# CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2016
Doc 13 Facade, Prototype, Abstract Factory
April 5, 2016

# Facade

# Size

| Item | Source Lines of Code (Millions) |
|---|---|
| F-22 Raptor US jet fighter | 1.7 |
| Boeing 787 | 6.5 |
| S-class Mercedes-Benz radio & navigation system | 20 |
| Mac OS 10.4 | 86 |
| Premium class automobile | ~100 |
| Debian 4.0 | 283 |

Design Patterns text  contains under 8,000 lines



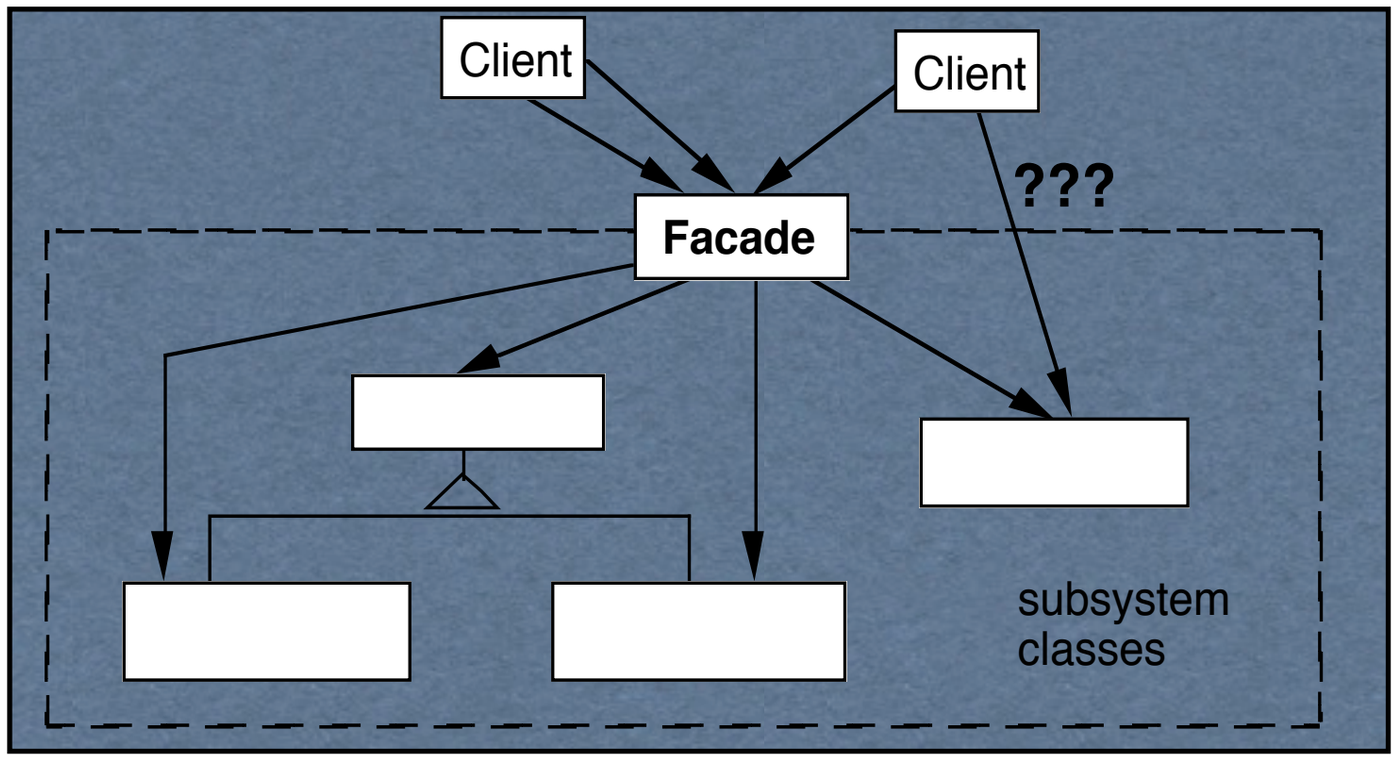


Reference: This Car Runs on Code, http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code
Source lIne s of code, http://en.wikipedia.org/wiki/Source_lines_of_code

