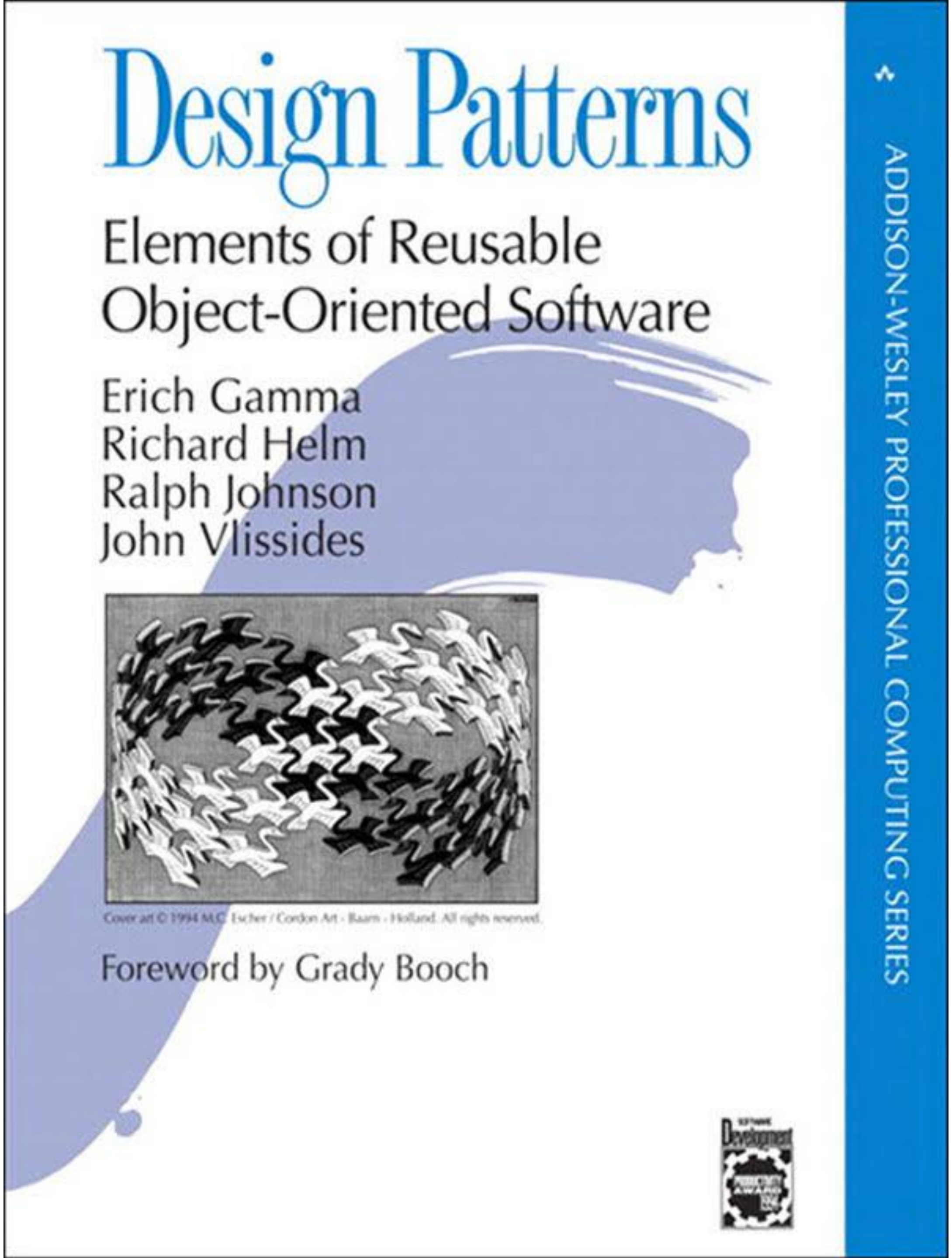
DESIGN PATTERNS

FELIZ GOUVEIA

UFP

WHAT ARE THEY?

- “In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.”
- *Design Patterns: Elements of Reusable Object-Oriented Software*



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

PATTERN TYPES

- Creational design patterns
  - About class instantiation
- Structural design patterns
  - About class composition
- Behavioral design patterns
  - About class communication

SINGLETON PATTERN

- For situations where we know we only need one instance:
  - Filesystem
  - Printer controller
  - UI controller
  - Some services (database pool)
- Single point of access to the object
- Possibility of creating more instances in the future

SINGLETON PATTERN

```
public class ClassicSingleton {
        private static ClassicSingleton instance = null;
        protected ClassicSingleton() { // Exists only to defeat
                                    instantiation. }
        public static ClassicSingleton getInstance() {
                if(instance == null){
                        instance = new ClassicSingleton();
                }
                return instance;
        }
}
```

SINGLETON PATTERN (1)

Prevent multi-threading issues:

```
public static Singleton getInstance() {
        if(singleton == null) {
                synchronized(Singleton.class) {
                        singleton = new Singleton();
                }
        }
        return singleton;
}
```

FAÇADE PATTERN

“Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.”

