UT DALLAS

CE/CS/SE 3354 Software Engineering

Lecture 09
Software Design
Design Patterns

Software Design Process

- Software design is usually created in two stages
  - Architecture design
  - Detailed design (component-level design)

    Design classes (e.g., using class diagrams)

    Design patterns help with better class design

Software Design Factors

- Fundamental software design factors
  - Modularity
    Abstraction
    Information hiding
  - Component independence
  - Fault prevention and fault tolerance

Modularity and Abstraction

- When we consider modular solutions to any problems, many levels of abstraction can be posed
    - At the highest level of abstraction, a solution is stated in broad terms of problem domain: architecture
    - At the lower levels of abstraction, a more detailed description of the solution is provided: class diagrams
- Modularity hides details and facilitates evolvement
    - Each component hides a design decision from the others

Component Independent

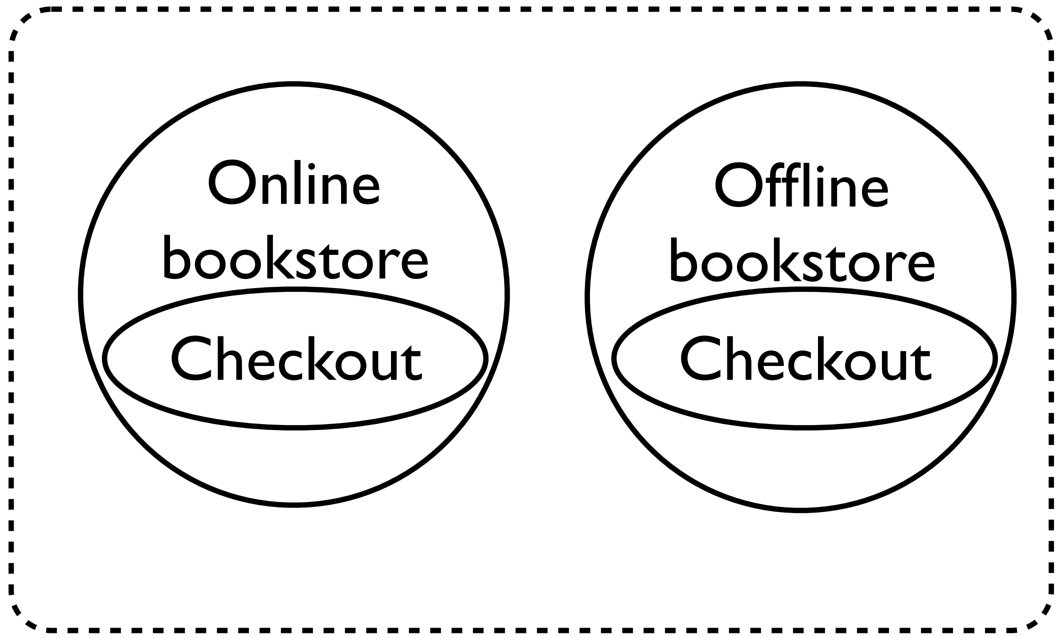
- We strive in most designs to make the components independent of one another.
- We measure the degree of component independence using two concepts
  - Low coupling
  - High cohesion

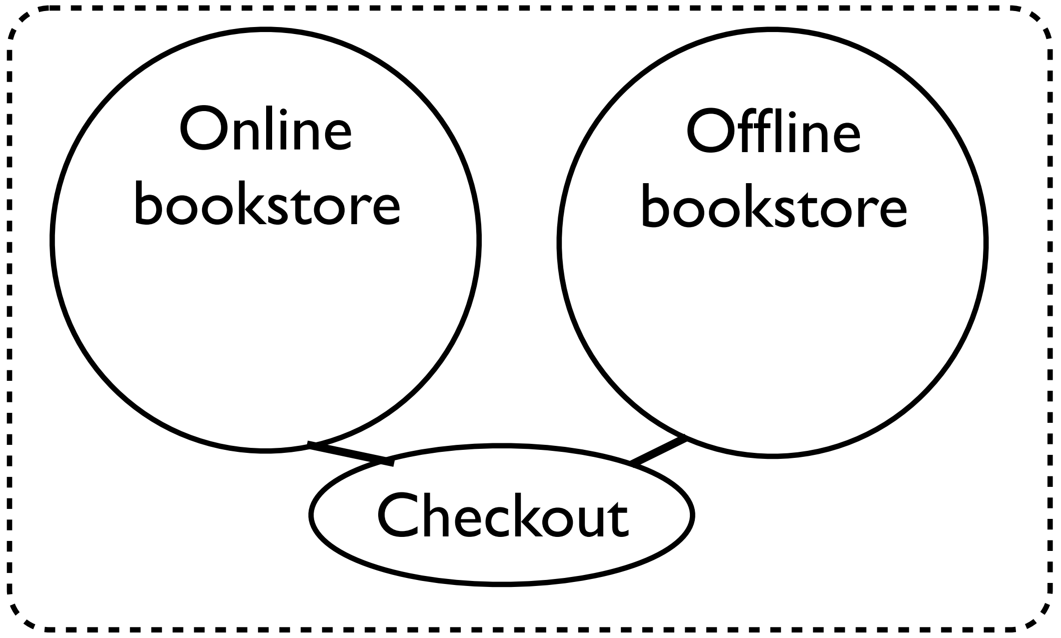
Coupling and Cohesion

- Coupling
  - Two components are highly coupled when there is a great deal of dependence between them
  - Two components are loosely coupled when they have some dependence, but the interconnections among them are weak
  - Two components are uncoupled when they have no interconnections at all
- Cohesion
  - A component is cohesive if the internal parts of the component are related to each other and to its overall purpose

Decoupling

- Most difficult part in design: need tradeoff
- Consider the following case:
  - Online and offline bookstore components
  - Both need shopping cart checkout





Should we have a shopping cart checkout module for each of them?

Low cohesion

Or have one, and let the two components to call the module?

High coupling

Fault Defense & Tolerance

- Defensive design anticipates situations the might lead to problems
  - Network Failure
  - Data corruption
  - Invalid user inputs
- Tolerate runtime errors
  - Exception handling
  - Redundant components (distributed system or critical software)
  - Timely error reporting

Criteria for Good Software Design

- High-quality designs should have characteristics that lead to quality products
  - Correct translation from the requirements specification
  - Ease of understanding
  - Ease of implementation
  - Ease of testing
  - Ease of modification

Criteria for Good Software Design

- Architecture
  - Using suitable architectural styles or patterns
  - Loose-coupled components
- Classes at a suitable abstract level
- Interfaces are clear and minimize the data transfer
- Design using an effective notation

Software Design Evaluation and Validation

- We check a design in two different ways
  - Validation: the design satisfies all requirements specified by the customer
  - Verification: the characteristics (quality) of a good design are incorporated
- We use some techniques for helping us to perform verification and validation
  - Measuring design quality
  - Design reviews

Measuring Software Design Quality

- We check a design using a set of measures
  - Coupling
  - Cohesion
  - Size

    Lines of code (be careful about comments and spaces)

    number of methods

    number of classes

    ...
  - Complexity

    Cyclomatic complexity metric

Cyclomatic Complexity

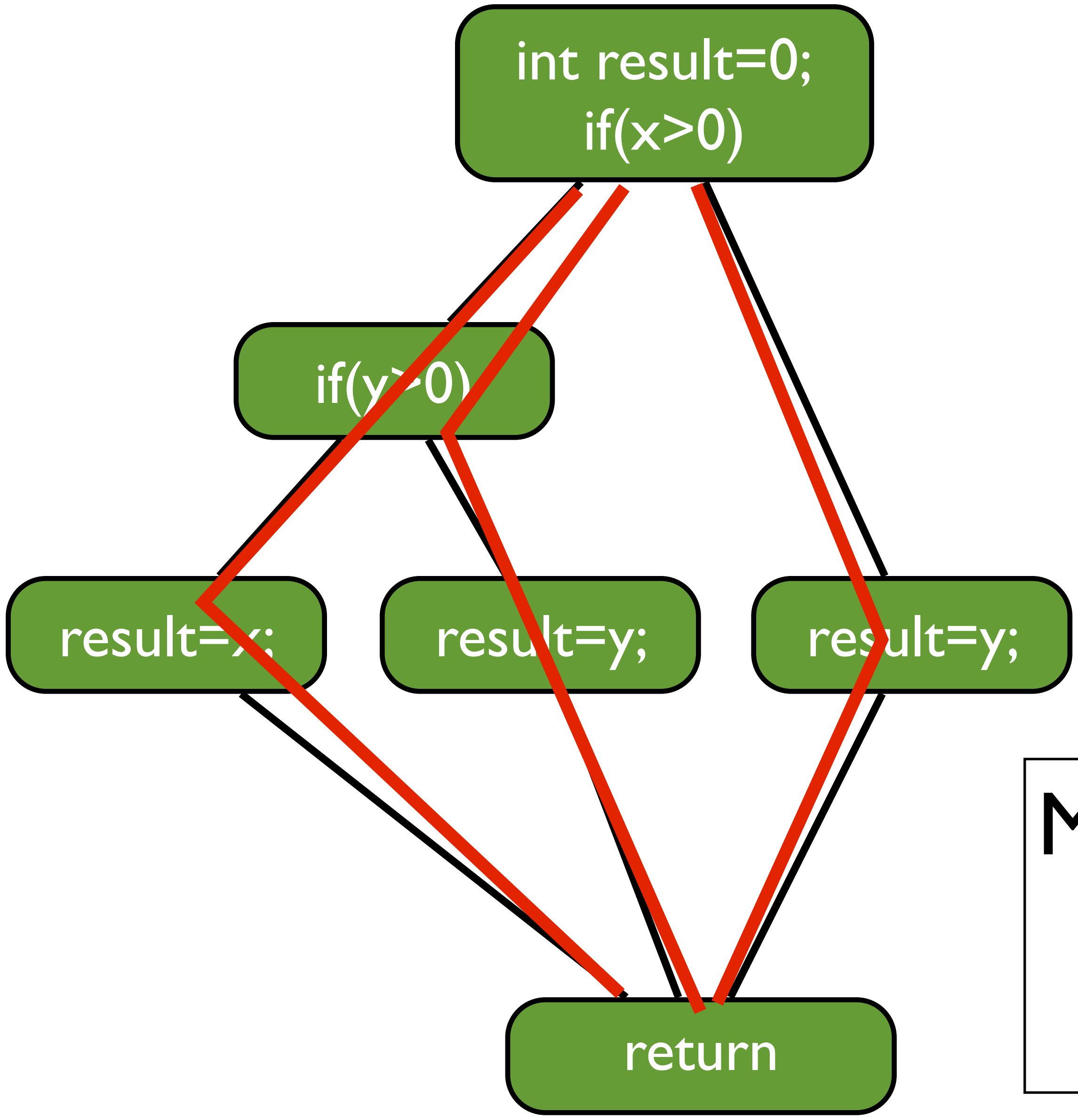
- Cyclomatic complexity is a software metric (measurement)
- Developed by Thomas J. McCabe in 1976. Also called the McCabe Complexity Metric
- A quantitative measure of the complexity of programming instructions
  - Lower value means easier-to-understand and less fault-prone code
- Directly measures the number of linearly independent paths through a program's source code

