# Object Oriented (OO) Design

Yufeng Wu

For CSE 3150 class
Spring 2020

# Object Oriented (OO) Design

- Compare OO design with structural design
  - Encapsulation (again): information hiding.
- Encapsulation, polymorphism, abstraction.
- Class: exports operations (procedures) to manipulate instance objects
  - often called *methods*
- Instance objects accessible via references
- Inheritance: class B may specialize class A
  - B inherits from A (B is a subclass of A)
  - A generalizes B (A is a superclass of B)

# Objects and classes

- Objects package data and procedures (*methods*) that operate on the data.
  - Client -> Send a request (message) ->Server performs an operation
  - Request is the **only** way to change object's internal data: encapsulation
- OO Design: decompose into a set of manageable and interacting objects
  - encapsulation, granularity, flexibility, performance, evolution, reusability, and on and on

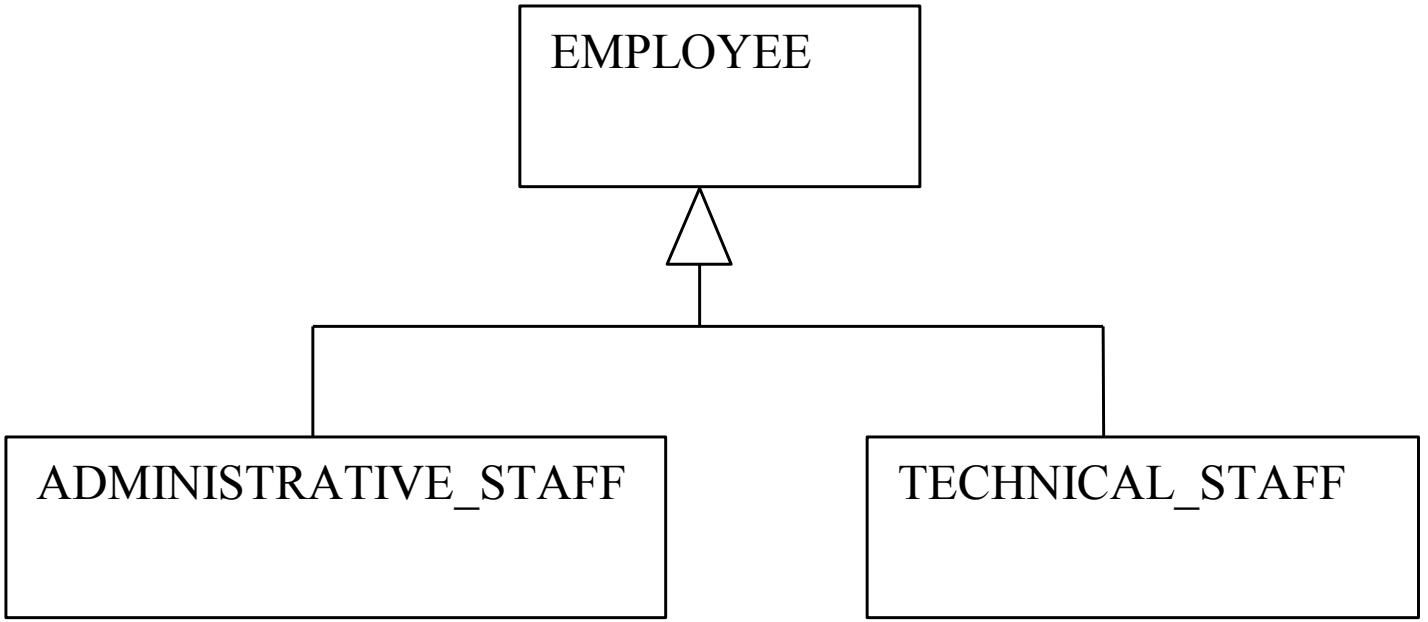
# Object Interface

- Signature of operation/method: name, parameters, and return value
- Object interface: the set of all signatures of its operations
- **Type**: name of an interface
  - Object *a* has type A ⇔ *a* accepts all requests in A
  - One object with multiple types (Java and C++)
  - Widely different objects with same type
  - Subtype: inherit another interface

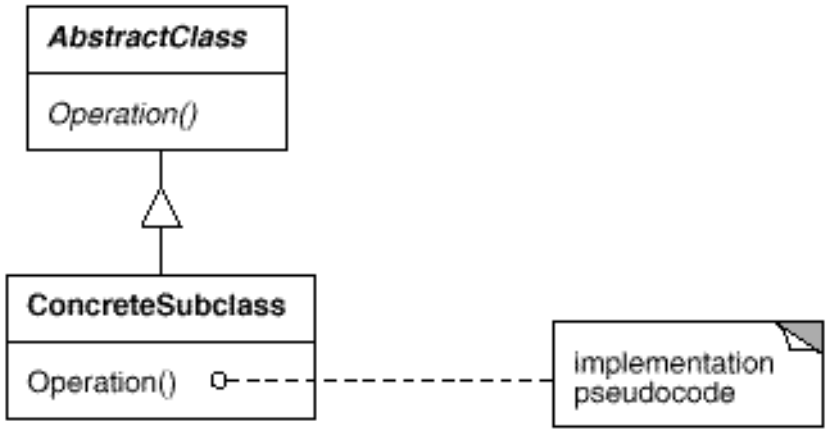
# Inheritance

- A way of building software incrementally
- A subclass defines a subtype
  - subtype is *substitutable* for parent type
- Polymorphism
  - a variable referring to type A can refer to an object of type B if B is a subclass of A
- Dynamic binding
  - Which method invoked through a reference?
  - Substitute objects with same interface at run-time
  - Decouple objects, which are variable at run-time
- UML (Unified Modeling Language): a widely adopted standard notation for representing OO designs