# The Facade Pattern

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

# Public versus Private Subsystem classes

Some classes of a subsystem are

- public
  - facade
- private

# Compiler Example

The VisualWorks Smalltalk compiler system has 75 classes

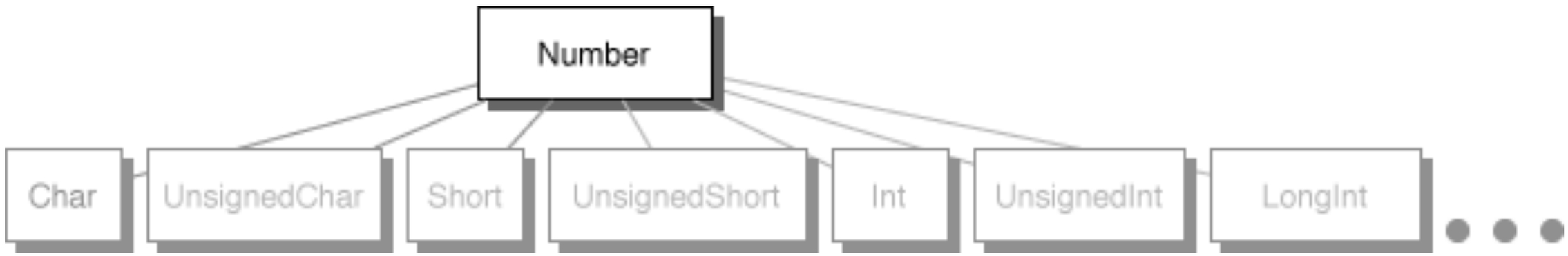
Programmers only use Compiler, which uses the other classes