Cyclomatic Complexity (Cont'd)

- A software module is described by a control flow graph
  - each node corresponds to a block of sequential code
  - each edge corresponds to a branch/jump transition (caused by if, while, for, switch, etc)
- $V(G) = e - n + 2p$
  - e = number of edges in the graph
  - n = number of nodes in the graph
  - p = number of connected module components in the graph

Cyclomatic Complexity: Example

```
int result=0;
if(x>0) {
    if(y>0){
        result =x;
    }else {
        result=y;
    }
}else {
    result =y;
}
return result;
```



e=7
n=6
p=1

McCabe=e-n+2p
=7-6+2
=3

Design Reviews

- We check a design with
  - Designers
  - Customers
  - Analysts
  - Prospective users
  - Developers
- Moderator leading the discussions
- Secretary recording the issues

This Lecture

- Software Design
  - Process
  - Factors
  - Measures
- Design Patterns
  - Composite pattern
  - Factory pattern
  - Visitor pattern

Design Patterns

- Become popular due to a book
  - Design Patterns: Elements of Reusable Object-Oriented Software
  - Gang of Four: Erich Gamma; Richard Helm, Ralph Johnson, and John Vlissides
- Provide solutions for common problems in micro-design

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns Structure

- Name
  - Important to know for easier communication between designers
- Problem solved
  - When to apply the pattern
- Solution
  - Usually a class diagram segment (sometimes attributes and operations)

Design Patterns

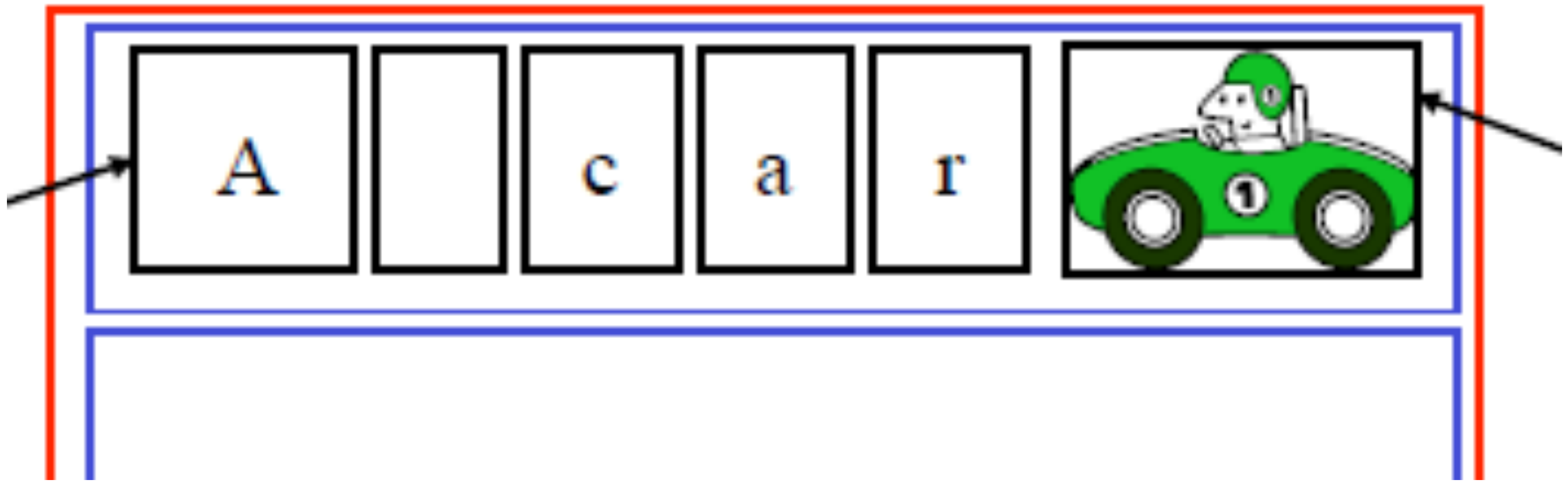
- Structural Patterns
  - deal with the composition of classes or objects
- Creational Patterns
  - concern the process of object creation
- Behavioral Patterns
  - characterize the ways in which classes or objects interact and distribute responsibility

Design Patterns

- Structural Patterns
  - Composite
- Creational Patterns
  - Factory
- Behavioral Patterns
  - Visitor

Running Example

- A Document Editor
  - Text and graphics can be freely mixed
  - Graphical user interface
  - Support different look-and-feel GUI styles
  - Traversal operations: spell checking