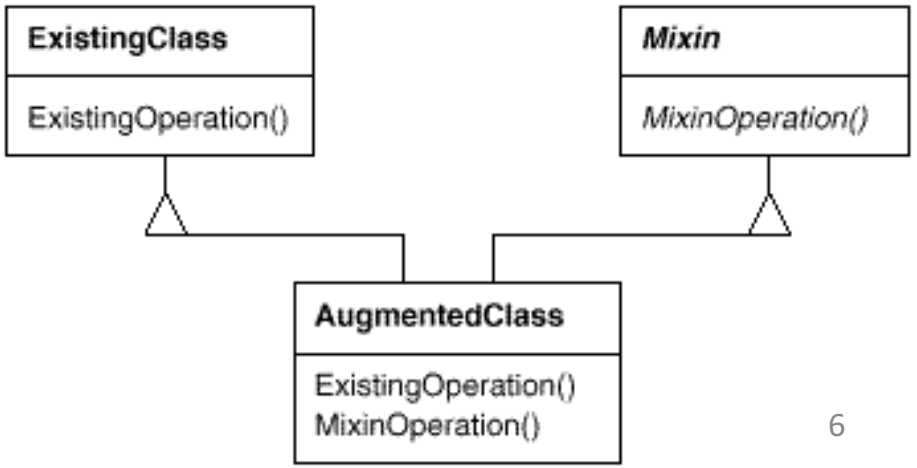
# UML class diagram for inheritance




Implement a method

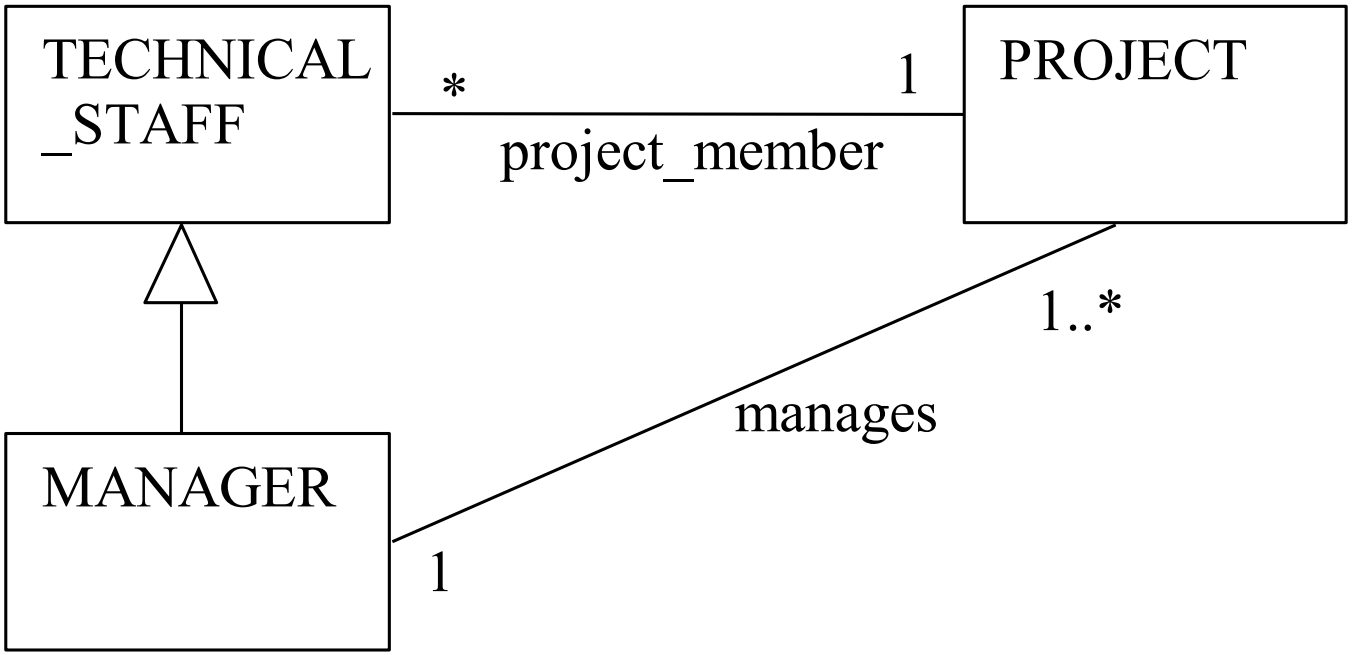



Multiple base classes



# UML association

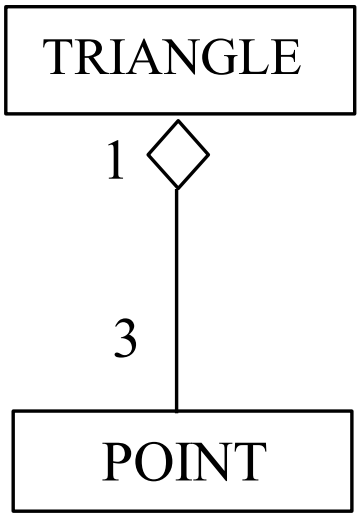
- Associations are relations that the implementation is required to support
- Can have multiplicity constraints



# Aggregation

- Defines a PART_OF relation

  Differs from IS_COMPOSED_OF




Question: what is the difference between association and aggregation?

# OO Principles

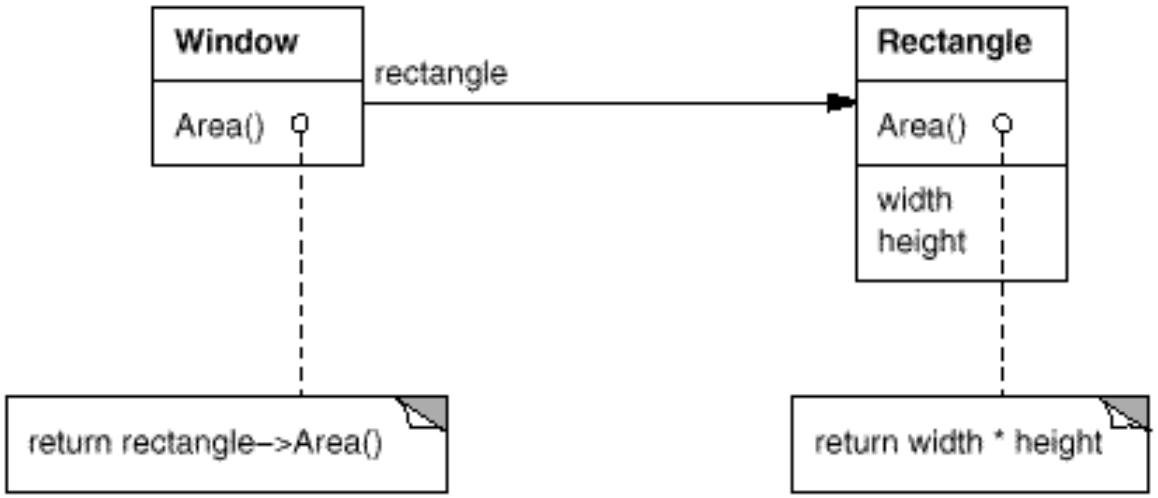
- Class for implementation, type for interface
- Class inheritance (*implementation* reuse)
  - e.g. Java extends a base class
  - Code sharing and reuse: extending and slightly changing parent class.
- Interface inheritance
  - e.g. Java implements an interface
  - Substitute in place of another object.
  - Client unaware of server's implementation (e.g. which class), only interface matters
- **Principle #1**: Program to an interface, not implementation!

# Reuse: inheritance vs. composition

- Inheritance: white-box reuse
- Composition: assemble objects to get complex functionality. Black-box reuse
- Example: Window and Rectangle
- **Discussion**: compare inheritance and composition.
- **Principle #2**: Favor object composition over class inheritance.
  - Think more of composition, while inheritance is still useful.

# Delegation

- A receiving object processes a request by calling a delegate.
  - Class inheritance: subclass defers to parent class.
- Pros and cons?



# More on OO Design

- Parameterized types or generics (templates): Not exactly OO.
  - Another way of reuse.
  - Can not change at run-time.
- Run-time and compile time structures.
- Aggregation and association: hard to differ sometimes statically, may need run-time info.
- Run-time structure: hard to determine.

# Design for change

- What can cause re-design?
  - Create an object by specifying a particular class.
  - Specify a particular operation (?)
  - Dependence on hardware/software platform.
  - Dependence on object representation and implementation.
  - Algorithmic dependence.
  - Tight coupling.
  - Subclassing.
  - Inability to alter classes conveniently.

# OO Design vs. procedural design

- OO design
  - Pros: flexible, reusable, easier to maintain through decoupling.
  - Cons: more complex, less efficient
- Procedural design: better if software needs little change.
- How to learn OO design? Learn through experience, from expert.
- Now case studies.

# Case study: Design of button-controlled table lamp

- Problem: design software for a button controlling a light bulb: push once, turn on, and push it again, turn off.
  - A simple client-server problem
- Now think about an OO design.
  - What classes?
  - How do the classes interact?