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
```

# Objective-C Class Clusters & Facade



# Prototype

# Prototype

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.

# Insurance Example

Insurance agents start with a standard policy and customize it

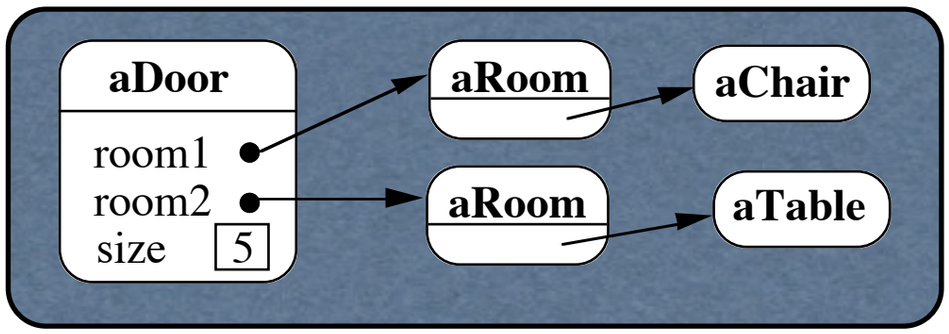
Two basic strategies:

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

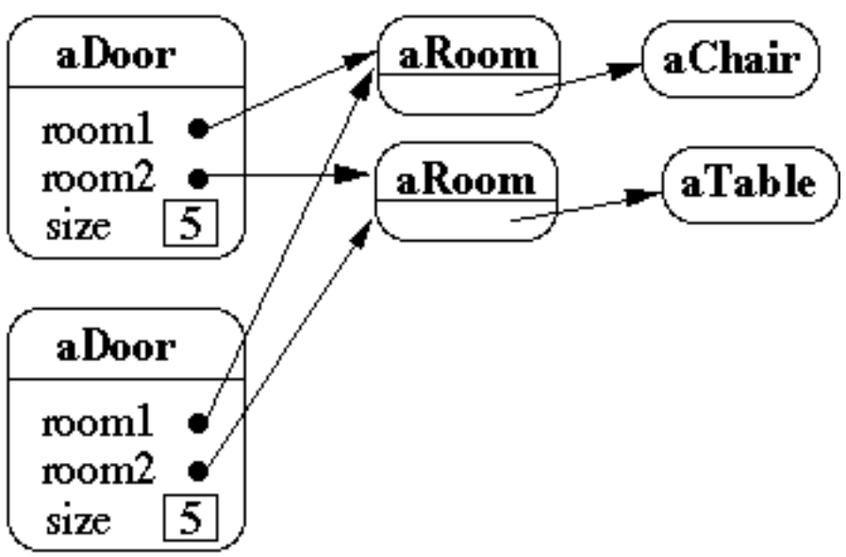
# Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

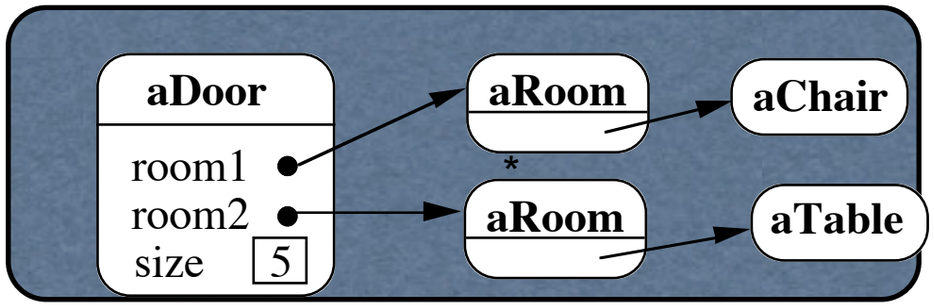



Shallow Copy

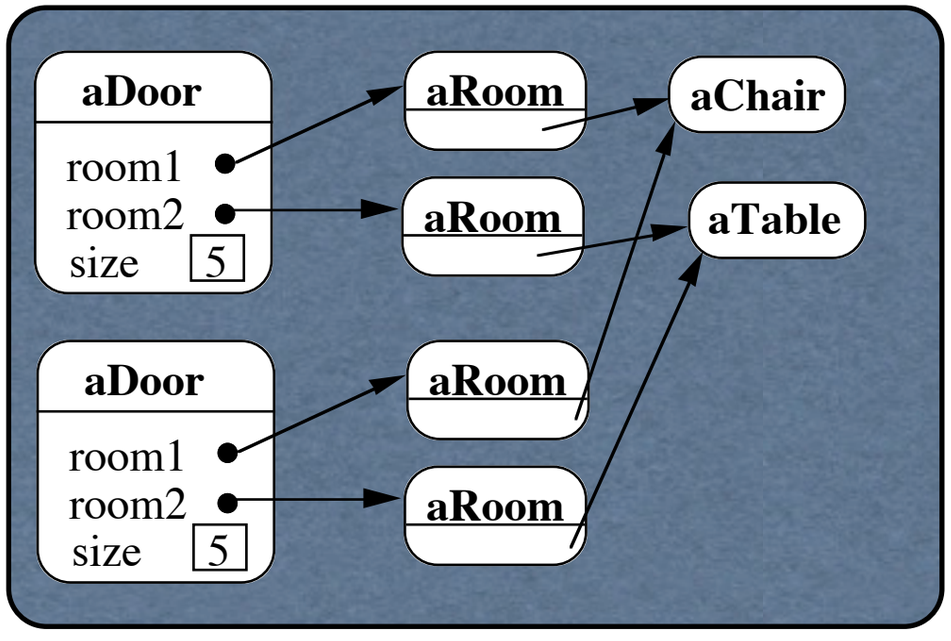


# Shallow Copy Verse Deep Copy

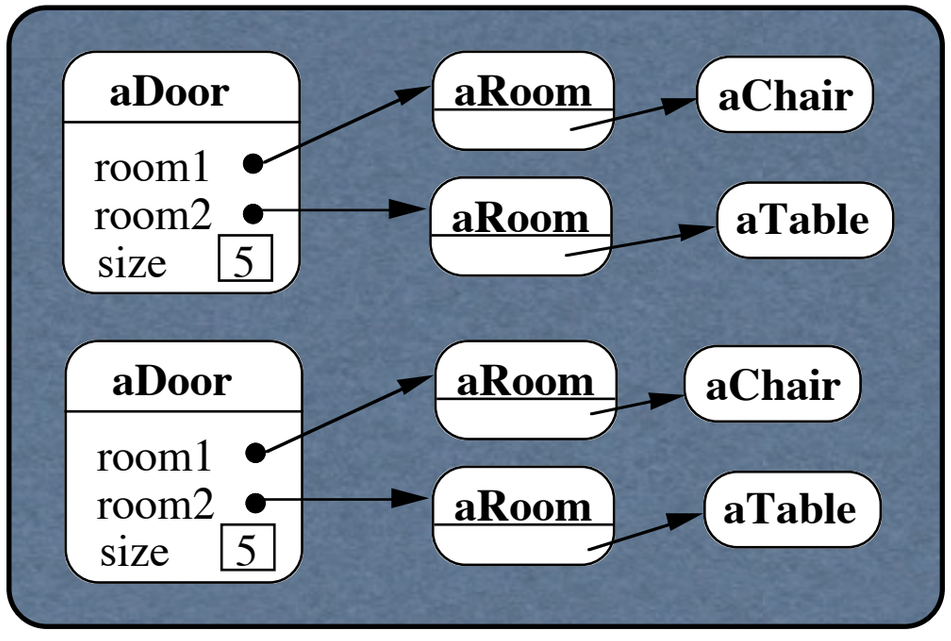
Original Objects




Deep Copy




Deeper Copy



# Cloning Issues - 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 );
        }
```