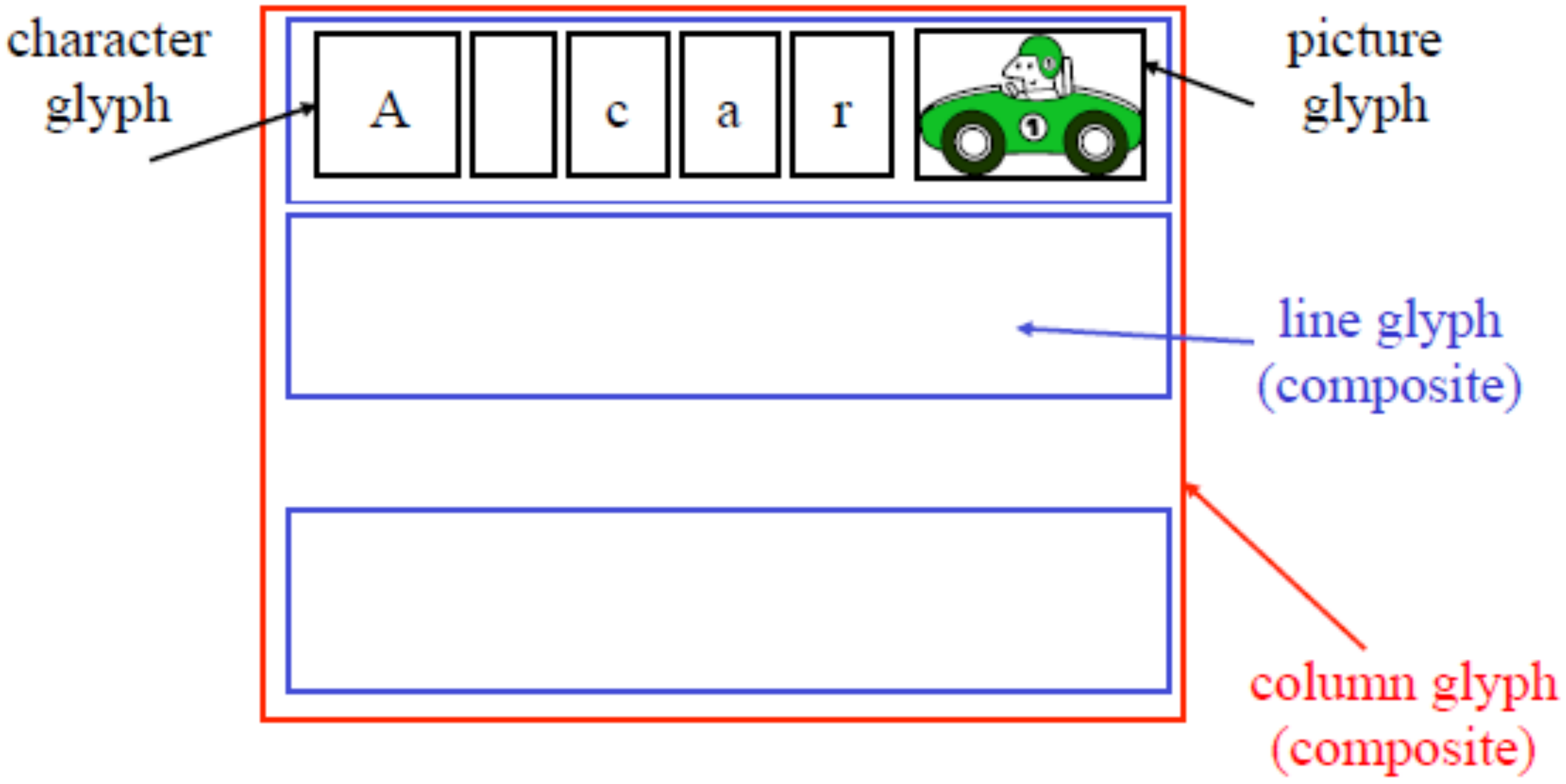


Composite Pattern: Example

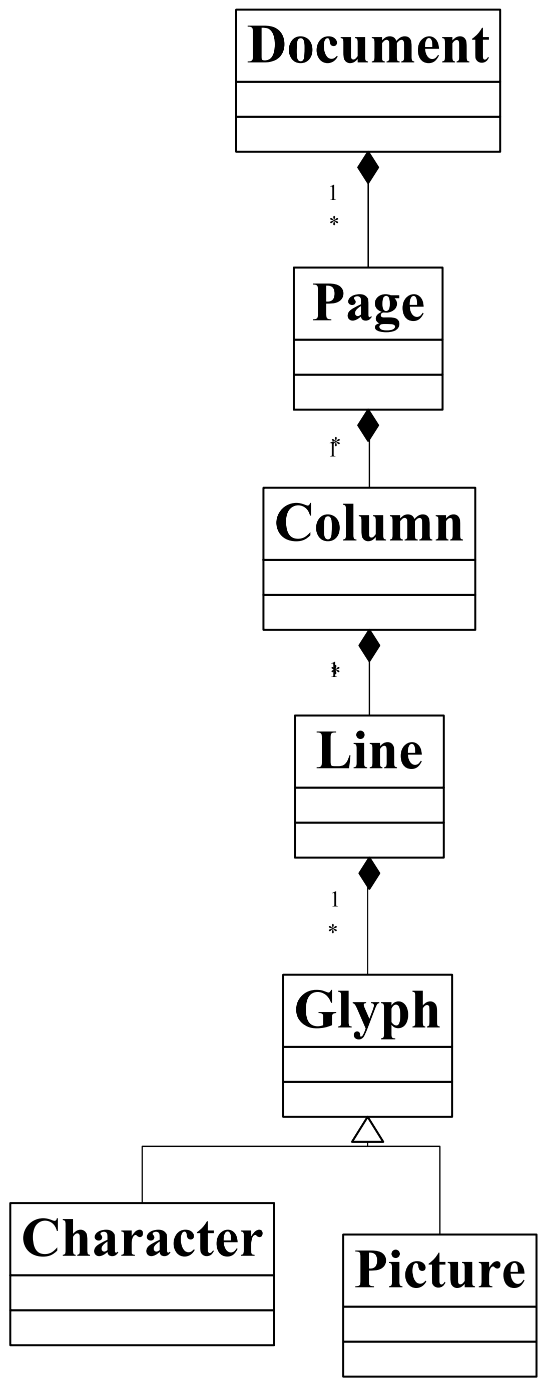
- A document is represented by structure
  - Primitive glyphs

    Characters, pictures
  - Lines: A sequence of glyphs
  - Columns: A sequence of Lines
  - Pages: A sequence of Columns
  - Documents: A sequence of Pages

Example of Hierarchical Document



Possible Designs?



- Classes of Character, Circle, Pictures, lines, Columns, pages, document
  - Composition relationship between classes
  - Not so good, why?

- A lot of duplication: All classes has onClick, draw behavior
- Difficult to add or remove levels in hierarchy
- Traverse/display operations need to change for any change of the hierarchy

Alternate Designs

- One class of Glyph
  - All elements are subclasses of Glyph
  - All elements uniformly present the same interface

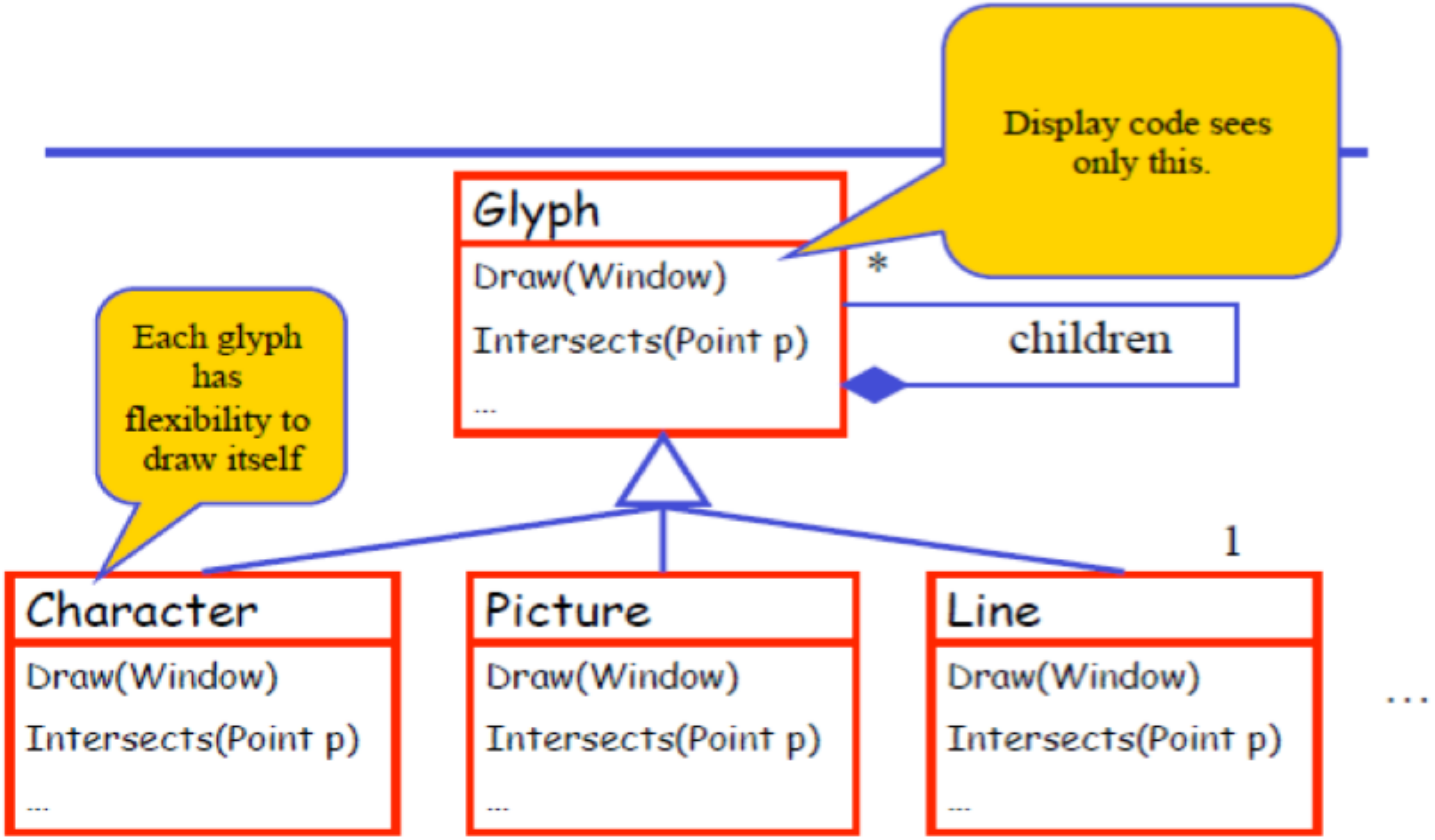
    How to draw

    Computing bounds

    Mouse hit detection

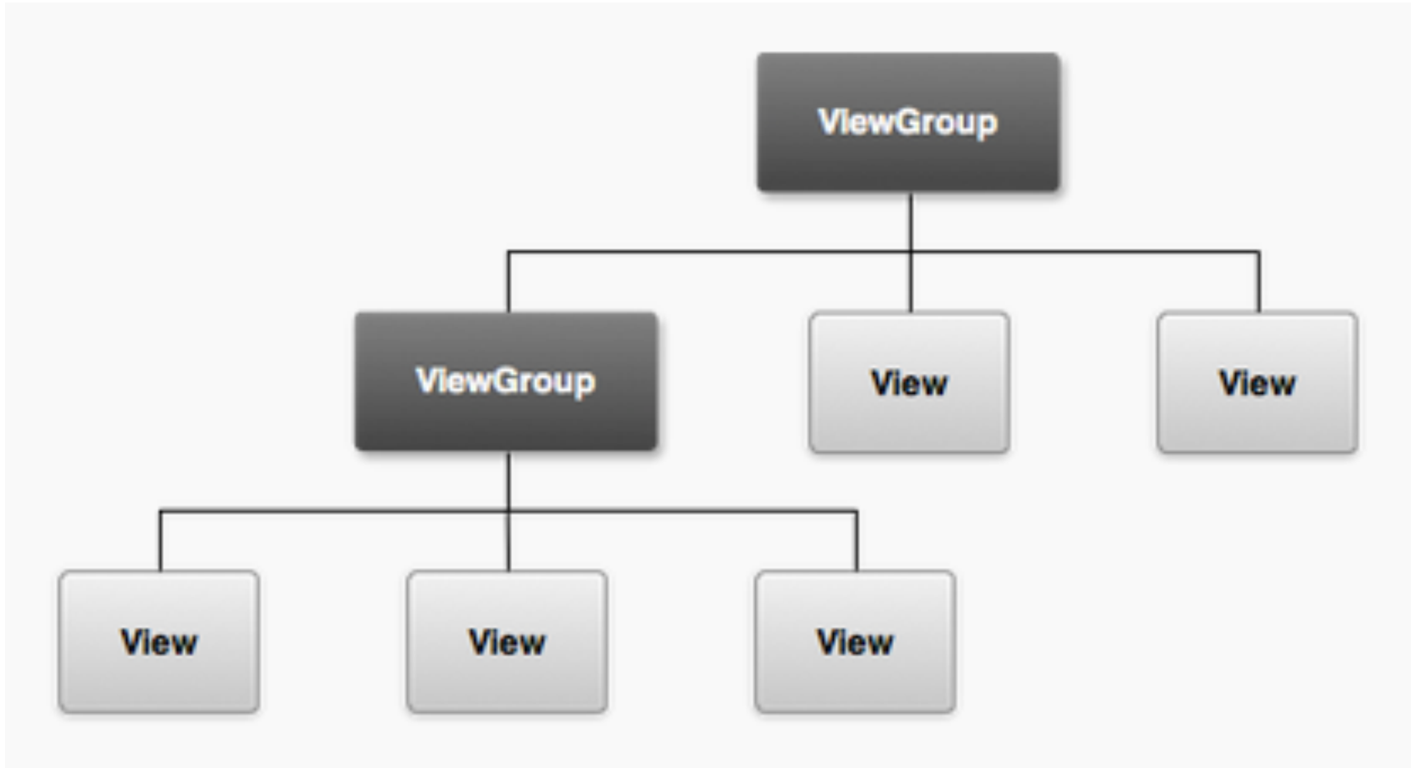
    …
  - Make extending the class easier