# Simple client-server design

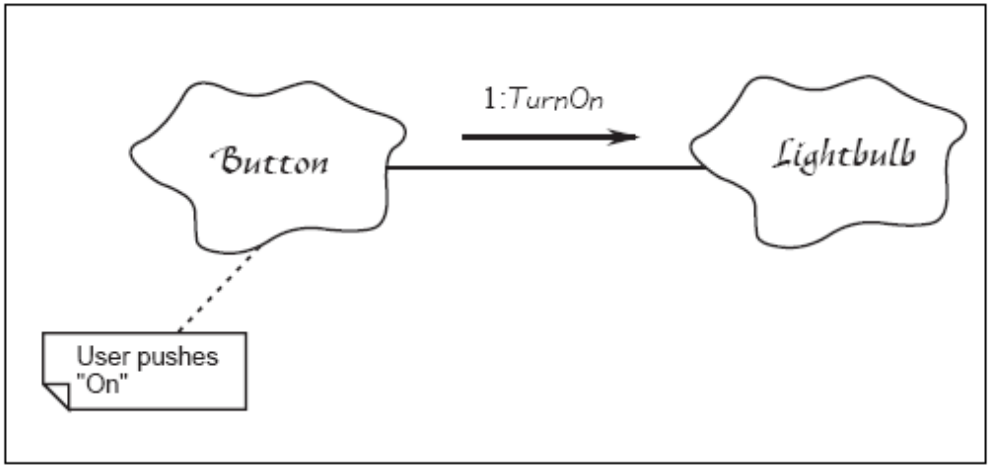



**FIGURE 1.**
User pushes button “on”

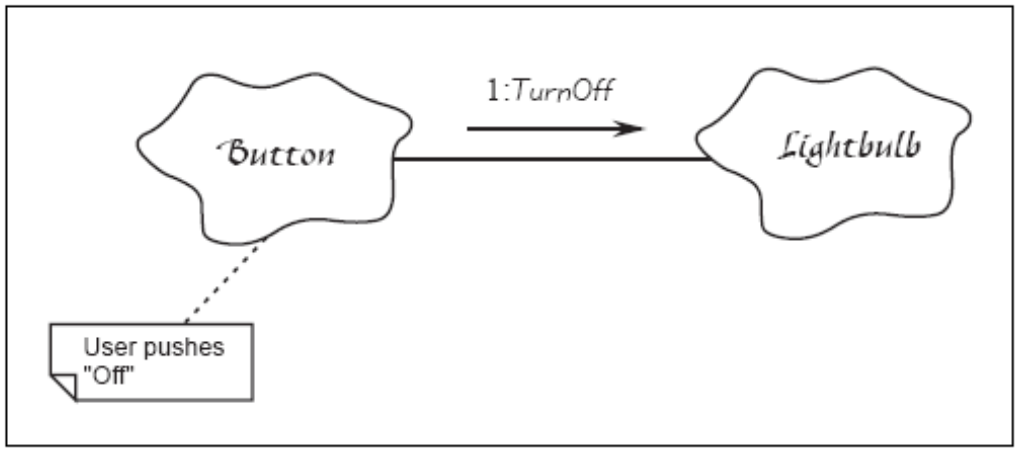
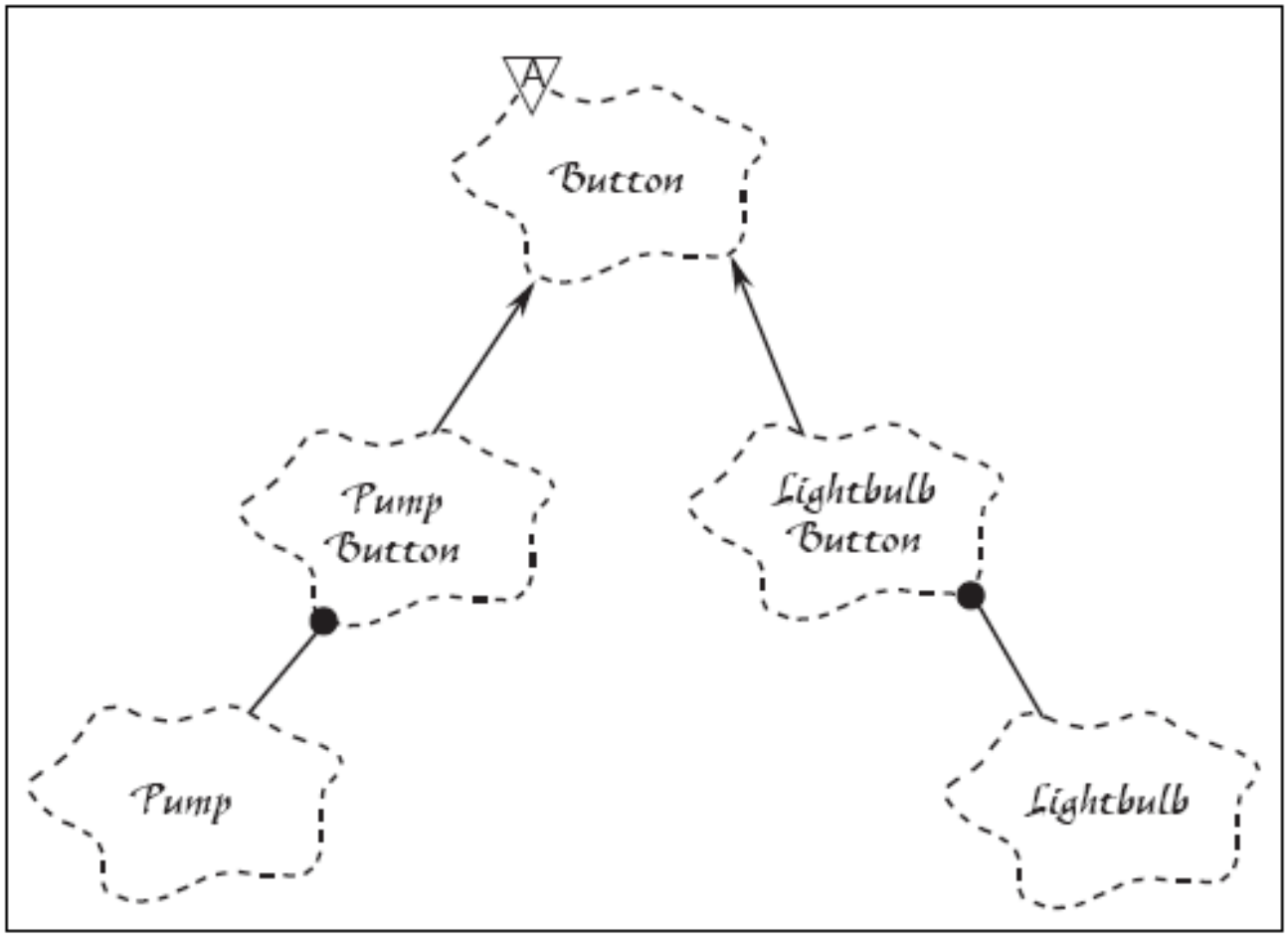
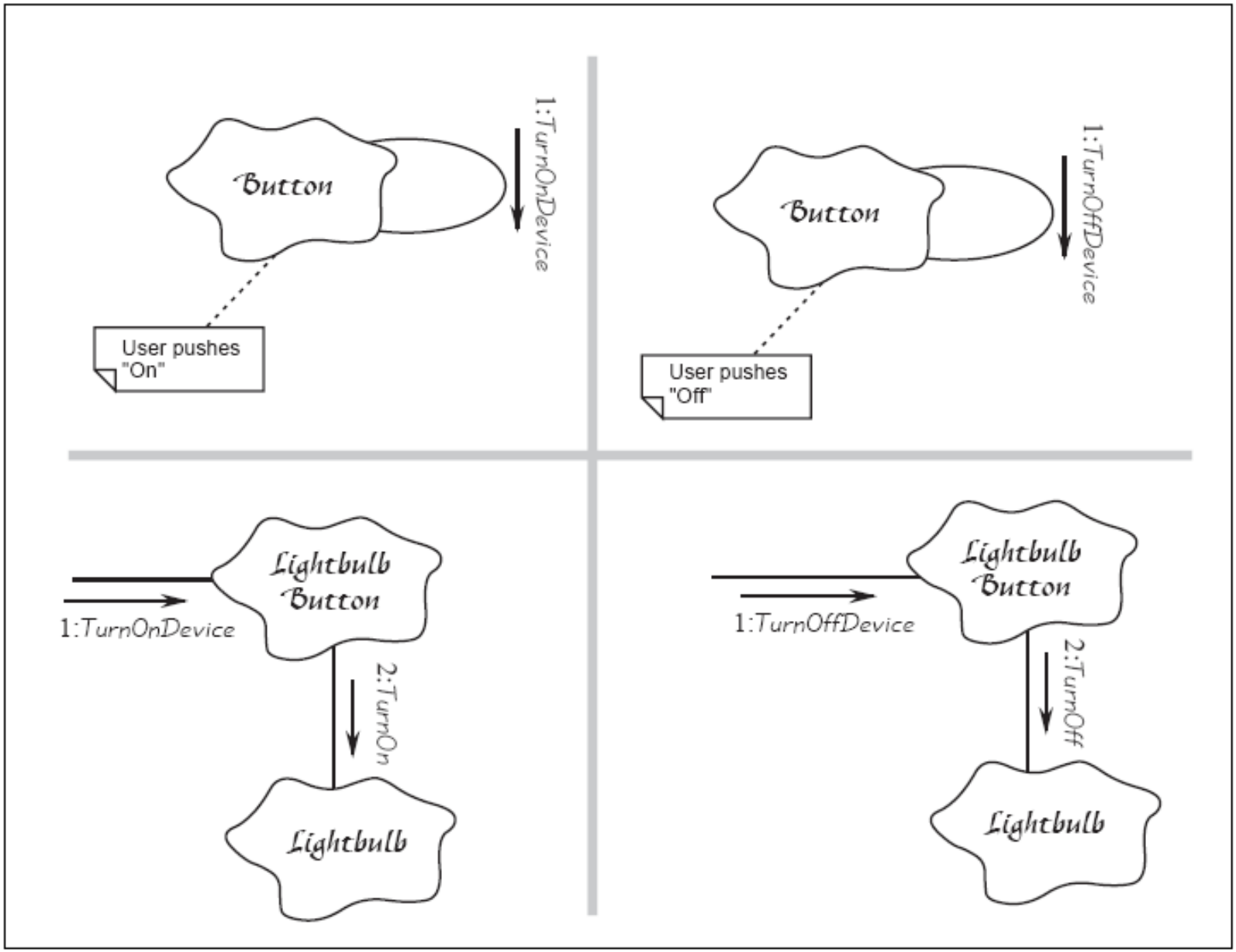



