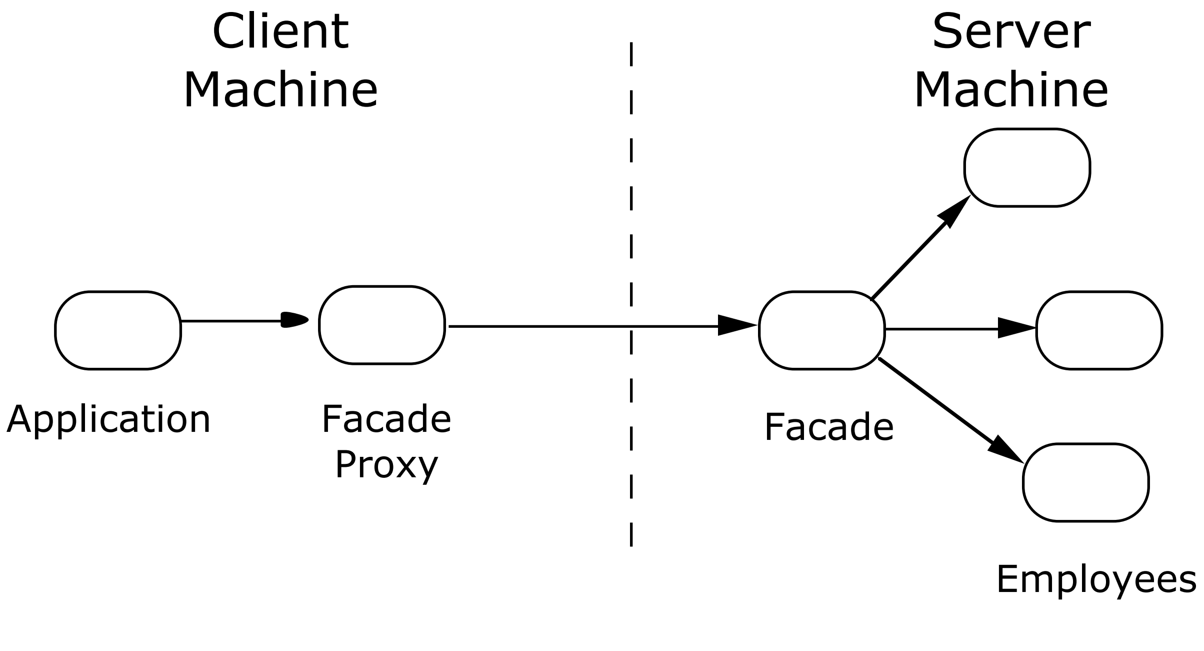
```
Compiler evaluate: '100 factorial'

| method compiler |
method := 'reset
   "Resets the counter to zero"
   count := 0.'.

compiler := Compiler new.
compiler
   parse:method
   in: Counter
   notifying: nil
```

## Distributed Object Systems





## Subsystems

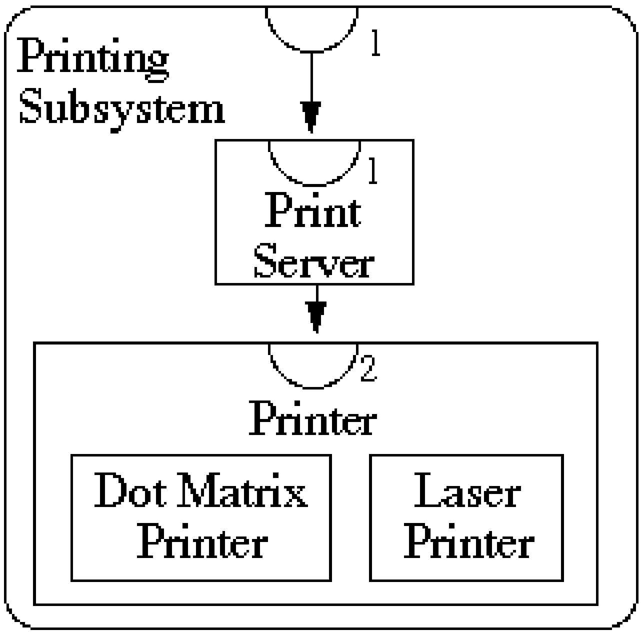
Subsystems are groups of classes, or groups of classes and other subsystems, that collaborate among themselves to support a set of contracts

There is no conceptual difference between the responsibilities of a class and a subsystem of classes

The difference between a class and subsystem of classes is a matter of scale

A subsystem should be a good abstraction

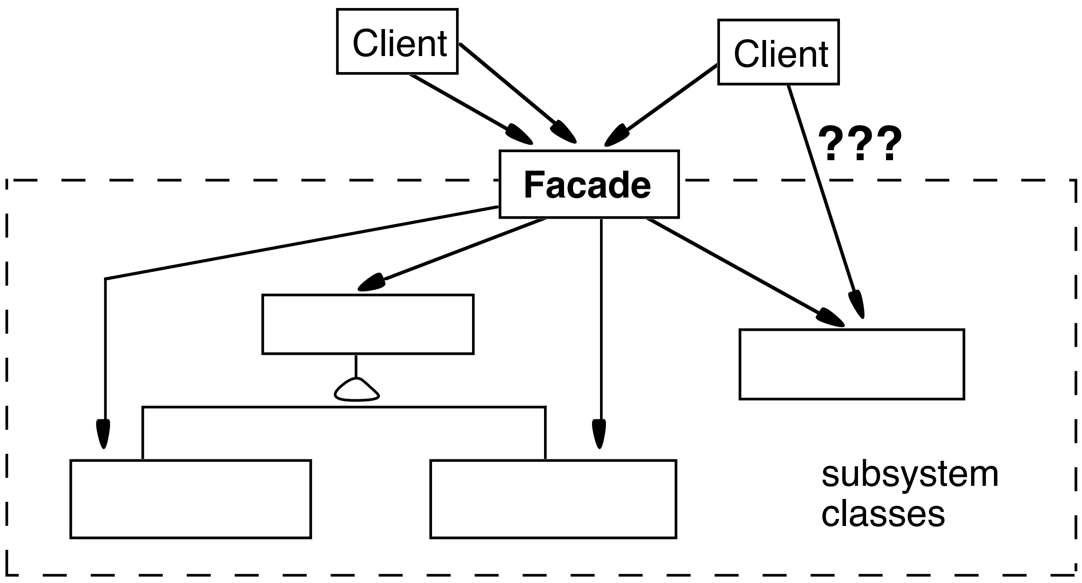
There should be as little communication between different subsystems as possible



## The Facade Pattern - Basic Idea

Create a class that is the interface to the subsystem

Clients interface with the Facade class to deal with the subsystem



## Consequences of Facade Pattern

It hides the implementation of the subsystem from clients

It promotes weak coupling between the subsystems and its clients

It does not prevent clients from using subsystem classes directly, should it?

Facade does not add new functionality to the subsystem