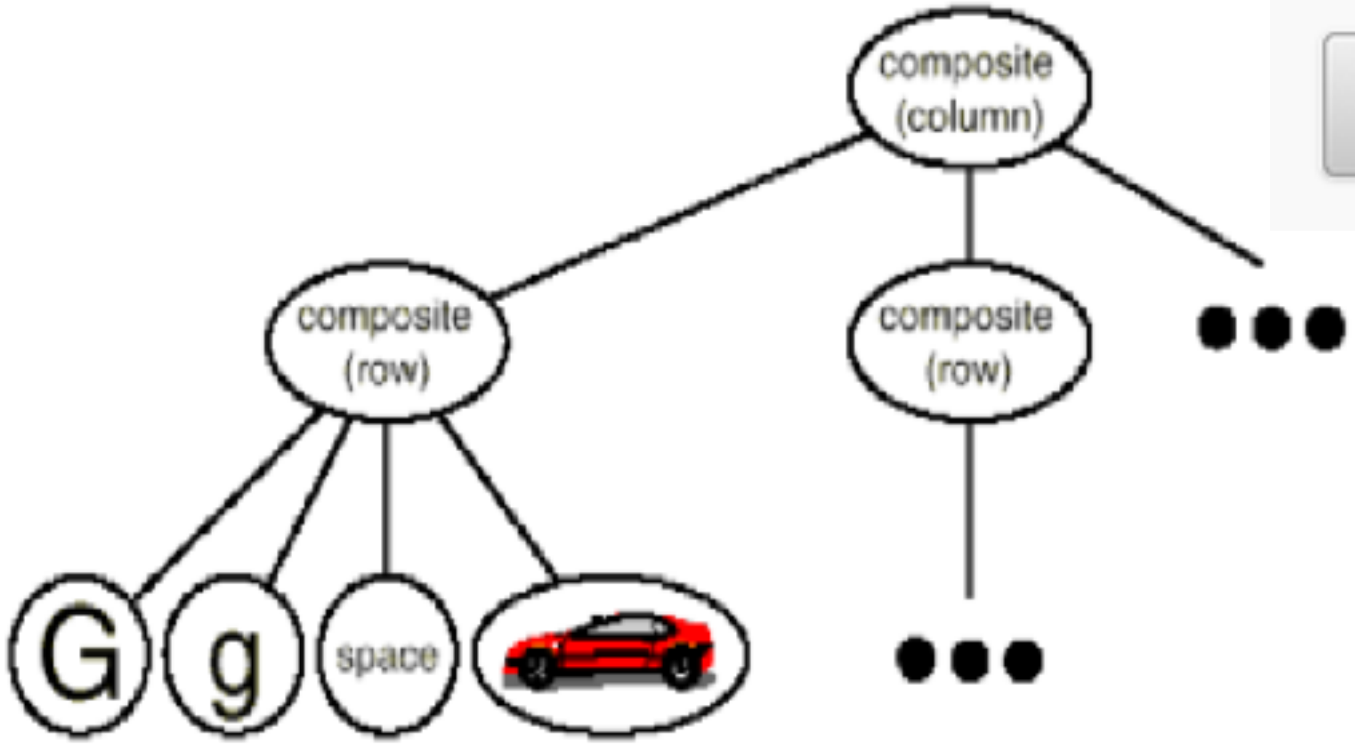
Composite - Class Diagram



Logic Object Structure

Leaves and internal nodes expose the same interface





Composite Pattern

- Applies to any hierarchy structure
  - Leaves and internal nodes have similar functionality

- No Composite Pattern
  - A lot of duplication
  - Difficult to add or remove levels in hierarchy
  - Traverse/display operations need to change for any change of the hierarchy

- With Composite Pattern
  - Less duplication, clean interface
  - Easy to add, remove new types
  - Traverse/display operations is uniform for internal nodes and leaves

Design Patterns

- Structural Patterns
  - Composite
- Creational Patterns
  - Factory
- Behavioral Patterns
  - Visitor

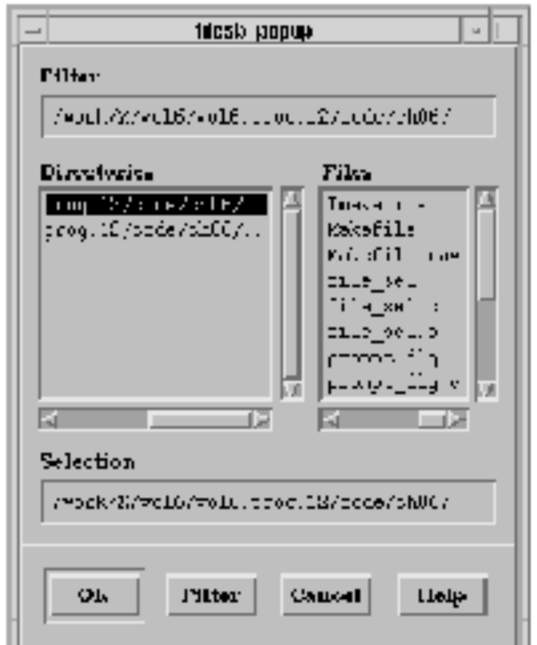
Problem: Supporting Different GUI Styles

- Different GUI styles
  - Appearance of scrollbars, menus, windows, etc., set by the user
  - We want to support them all

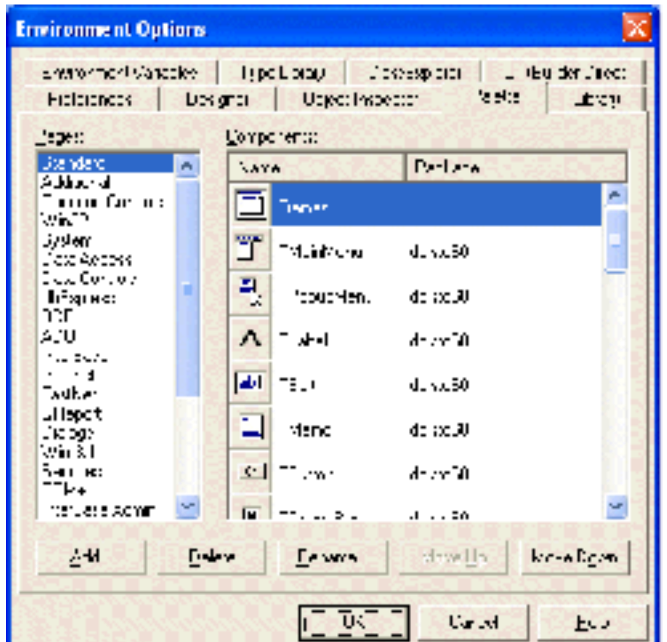
    For example: MotifScrollBar and WindowsScrollBar

    Both are subclasses of ScrollBars

    How should we create a new scrollbar in a window?



Motif Style



Windows Style

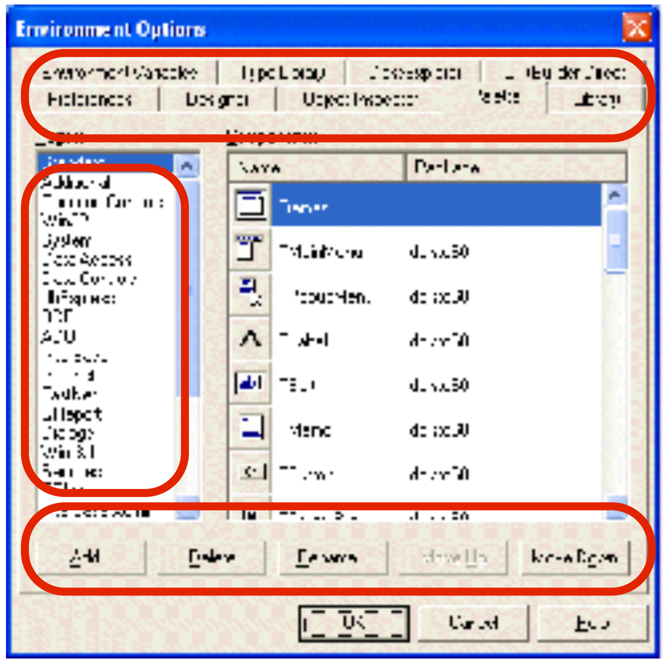
Possible designs?