**FIGURE 2.**
User Pushes Button “Off”

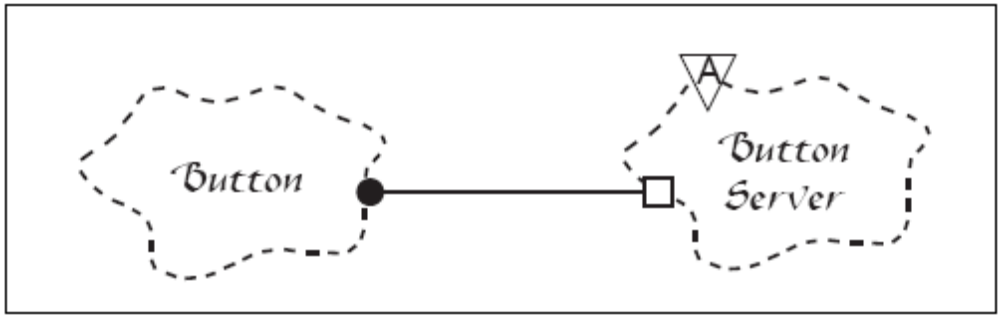
# Decoupling button and light bulb

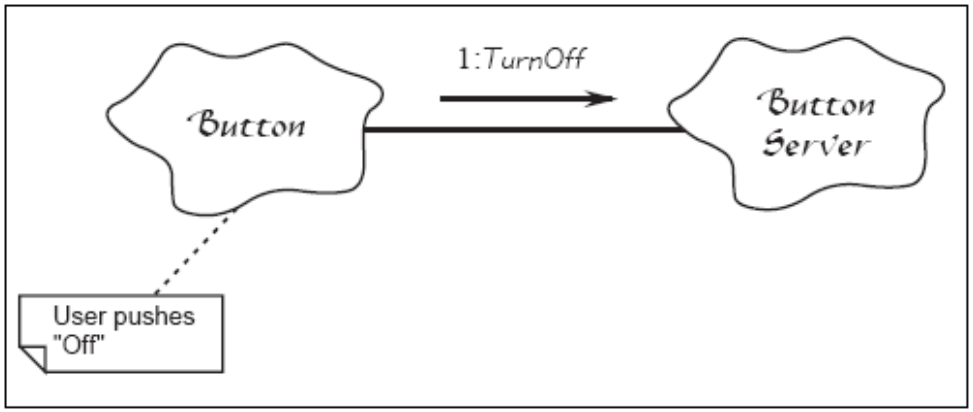


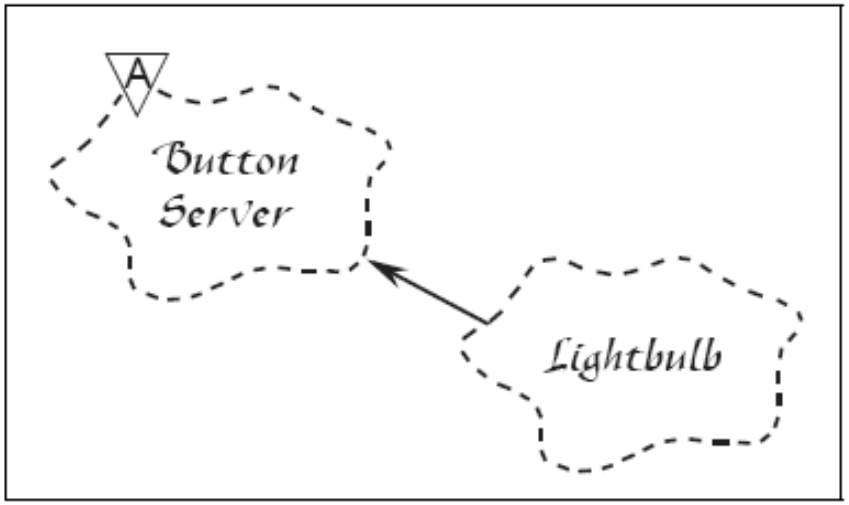
# Dynamic Behavior

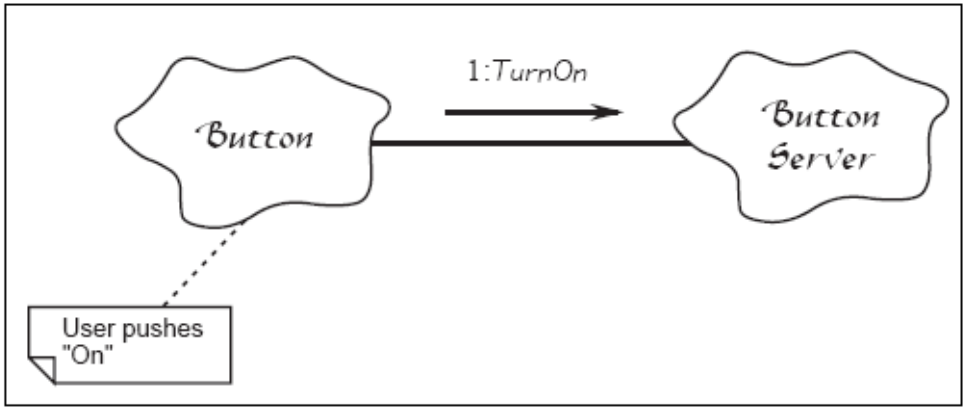


# Another decoupling way



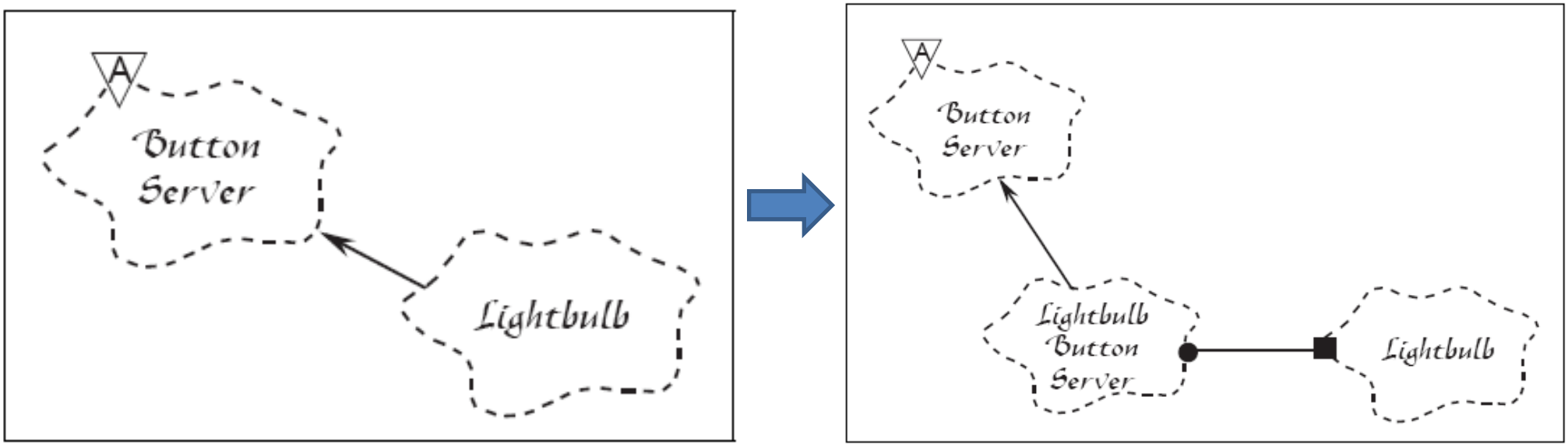



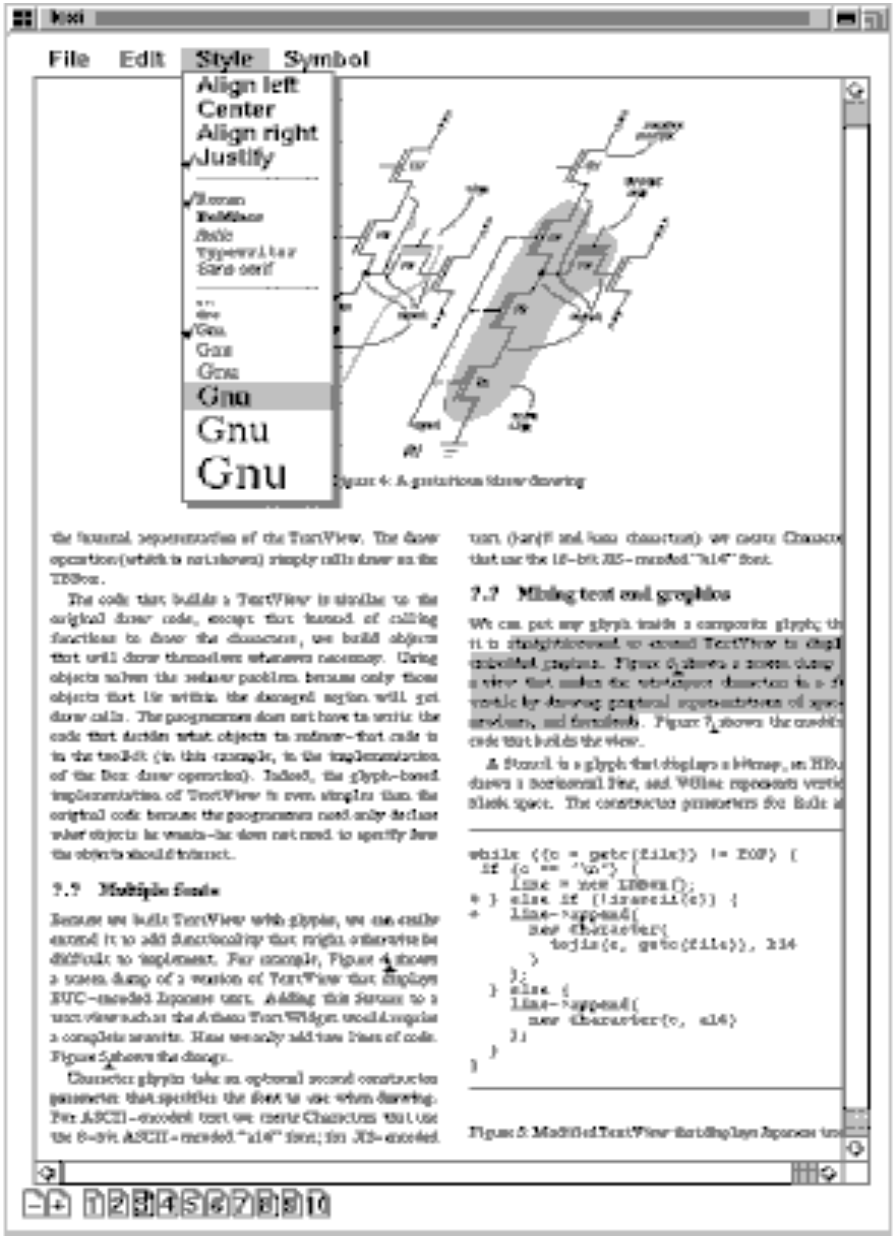

- Any issue with this design? Can this be further improved?

# Improvement




- What is the motivation of this design?
- Comparing the two different approaches.