- Hides complexity and detail
- Heavy work is done by the façade’s code, not by the developer
- Less business objects need to be exposed, which increases flexibility

FAÇADE PATTERN (1)

Example from the Java Pet Store. A façade centralizes services common to all shopping actions

```
public interface ShoppingClientFacadeLocal extends
EJBLocalObject {
        public ShoppingCartLocal getShoppingCart();
        public void setUserId(String userId);
        public String getUserId();
        public CustomerLocal getCustomer() throws
                                        FinderException;
        public CustomerLocal createCustomer(String userId); }
```

FACTORY PATTERN

- A superclass specifies all standard and generic behavior and then delegates the creation details to subclasses that are supplied by the client.
- Lets a class defer instantiation to subclasses

FACTORY PATTERN (1)

```
public class SimpleFactory {
            public Toy createToy(String toyName) {
                    if (“car”.equals(toyName)){
                         return new Car();
                    } else if (“helicopter”.equals(toyName)){
                        return new Helicopter();
                    else
                         return null;
            }
}
```

FACTORY PATTERN (2)

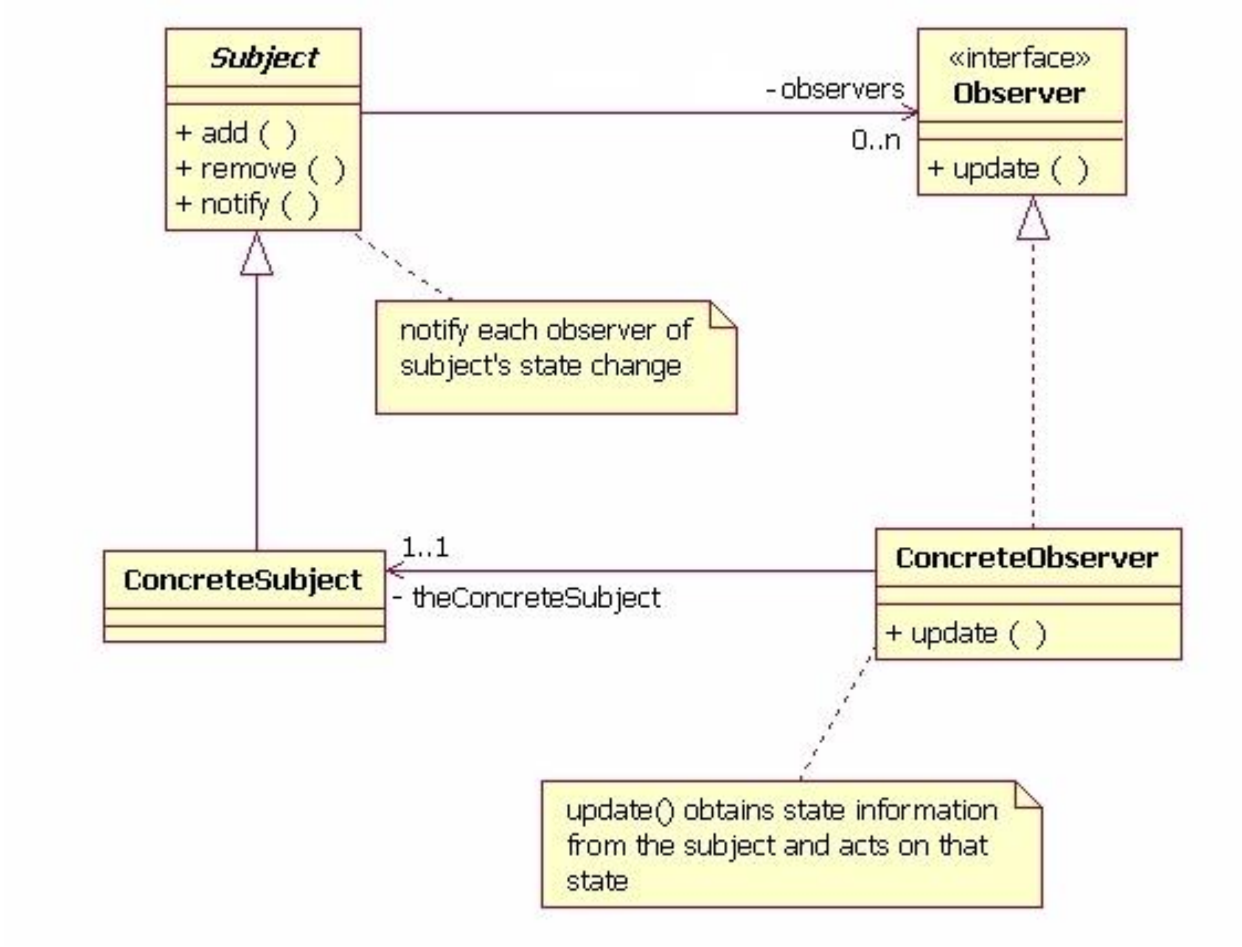
```
public class ToysFactory {
    private simpleFactory;
    public ToysFactory(SimpleFactory simpleFactory) {
        this.simpleFactory = simpleFactory;
    }

    public Toy produceToy(String toyName) {
        Toy toy = simpleFactory.createToy(toyName);
        toy.build();
        toy.package();
        return toy;
    }
}
```