- Terrible

```
ScrollBar sc = new WindowScrollBar();
```

- Better, problems?

```
If (style==MOTIF){
     sc = new MotifScrollBar();
}else{
     sc = new WindowScrollBar();
}
```

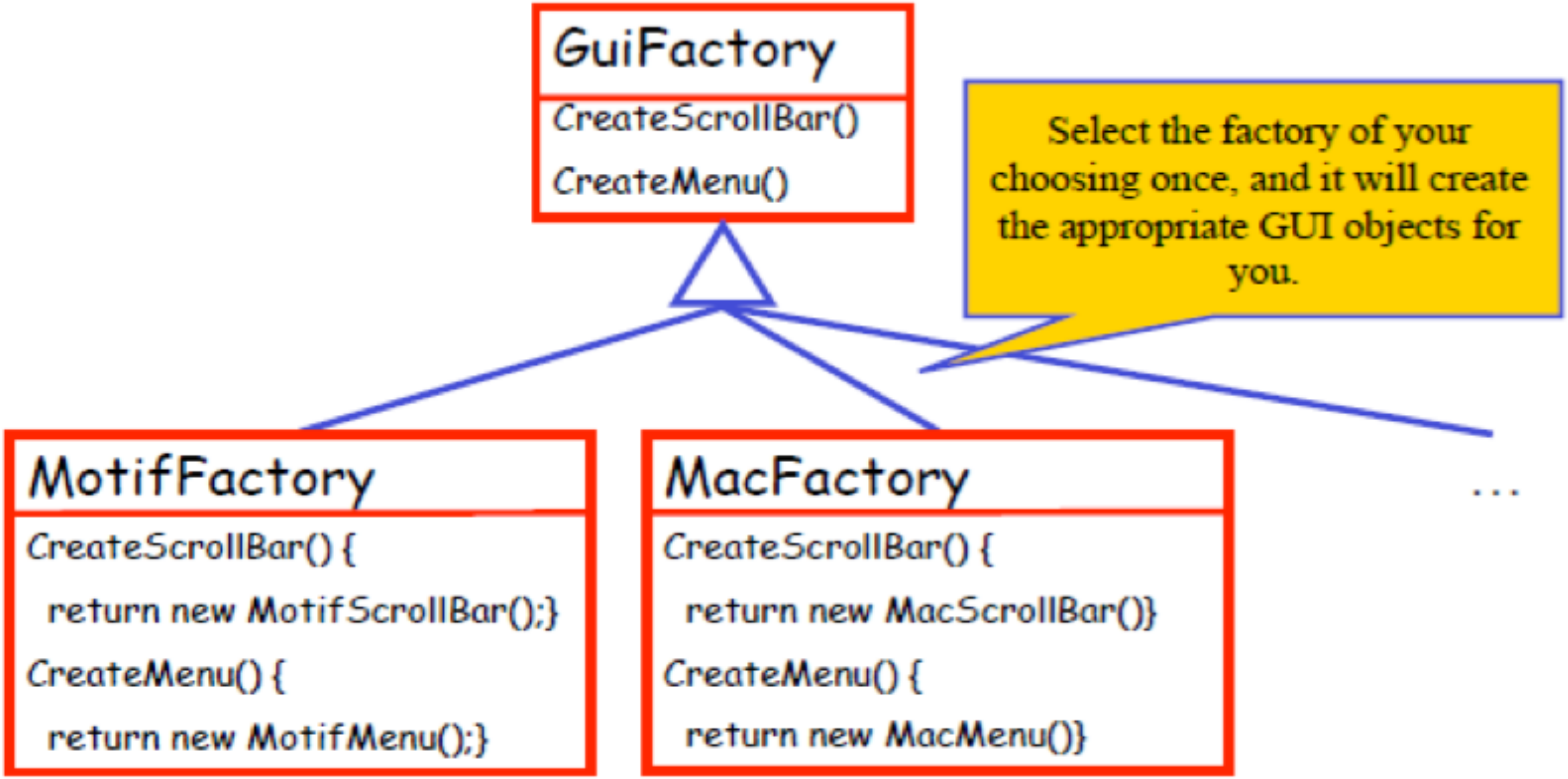


Need conditions for other UI elements as well: menus, etc.
Hard to add new styles

Factory Pattern

- One method to create a style dependent object
- Define a GUIFactory Class
  - One GUIFactory object for each GUI style
  - Create itself using conditions set by the user

Factory Pattern

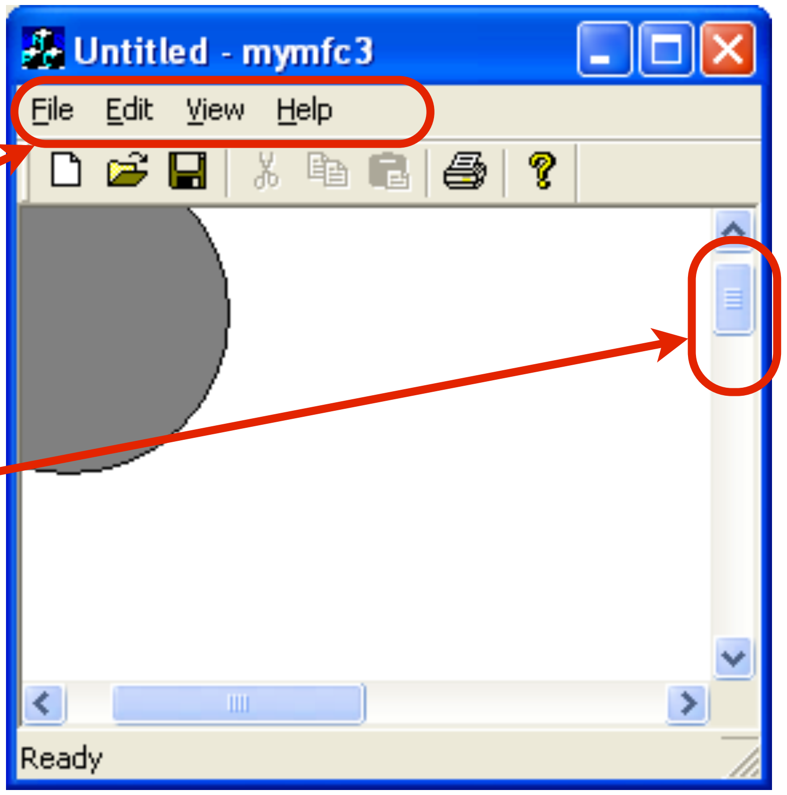


Factory Pattern: Factory Design

```
abstract class GUIFactory{
  abstract ScrollBar CreateScrollBar();
  abstract Menu CreateMenu();
  ...
}

class MotifFactory extends GUIFactory{
  ScrollBar CreateScrollBar(){
     return new MotifScrollBar()
  }
  Menu createMenu(){
     return new MotifMenu();
  }
  ...
}
```

Factory Pattern: GUI code

```
GUIFactory factory;
if(style==MOTIF){
    factory = new MotifFactory();
}else if(style==WINDOW){
    factory = new WindowFactory();
}else{
  ...
}
Menu mn = factory.createMenu();
ScrollBar bar = factory.createScrollBar();
```



Factory Pattern

- Applies to the object creation of a family of classes
- The factory can be changed at runtime
- Pros & Cons
  - Lift the conditional creation of objects to the creation of factories
  - Flexible for add new type of objects
  - Sometimes make the code more difficult to understand

Visitor Pattern

"Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

- Gang of Four

Polymorphism

- In object-oriented development, objects of different sub-type can have different behaviors
- Alice (teacher), Bob (engineer) are both employee, and they both work
- Alice works (she teach), while Bob works (he design)

- An object can be referred (called) by its super type
- But it behaviors only as its own type (the type it gets at its construction)

A Question

```
Class A{public String toString(){ return "ClassA";}}
Class B extends A{public String toString(){ return "ClassB";}}
Class Main{
    public void print(Object obj){ System.out.println(obj.toString());}
    public static void main(String args[]){
        Object obj = new B();
        A a = (A)obj;
        print(a);
    }
}
```

What will be printed out?
A:"ClassA", B:"ClassB",
C: the address of a (default behavior of Class Object).

Polymorphism Basic Idea

- Polymorphism can replace conditional statements by feed in a super-type reference
- An interface: when my apartment has problems, the manager will send someone to fix
- An instance: when water pipe leaks, a plumber will come to fix the leak