- Is this really practical? Is this overkill?

# A more complex case study: Design a WYSIWYG document editor



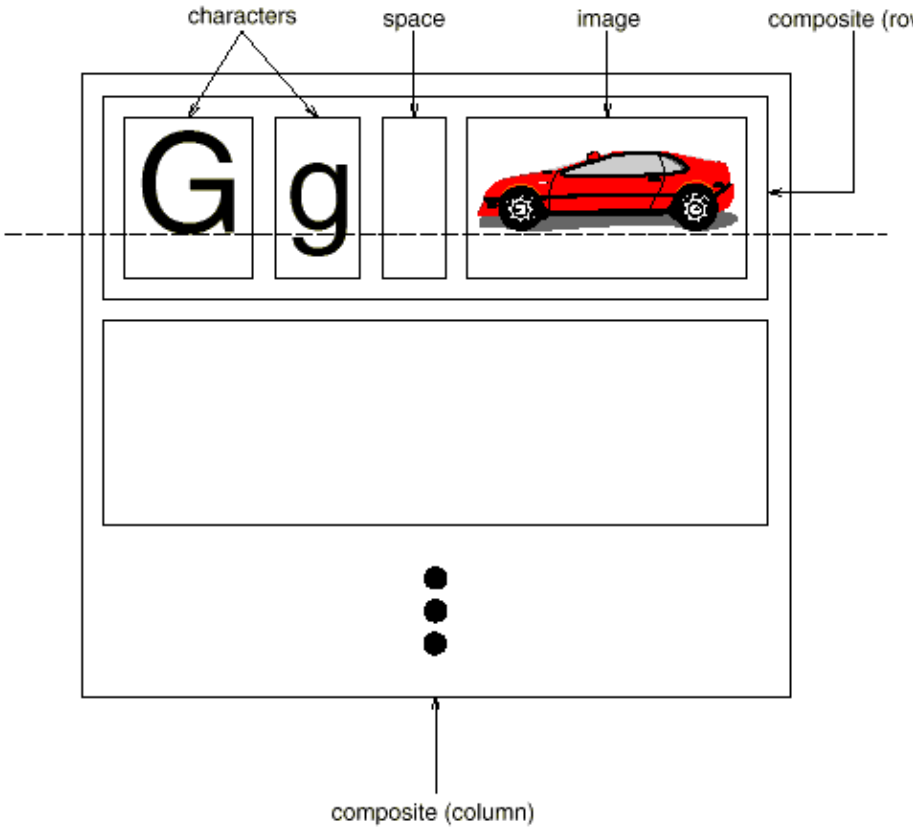

Design considerations

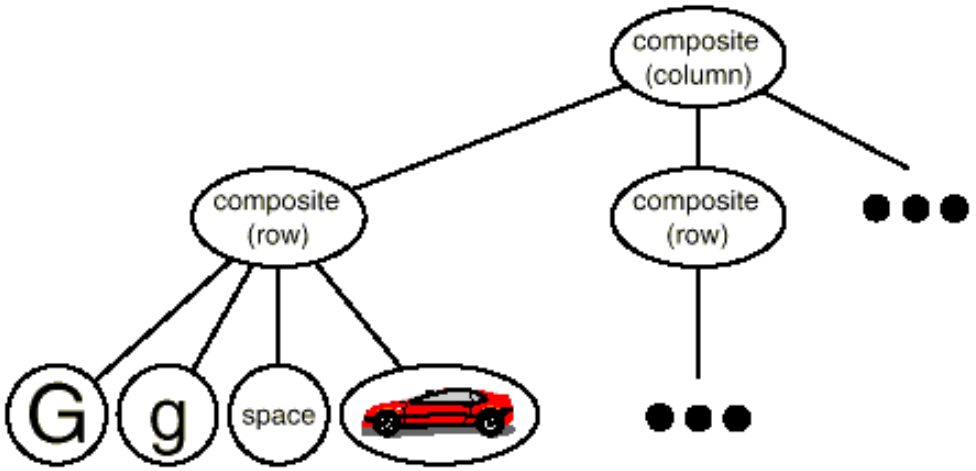
- Document structure
- Formatting
- Embellishing UI
- Multiple look-and-feel
- User operations (buttons, menus)
- More…

First question: how to organize document structure?

- Characters, images, and maybe other type of graphic features
- What are objects? How do these objects related to each other? How about rows and columns?