OBSERVER PATTERN

“Define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

- There are many observers as needed
- Some objects depend on the state of other objects and would like to be notified about any changes to that state
- Classic problem in UI. When user clicks, several objects should be notified

OBSERVER PATTERN (1)



OBSERVER PATTERN (2)

- The subject keeps a list of observers that registered to be notified
- Subjects implement a notify() method
- Observers have their update() meyhod called by the subject’s notify()

OBJECT POOL PATTERN

- In some situations objects can be reused
- Object creation can be time-consuming
- An object pool caches objects so that they can be reused when needed
  - Example: database connection pools

OBJECT POOL PATTERN

- ObjectPool has an internal array of objects
- ObjectPool has acquire() and release() methods
- ObjectPool is a singleton

PROTOTYPE PATTERN

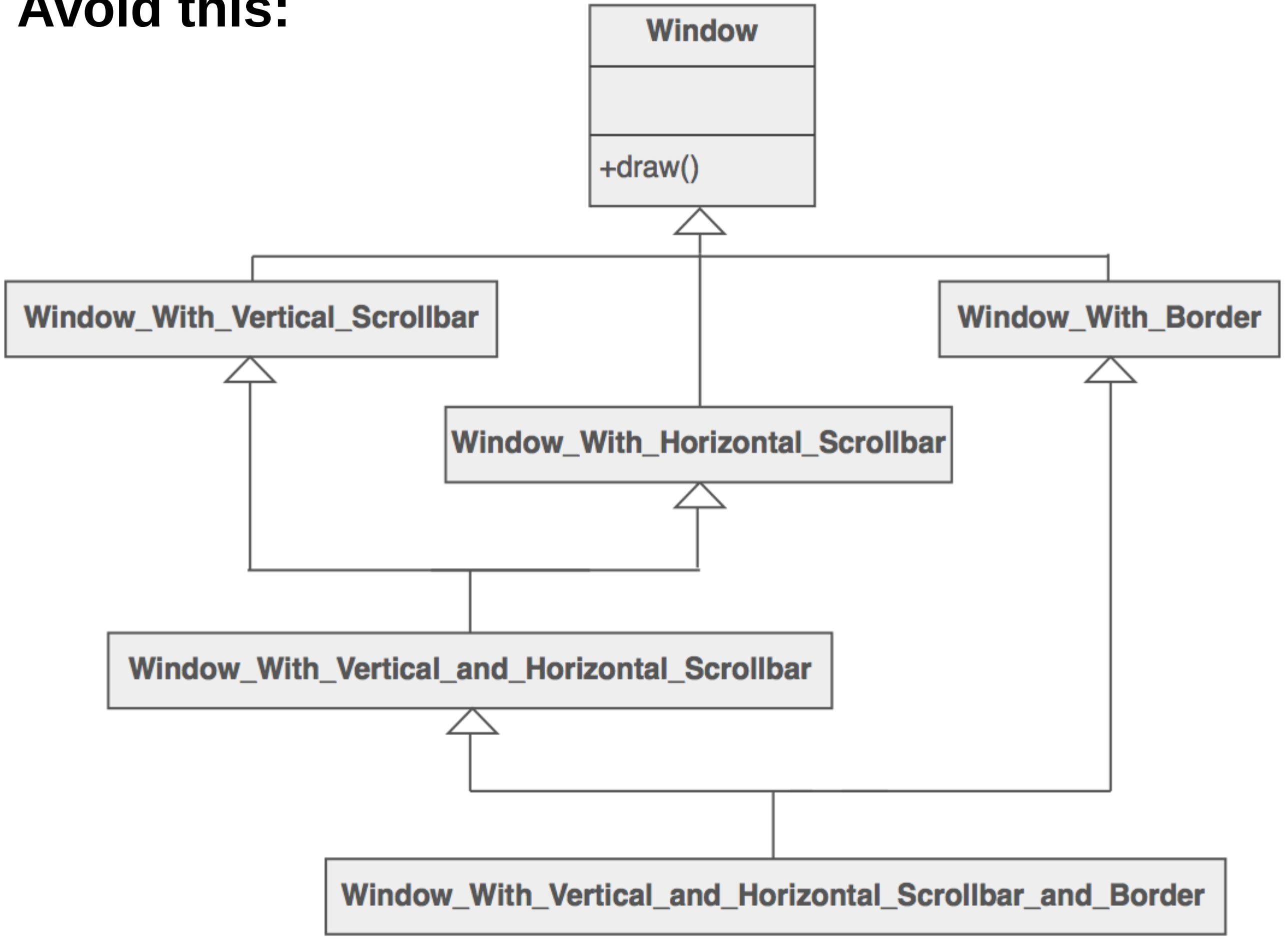
- Maintain a list of prototypes
- Has a clone() method
- The client calls the clone() method to get a new object