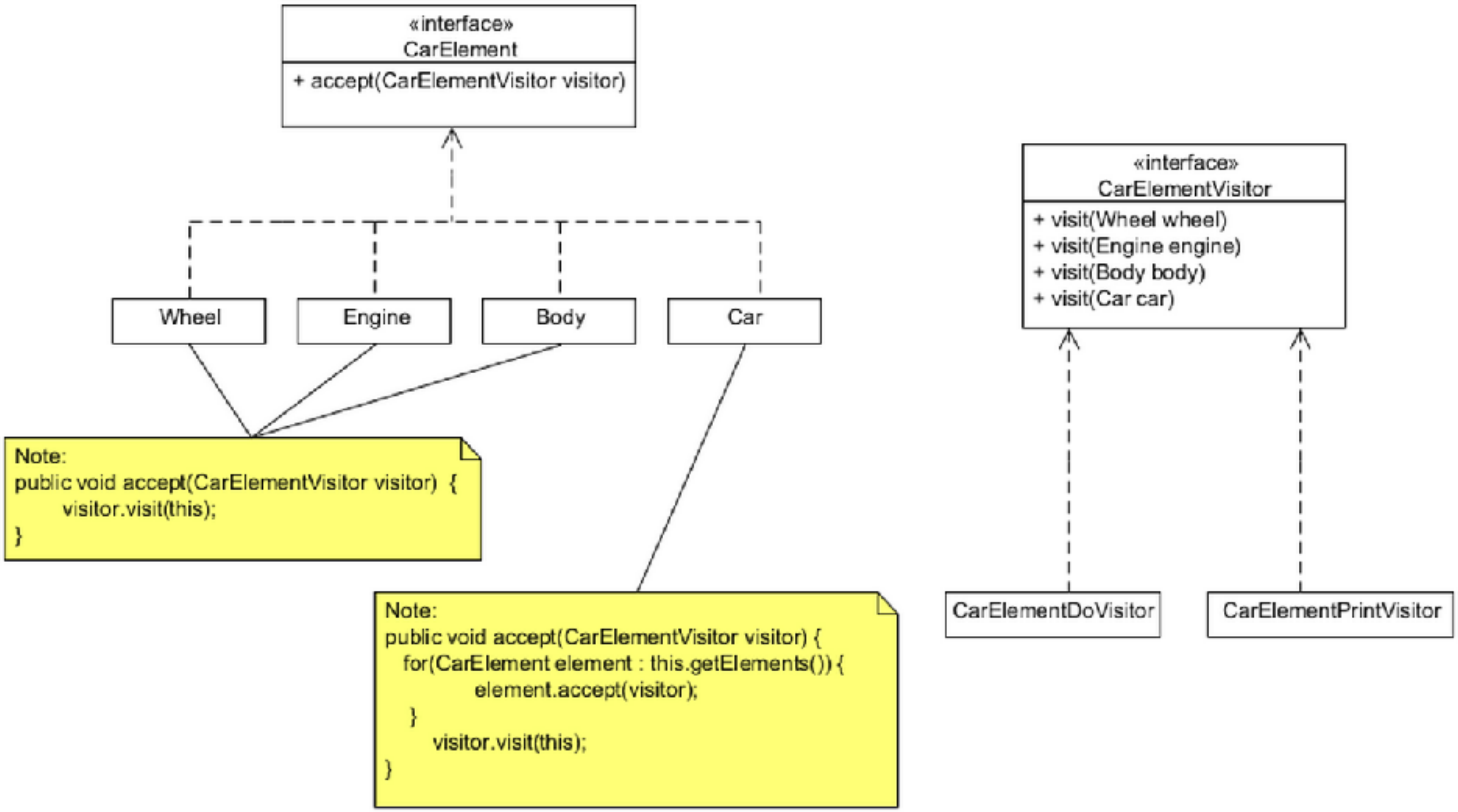
Visitor Pattern - Example

- Check car parts
  - engine, wheels, brakes, sensors, …

Visitor Pattern Mechanism

- The visitor pattern requires a programming language that supports polymorphism
- Uses two types of objects, each of some class type; one is called "element", and the other is called "visitor"
  - “element” includes the accept() method for traverse
  - “visitor” defines the visit() method for action
  - Try to decouple traverse and action

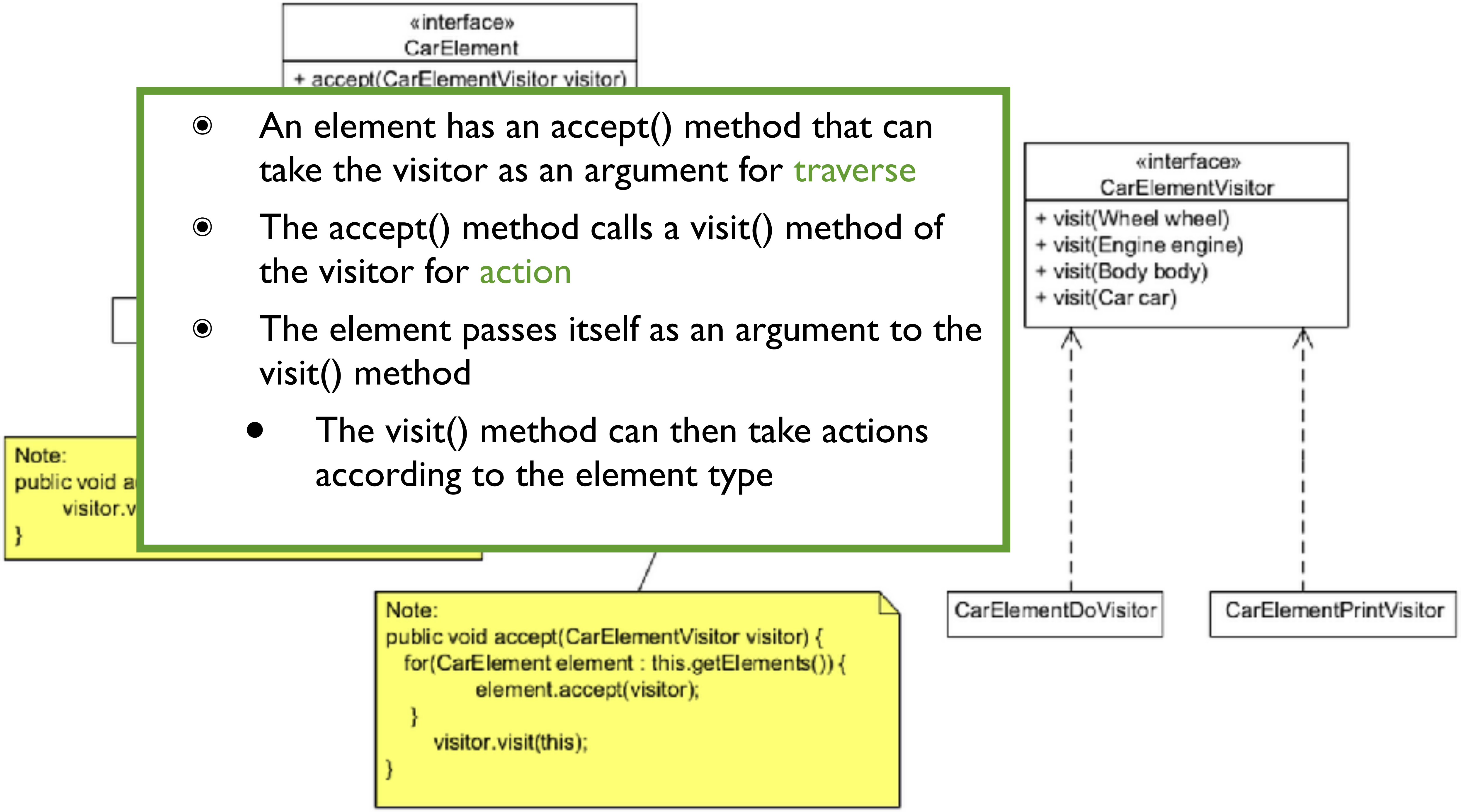
Visitor Pattern - Example



Note:
```
public void accept(CarElementVisitor visitor)  {
        visitor.visit(this);
}
```

Note:
```
public void accept(CarElementVisitor visitor) {
   for(CarElement element : this.getElements()) {
            element.accept(visitor);
    }
        visitor.visit(this);
}
```

Visitor Pattern - Example



```
Note:
public void accept(CarElementVisitor visitor) {
  for(CarElement element : this.getElements()) {
         element.accept(visitor);
   }
      visitor.visit(this);
}
```

- An element has an accept() method that can take the visitor as an argument for traverse
- The accept() method calls a visit() method of the visitor for action
- The element passes itself as an argument to the visit() method
  - The visit() method can then take actions according to the element type

Visitor Pattern - Interface

```
interface ICarElement {
    void accept(ICarElementVisitor visitor);
}
```

Element Interface (Traverse)

```
interface ICarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}
```

Visitor Interface (Action)

Traverse - Code

```
class Wheel implements ICarElement {
    private String name;
    public Wheel(String name) {
        this.name = name;
    }
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}
class Engine implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}
class Body implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