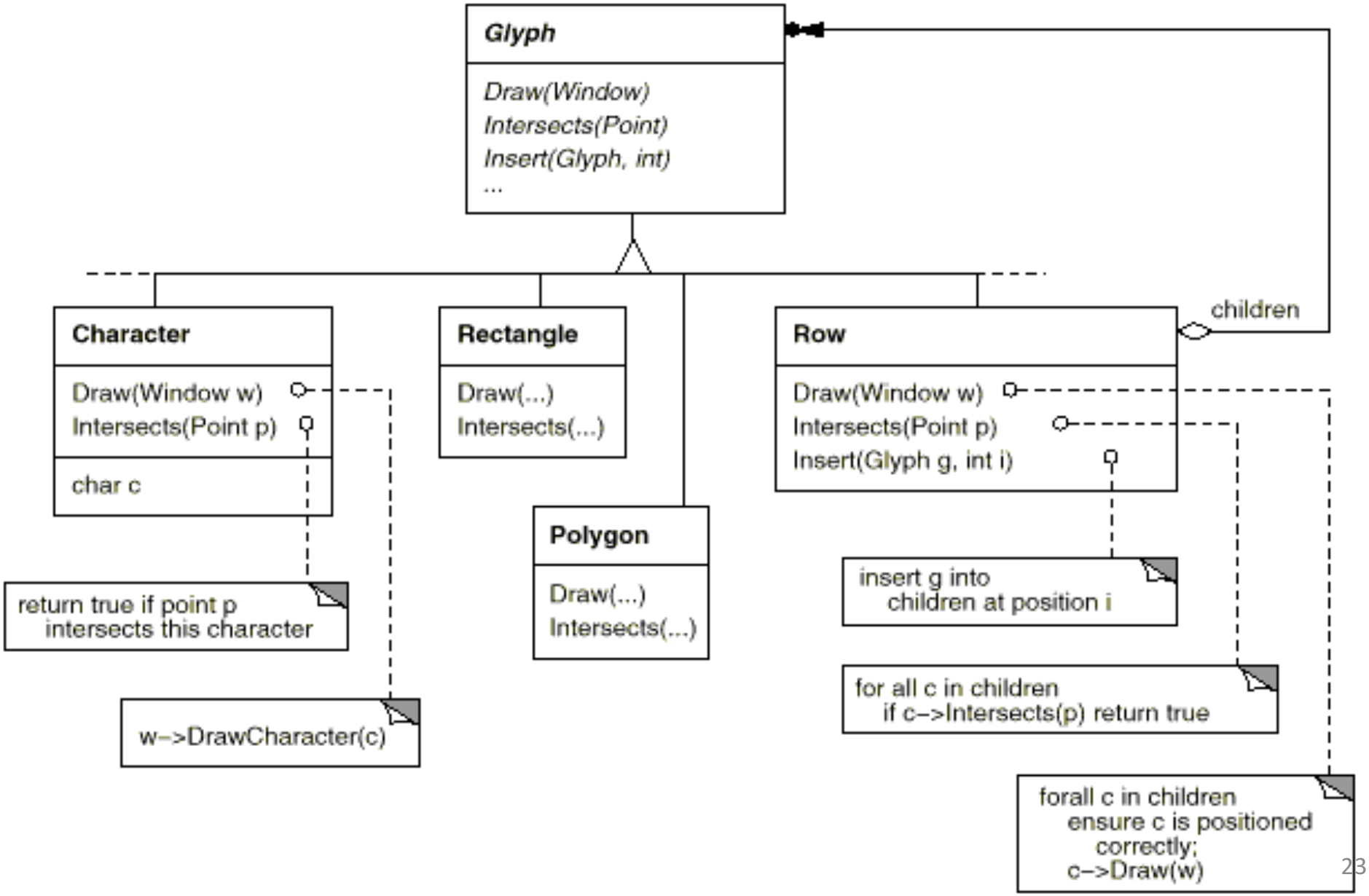
# Document structure: Recursive composition




- **Key**: *uniform* interface for visibles (e.g. chars, images) and invisibles (e.g. rows, columns).
- Build complex structures (e.g. rows) from simpler components (chars, images)

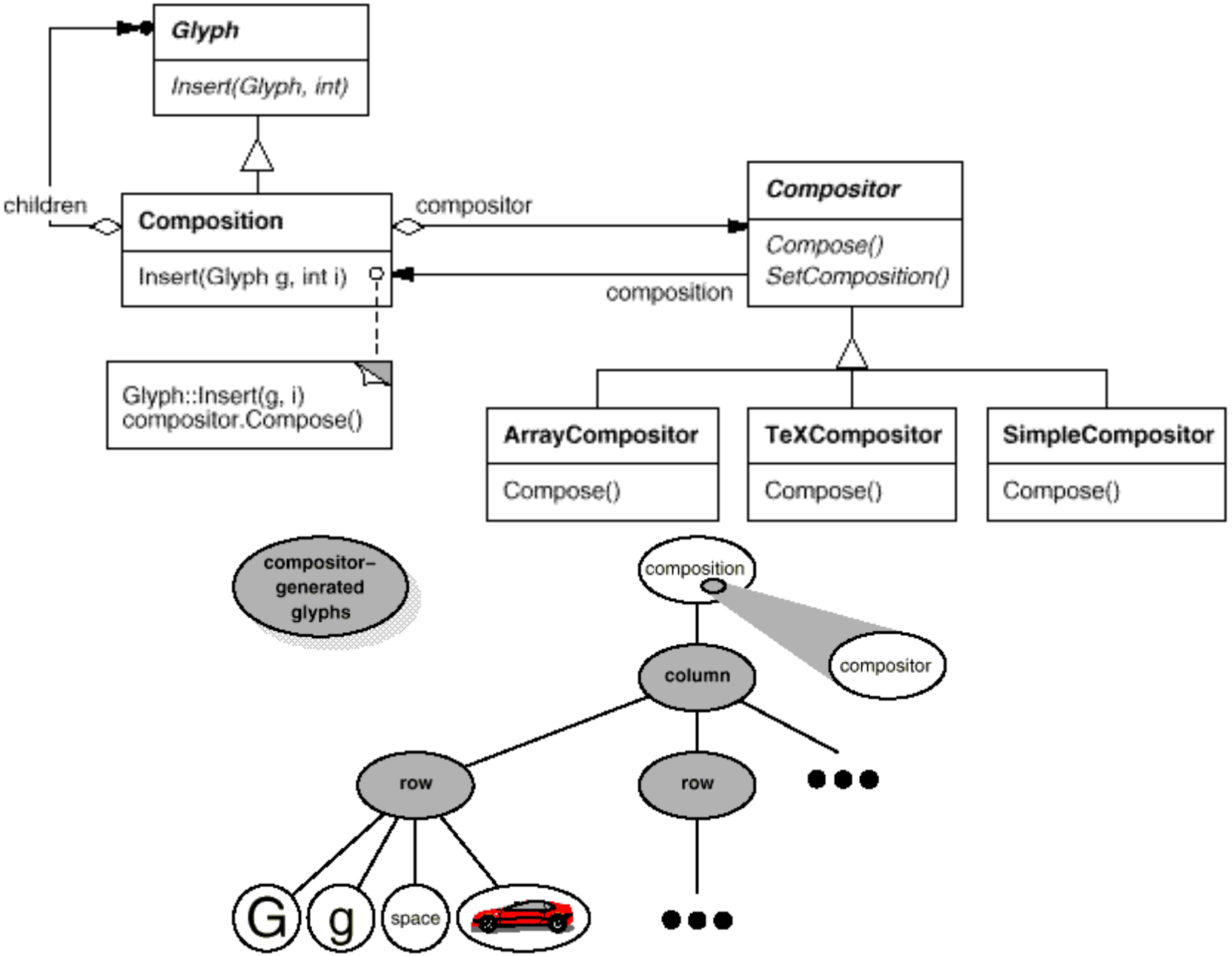
# Glyph class (Composite pattern)



# Formatting

- How to create document structure in the first place? Called line-breaking.
- What we want about formatting?
  - Decouple formatting algorithm with glyph classes.
  - Can change formatting algorithm at least at compile time.
- Idea: create an object to encapsulate formatting algorithm. Common interface, subclass for a particular algorithm.

# Composition and compositor (Strategy pattern)



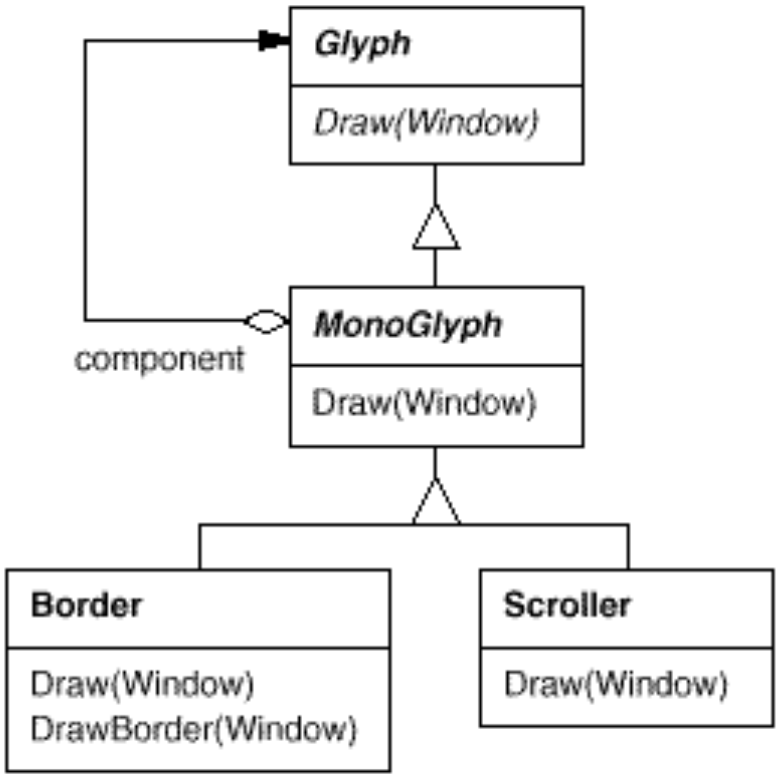
# Embellishing User Interface

- How can we add (or remove) a scroll bar on the side or a border around the view at run-time?
- Inheritance: Composition ➔ BorderedComposition. Or Composition ➔ScrollbarComposition. But what if more embellishment? What if we need to border and scroll bar at the same time?