# Cloning Issues - Java Clone

**Shallow Copy**

```
class Door implements Cloneable {
        private Room room1;
        private Room room2;

        public Object clone() throws  CloneNotSupportedException {
                return super.clone();
        }
}
```

**Deep Copy**

```
public class Door implements Cloneable {
        private Room room1;
        private Room room2;

        public Object clone() throws CloneNotSupportedException {
                Door thisCloned =(Door) super.clone();
                thisCloned.room1 = (Room)room1.clone();
                thisCloned.room2 = (Room)room2.clone();
                return thisCloned;
        }
}
```

# Prototype-based Languages

No classes

Behaviour reuse (inheritance)

- Cloning existing objects which serve as prototypes

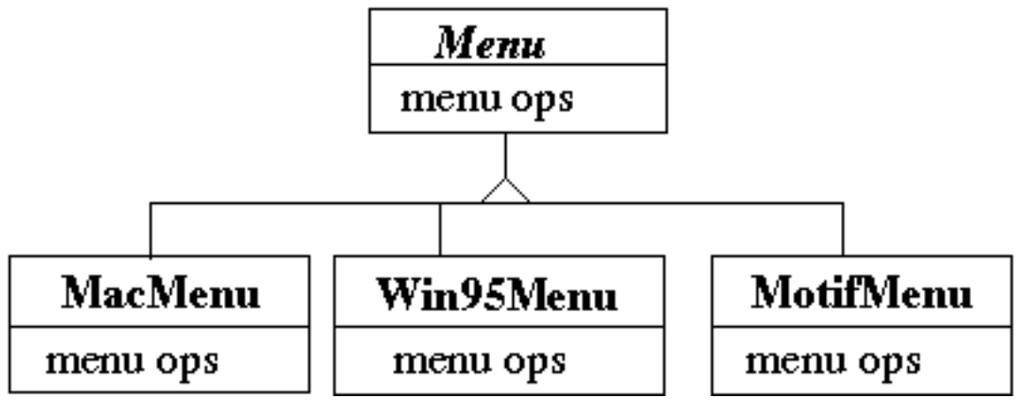
Some Prototype-based languages

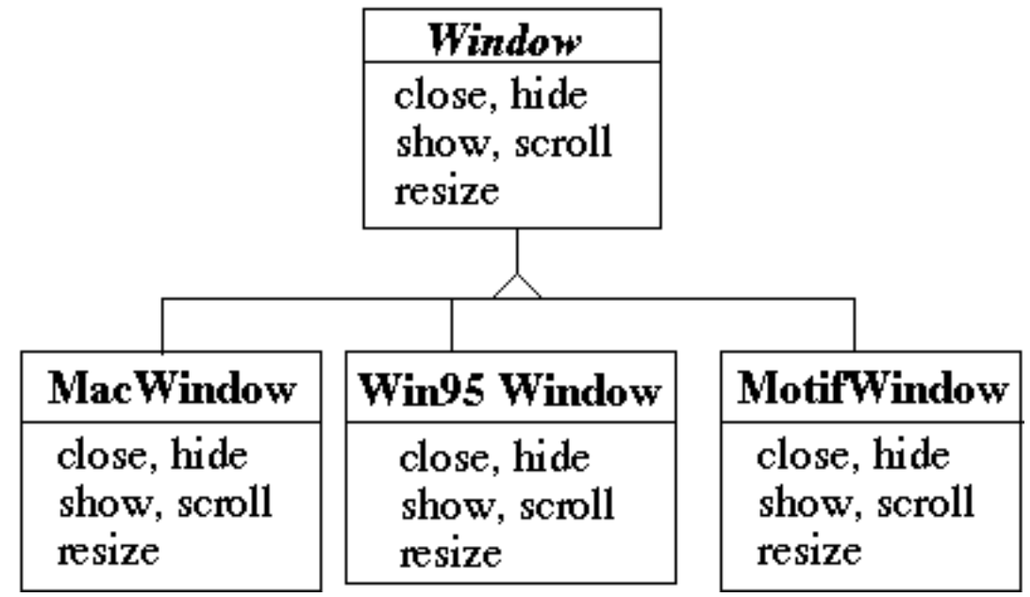
- Self
- JavaScript
- Squeak (eToys)
- Perl with Class::Prototyped module