```
class Car implements ICarElement {
    ICarElement[] elements;
    public Car() {
        //create new Array of elements
        this.elements = new ICarElement[] { new
Wheel("front left"), new Wheel("front right"), new
Wheel("back left") , new Wheel("back right"), new Body(),
new Engine() };
    }
    public void accept(ICarElementVisitor visitor) {
        for(ICarElement elem : elements) {
            elem.accept(visitor);
        }
        visitor.visit(this);
    }
}
```

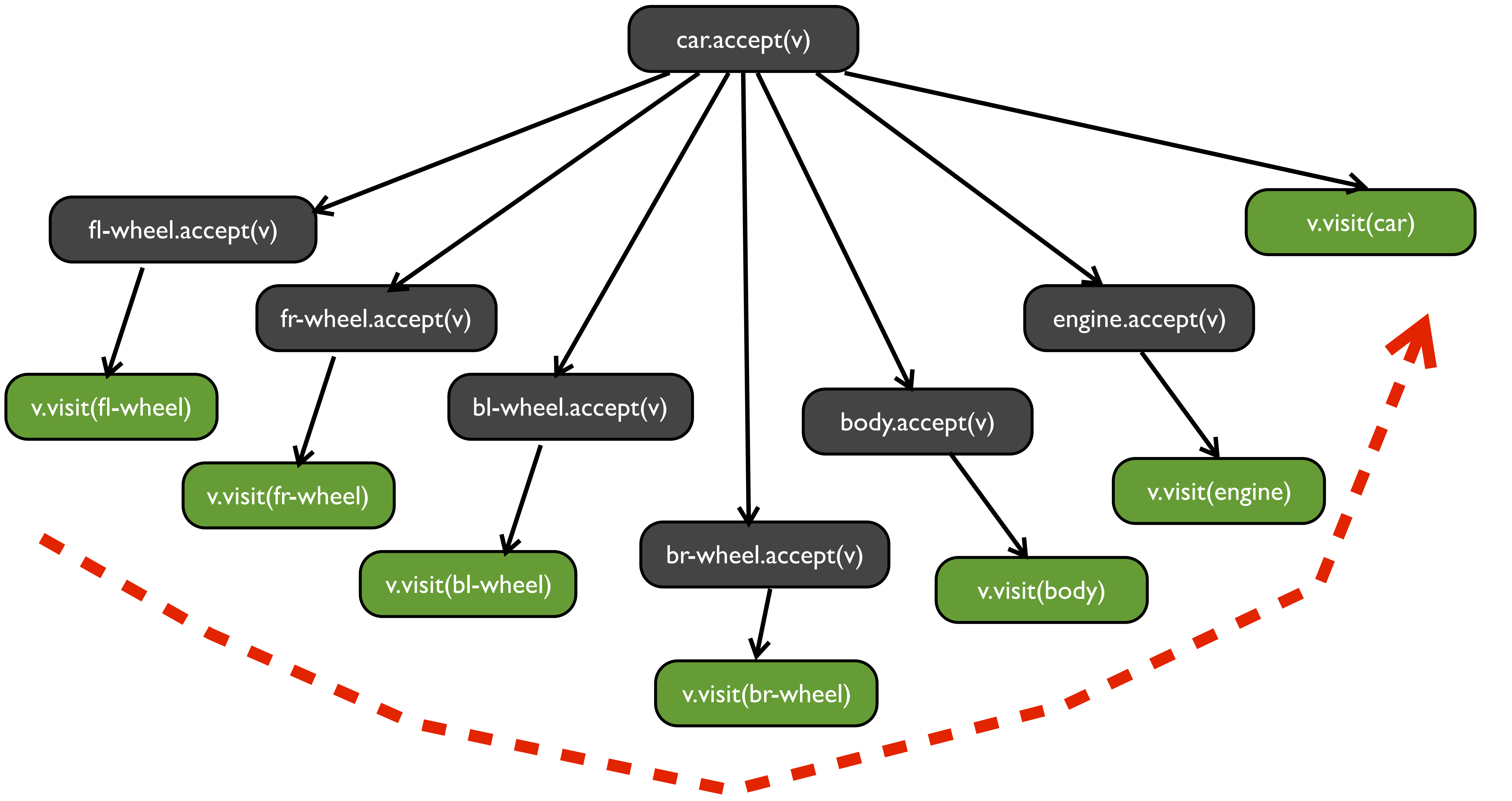
Action - Code

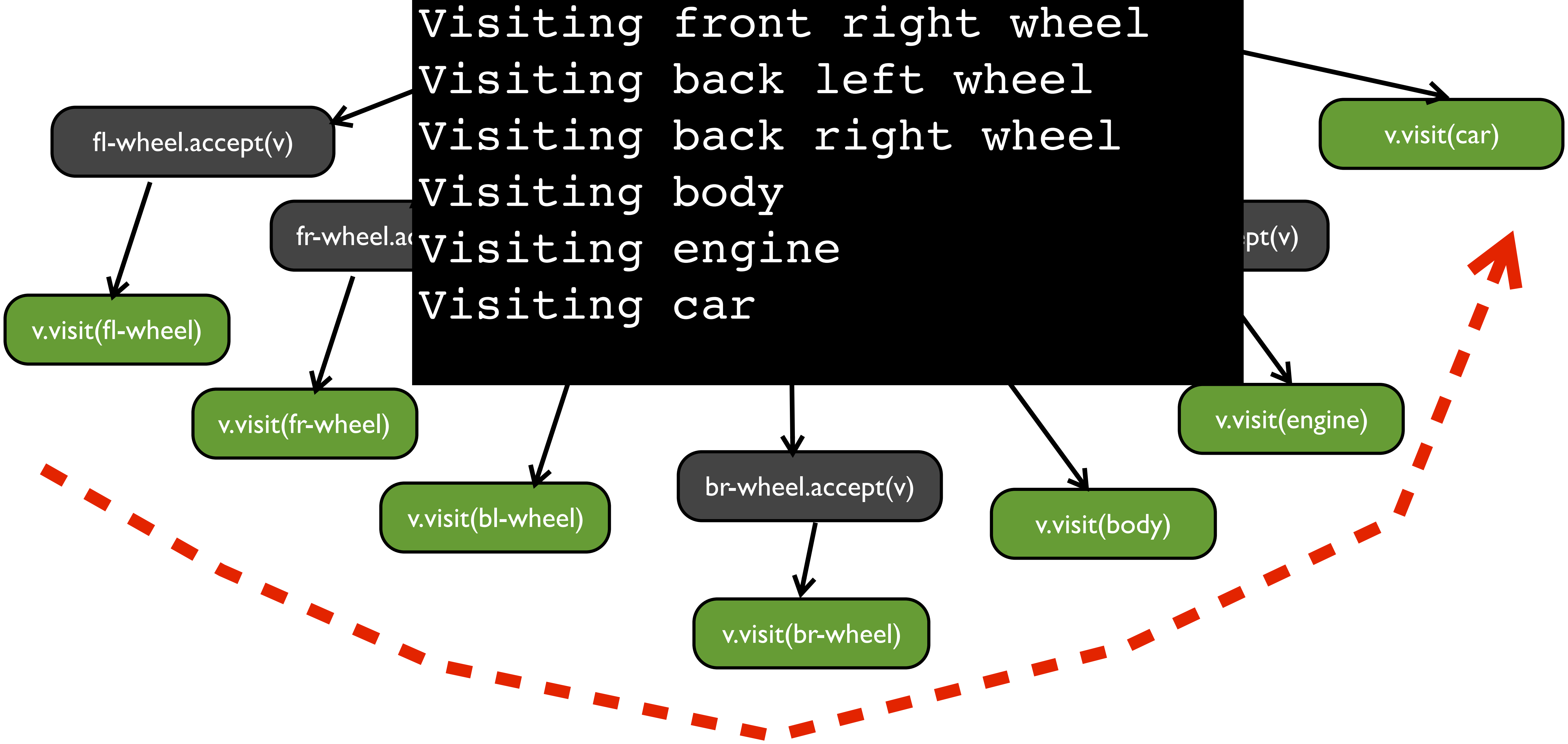
```
class CarElementPrintVisitor implements
ICarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting " + wheel.getName()
+ " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }

    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        ICarElement car = new Car();
        car.accept(new CarElementPrintVisitor());
    }
}
```

```
Visiting front left wheel
Visiting front right wheel
Visiting back left wheel
Visiting back right wheel
Visiting body
Visiting engine
Visiting car
```

OutPut


car.accept(v)
fl-wheel.accept(v)
v.visit(fl-wheel)
fr-wheel.accept(v)
v.visit(fr-wheel)
bl-wheel.accept(v)
v.visit(bl-wheel)
br-wheel.accept(v)
v.visit(br-wheel)
body.accept(v)
v.visit(body)
engine.accept(v)
v.visit(engine)
v.visit(car)



```
Visiting front left wheel
Visiting front right wheel
Visiting back left wheel
Visiting back right wheel
Visiting body
Visiting engine
Visiting car
```

Additional Action - Code

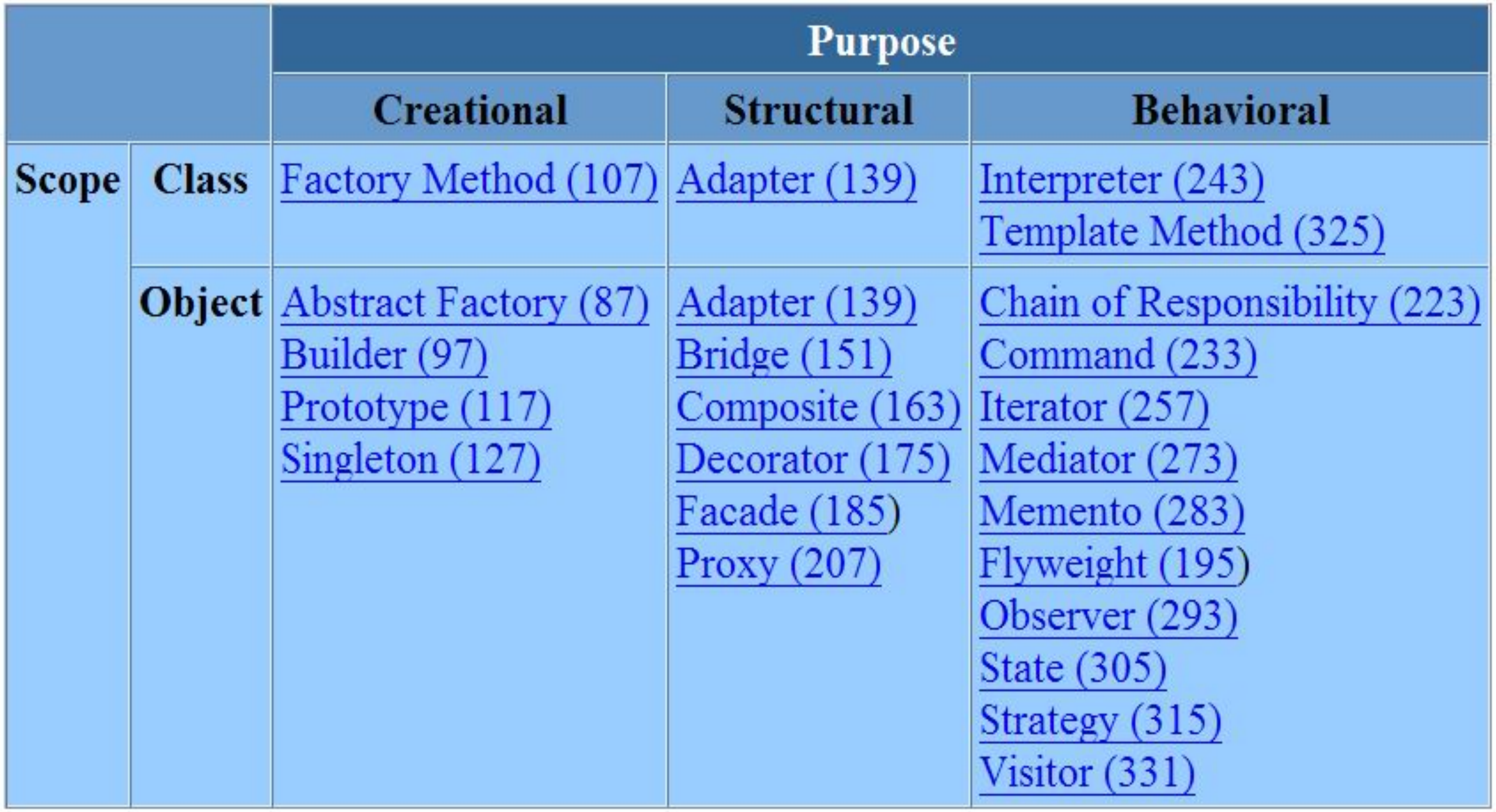
```java
class CarElementDoVisitor implements ICarElementVisitor
{
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " +
wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }

    public void visit(Body body) {
        System.out.println("Moving my body");
    }