ABC
DEF
…

# Embellishing via object composition

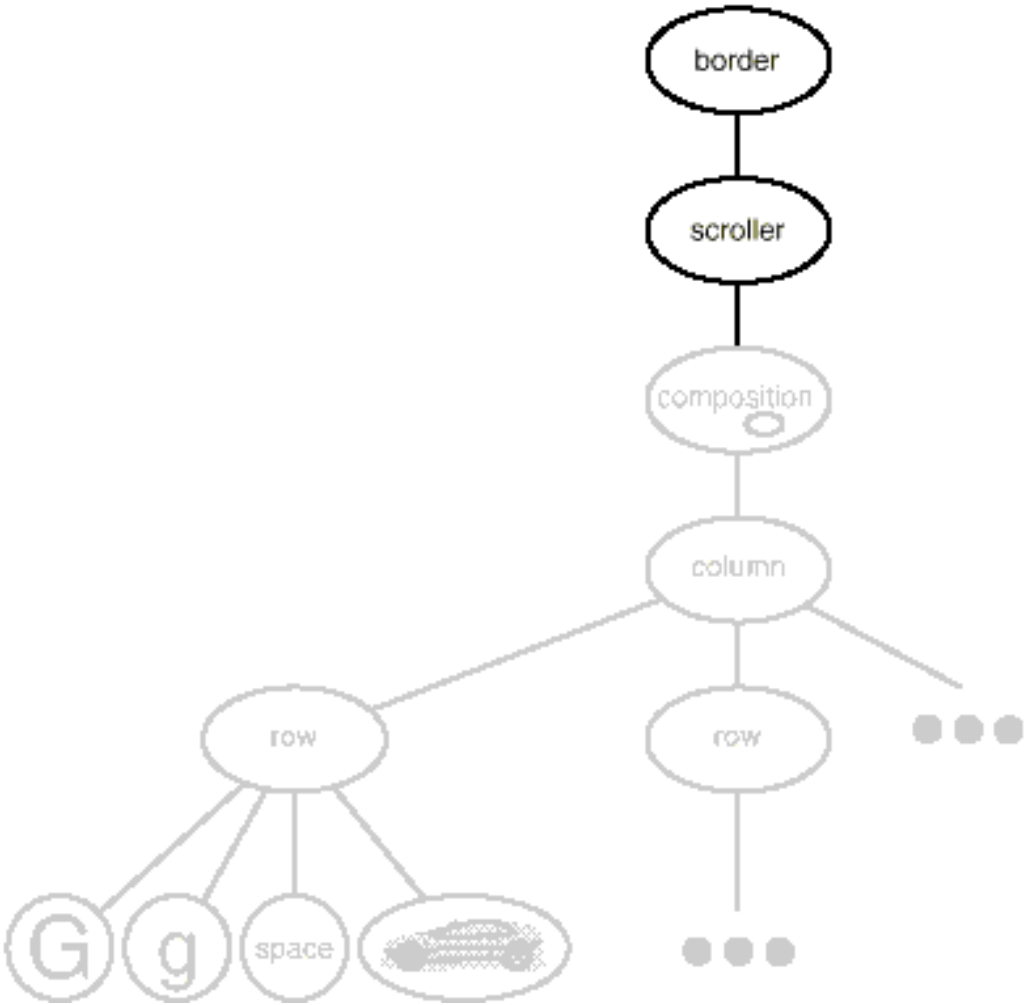
- What type is border anyway? A glyph!
- Border and an existing glyph: which contains which? The consequence of letting border being part of the glyph?
- Now border contains glyph, called transparent enclosure.




```
Public class MonoGlyph  implements Glyph {
Public void Draw (Window w)  {
                _component.Draw(w);
}.. }

Public class Border extends MonoGlyph {
Public void Draw (Window w)  {
                super.Draw(w);
                DrawBorder(w);
} … }
```

# Embellishing UI: Decorator pattern




- Scrollbar contains the composition, and then border contains scrollbar.
- When rendering, border first renders its component (scrollbar). In turn, scrollbar first renders the composition.

# Multiple look-and-feel

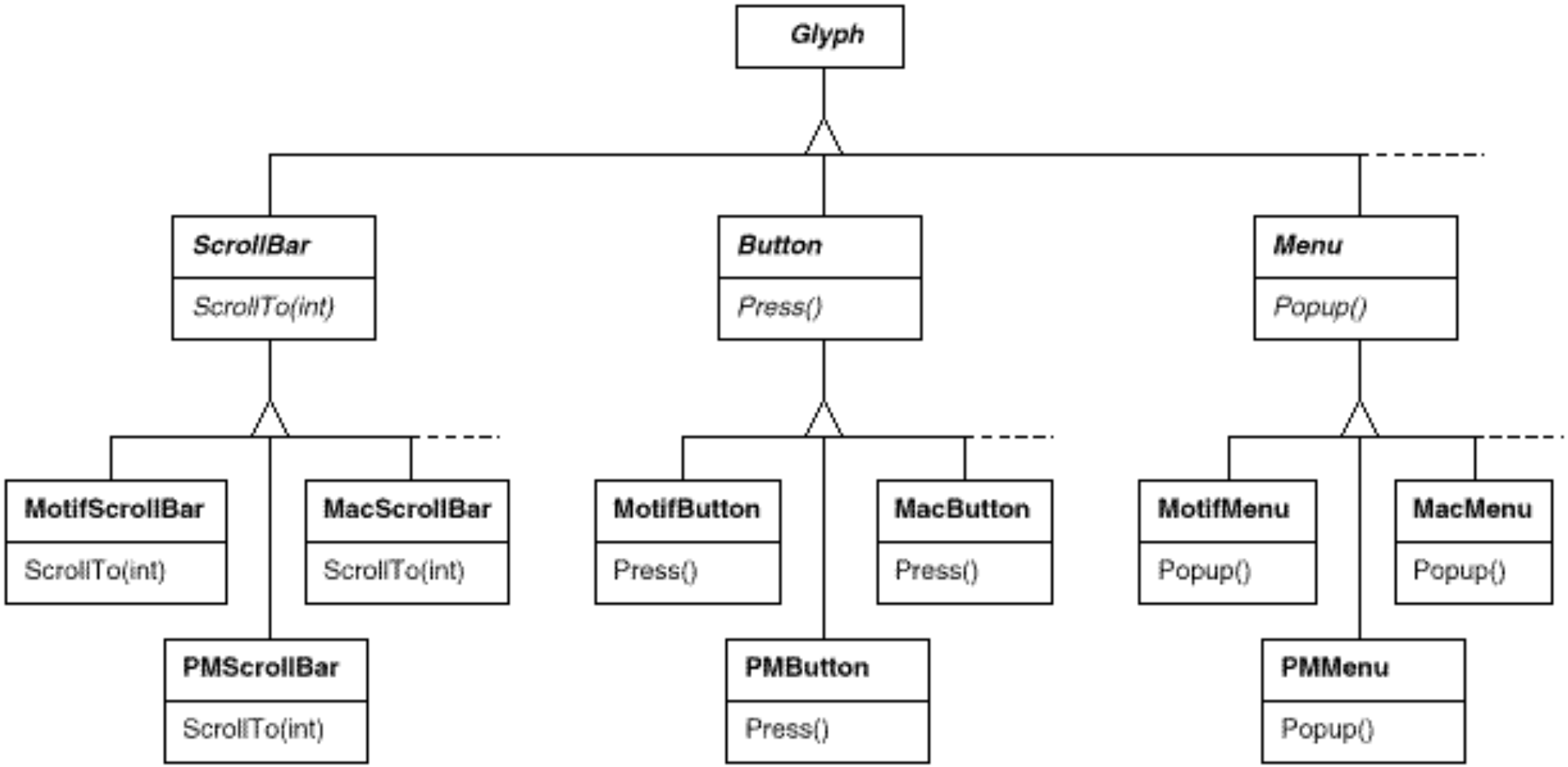
- How to ensure portability across hardware and software platforms?
  - Consider, e.g. UI components are different on different GUI platform
- It is not desirable to commit to a single type of object like:

```
ScrollBar sb = new MotifScrollBar;
```