# Abstract Factory

# Abstract Factory

Write a cross platform window toolkit



# Bad Code Dependencies

```
public void installDisneyMenu()
    {
    Menu disney = new MacMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
    }
```

# Use Abstract Factory

```
abstract class WidgetFactory {
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
    }
```

```
class MacWidgetFactory extends WidgetFactory {
    public Window createWindow()
        { code to create a mac window }

    public Menu createMenu()
        { code to create a mac Menu }

    public Button createButton()
        { code to create a mac button }
}

class Win95WidgetFactory extends WidgetFactory {
    public Window createWindow()
        { code to create a Win95 window }

    public Menu createMenu()
        { code to create a Win95 Menu }

    public Button createButton()
        { code to create a Win95 button }
}
```

# Use one Factory per Application

```
public void installDisneyMenu(WidgetFactory myFactory)
    {
    Menu disney = myFactory.createMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
    }
```

# Abstract Factory

Encapsulate a group of individual factories that have a common theme

Separates the details of implementation of a set of objects
from its general usage

# How Do Abstract Factories create Things?

# Use Subclass Factory Method

```
abstract class WidgetFactory
    {
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
    }

class MacWidgetFactory extends WidgetFactory
    {
    public Window createWindow()
        { return new MacWidow() }

    public Menu createMenu()
        { return new MacMenu() }

    public Button createButton()
        { return new MacButton()  }
    }
```

# Use Widget Factory Method

```
abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public Window createWindow()
        { return windowFactory.createWindow() }

    public Menu createMenu();
        { return menuFactory.createMenu() }

    public Button createButton()
        { return buttonFactory.createMenu() }
}
```

```
class MacWidgetFactory extends WidgetFactory {
    public MacWidgetFactory() {
        windowFactory = new MacWindow();
        menuFactory = new MacMenu();
        buttonFactory = new MacButton();
    }
}

class MacWindow extends Window {
    public Window createWindow() { blah }
    etc.
```

# Why Widget Factory Method?

```
abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public Window createWindow()
        { return windowFactory.createWindow() }

    public Window createWindow( Rectangle size)
        { return windowFactory.createWindow( size ) }

    public Window createWindow( Rectangle size, String title)
        { return windowFactory.createWindow( size, title) }

    public Window createFancyWindow()
        { return windowFactory.createFancyWindow() }

    public Window createPlainWindow()
        { return windowFactory.createPlainWindow() }
```

Multiple ways to create Widget

# Use Prototype

```
class WidgetFactory{
        private Window windowPrototype;
        private Menu menuPrototype;
        private Button buttonPrototype;

        public WidgetFactory( Window windowPrototype,
                                        Menu menuPrototype,
                                        Button buttonPrototype)
                {
                this.windowPrototype = windowPrototype;
                this.menuPrototype = menuPrototype;
                this.buttonPrototype = buttonPrototype;
                }

        public Window createWindow()
                { return windowPrototype.createWindow() }

        public Window createWindow( Rectangle size)
                { return windowPrototype.createWindow( size ) }

        public Window                ()
                { return menuPrototype.createMenu() }
        etc.
```

# How to prevent Cheating?

```
public void installDisneyMenu(WidgetFactory myFactory)
    {
    // We ship next week, I can't get the stupid generic Menu
    // to do the fancy Mac menu stuff
    // Windows version won't ship for 6 months
    // Will fix this later

    MacMenu disney = (MacMenu) myFactory.createMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addMacGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
    }
```