DECORATOR PATTERN

- Add aditional behavior or structure to an object at run time
- The client embelishes an object by wrappint it
- Inheritance does not work because it applies to the class (and to all instances) and it is static

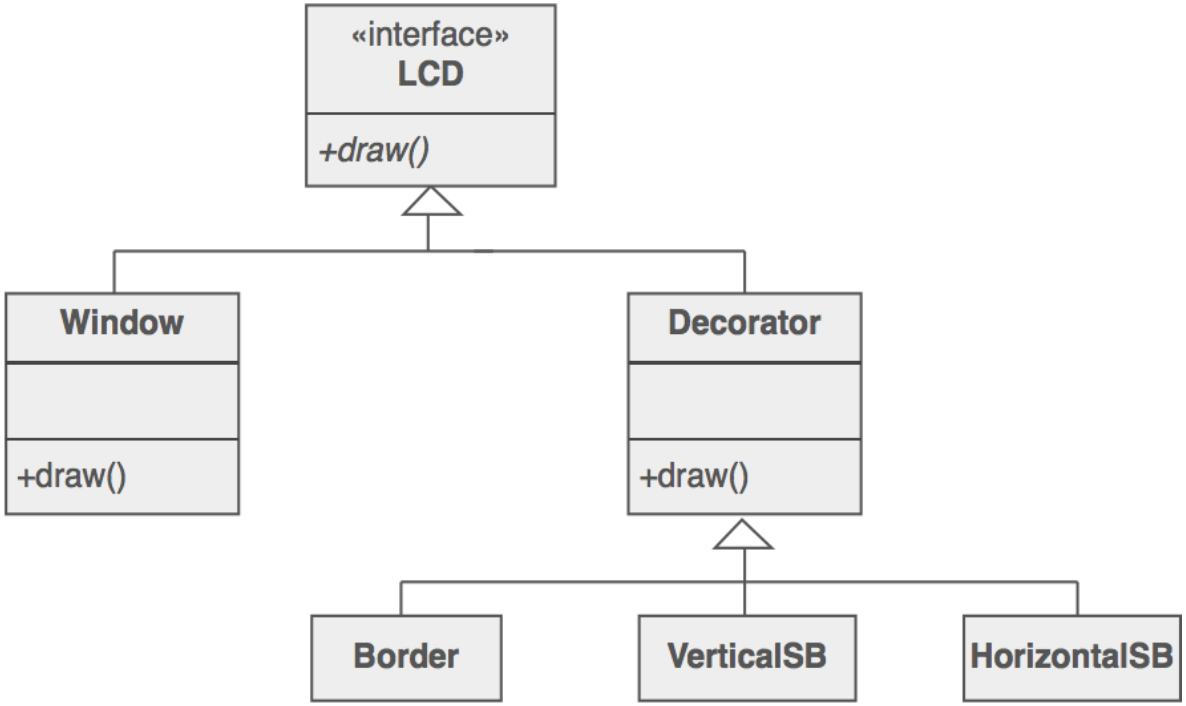
DECORATOR PATTERN

Avoid this:


Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Border
Window_With_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border

DECORATOR PATTERN

```
Widget* aWidget = new BorderDecorator(new
        HorizontalScrollBarDecorator(new
            VerticalScrollBarDecorator( new Window( 80, 24 ))));

aWidget->draw();
```


«interface»
LCD
+draw()
Window
+draw()
Decorator
+draw()
Border
VerticalSB
HorizontalSB