- A better way is:

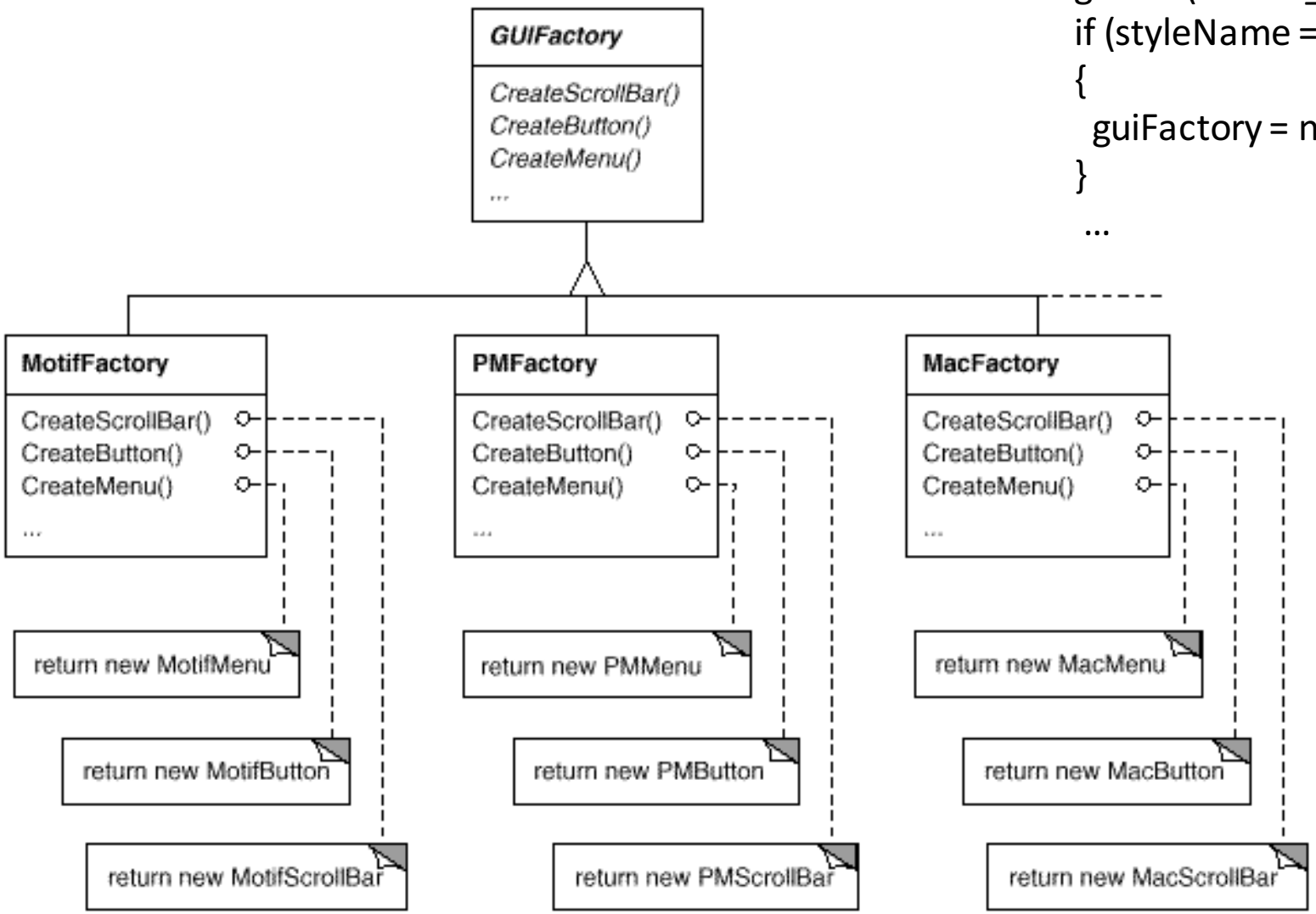
```
ScrollBar sb = guiFactory.CreateScrollBar();
```

# GUI factory class hierarchy (Abstract Factory Pattern)



# GUI factory class hierarchy (Abstract Factory Pattern)

```
GUIFactory guiFactory;
const string styleName =
getenv("LOOK_AND_FEEL");
if (styleName ==  "Motif") )
{
  guiFactory = new MotifFactory;
}
 …
```

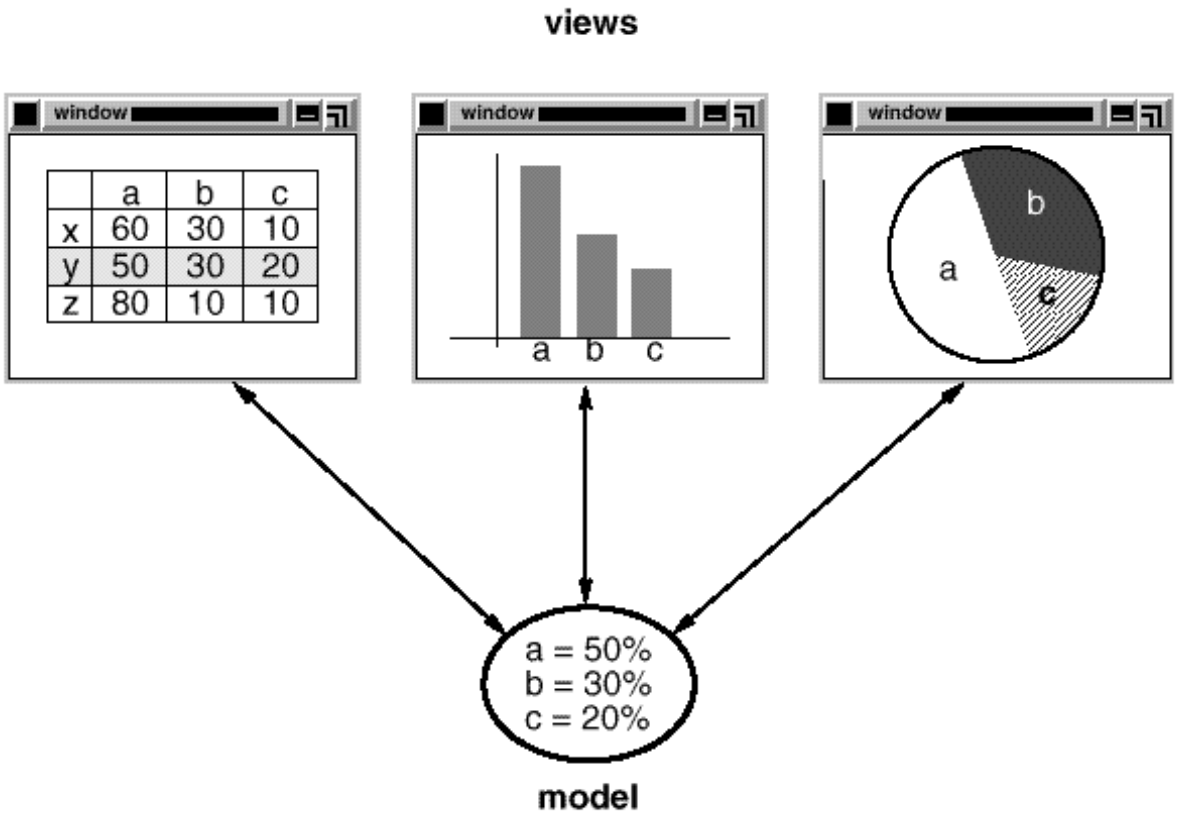


# Design patterns

- A number of very popular design patterns in previous case study
- Now we will study them closely
  - Focus on concepts and real application to, say Java JDK framework.
- What is design pattern?
- Why design pattern?
- Example: Model-View-Controller.

# MVC Pattern

- One class solution vs. three classes solution
- Subscribe-notify protocol: why is a good design?
- View-controller: allow change of user input response.



# Conclusion

- This lecture: OO-design basics and case studies.
- Now, design patterns in details, with three types
  - Creational
  - Structural
  - Behavioral