    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        ICarElement car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

```
Visiting front left wheel
Visiting front right wheel
Visiting back left wheel
Visiting back right wheel
Visiting body
Visiting engine
Visiting car
Kicking my front left wheel
Kicking my front right wheel
Kicking my back left wheel
Kicking my back right wheel
Moving my body
Starting my engine
Starting my car
```

OutPut

Design Patterns



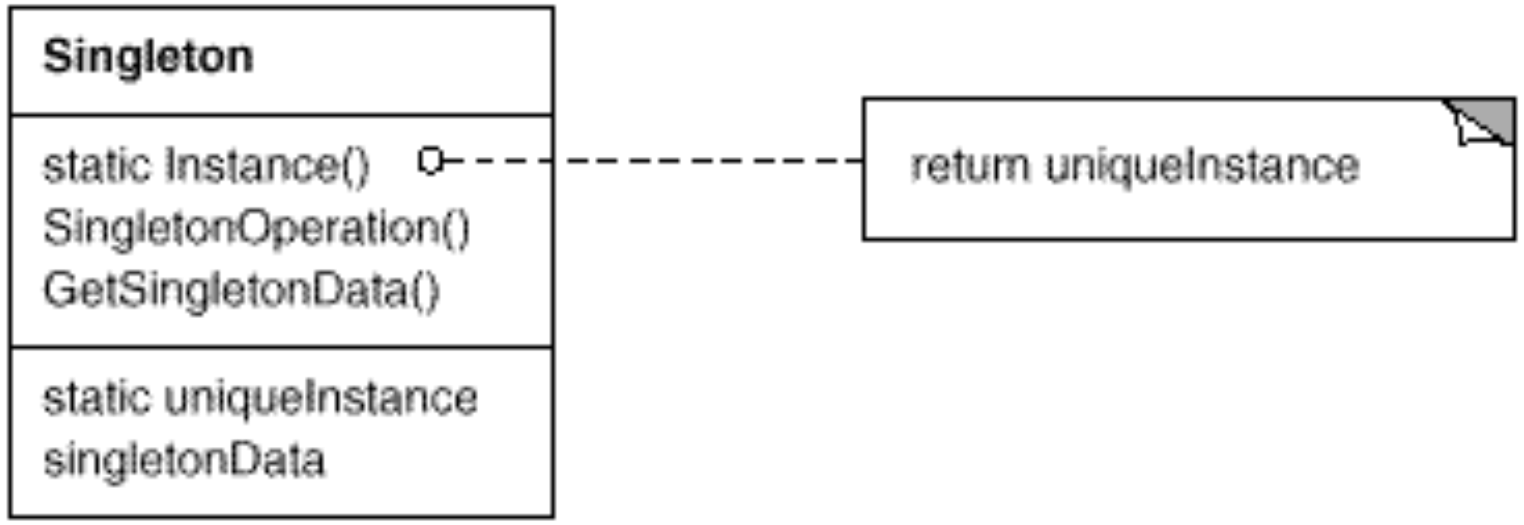
		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Singleton Pattern

- Ensure a class only has one instance, and provide a global point of access to it.
- It's important for some classes to have exactly one instance.
  - Although there can be many printers in a system, there should be only one printer spooler.
  - There should be only one file system and one window manager.

Singleton Pattern

- Class is responsible for tracking its sole instance
  - Make constructor private
  - Provide static method/field to allow access to the only instance of the class

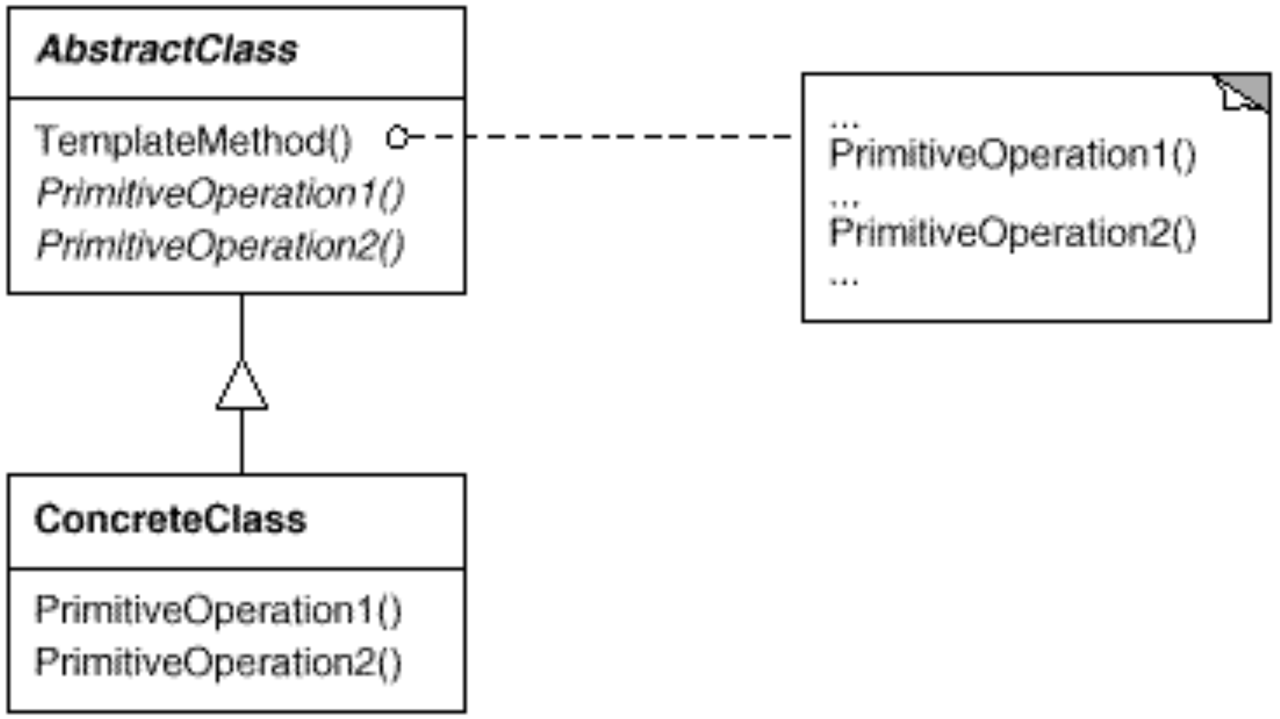


Template Method Pattern

- Problem
  - You're building a reusable class
  - You have a general approach to solving a problem,
  - But each subclass will do things differently

Template Method Pattern

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

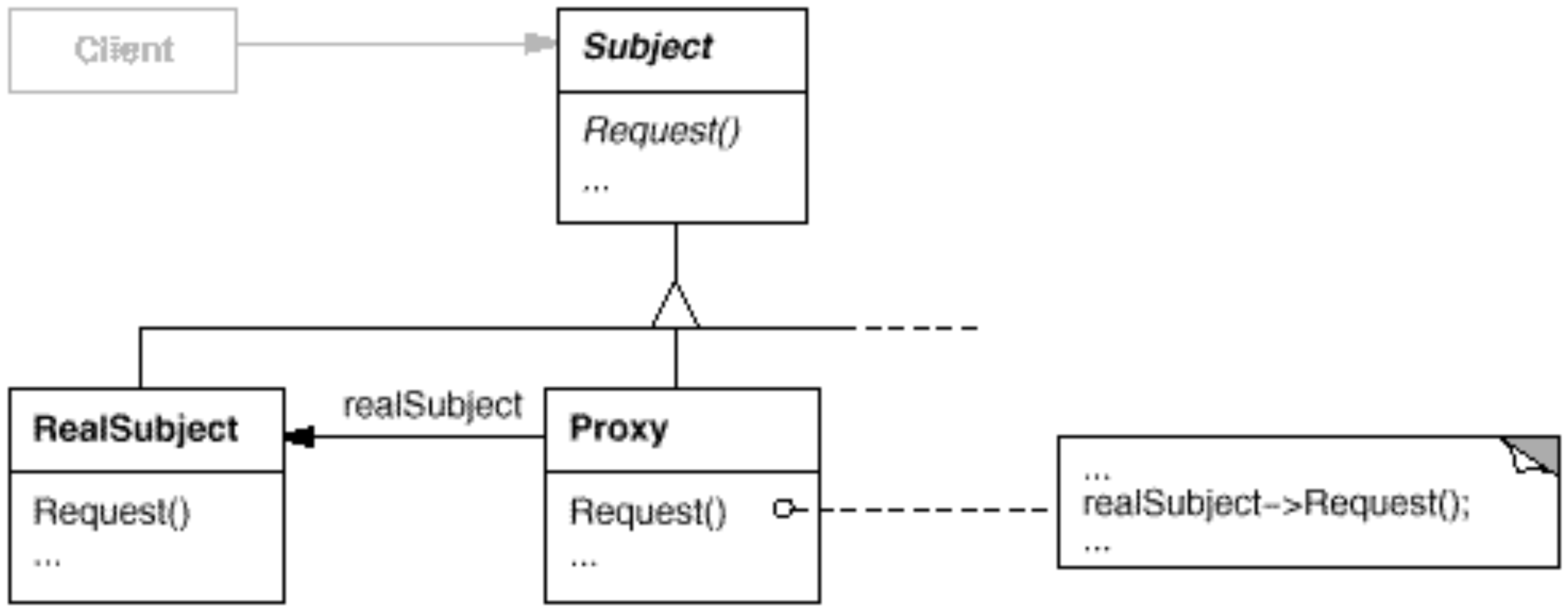


Proxy Pattern

- Goal:
  - Prevent an object from being accessed directly by its clients

Proxy Pattern

- Solution:
  - Use an additional object, called a proxy
  - Clients access to protected object only through proxy
  - Proxy keeps track of status and/or location of protected object


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...