

How can we design good quality software?

# Why Patterns?



What are patterns?

Where to find them?



# What are patterns anyway?

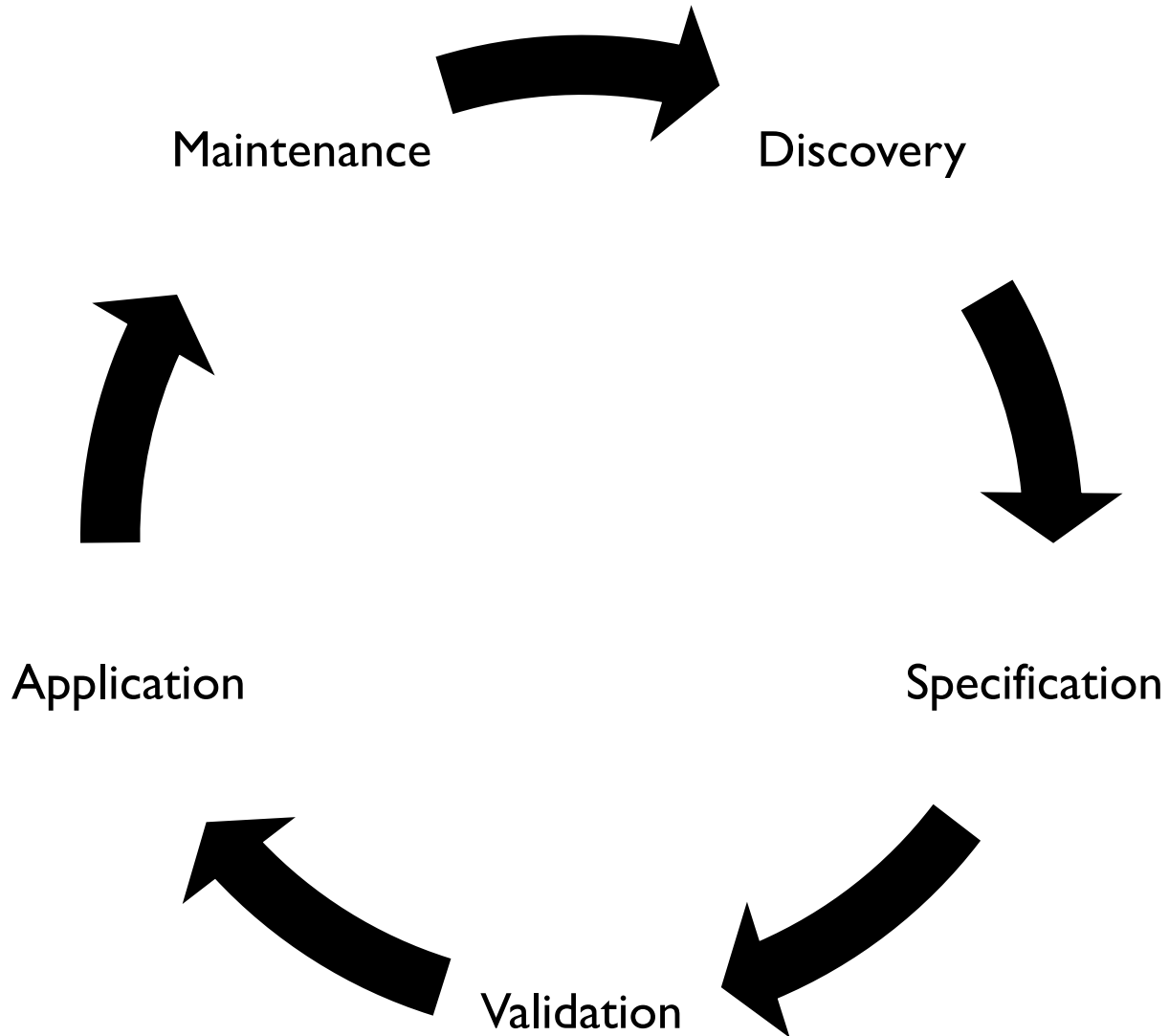


Nature

Man-made

Software?

# Patterns Life Cycle



# Basic Concepts

- *Context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.
- A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how
  - the problem can be interpreted within its context and
  - how the solution can be effectively applied.

# Design patterns (GOF)

- Design Patterns communicate solutions to common problems.
  - It's a problem-solution pair.
- The seminal book on design patterns, *Design Patterns, Elements of Reusable Object-Oriented Software* by Gamma et al, identifies three categories of design patterns
  - Creational
  - Structural
  - Behavioral

# Design for Software Quality

- **Creational patterns** focus on the “creation, composition, and representation of objects, e.g.,
  - **Singleton pattern**: Control the creation of instances to just one
  - **Abstract factory pattern**: centralize decision of what **factory** to instantiate
  - **Factory method pattern**: centralize creation of an object of a specific type choosing one of several implementations
- **Structural patterns** focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,
  - **Adapter pattern**: 'adapts' one interface for a class into one that a client expects
  - **Aggregate pattern**: a version of the **Composite pattern** with methods for aggregation of children
- **Behavioral patterns** address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects, e.g.,
  - **Chain of responsibility pattern**: Command objects are handled or passed on to other objects by logic-containing processing objects
  - **Command pattern**: Command objects encapsulate an action and its parameters
  - **Observer pattern**: Enable loose coupling between publishers and subscribers

# THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

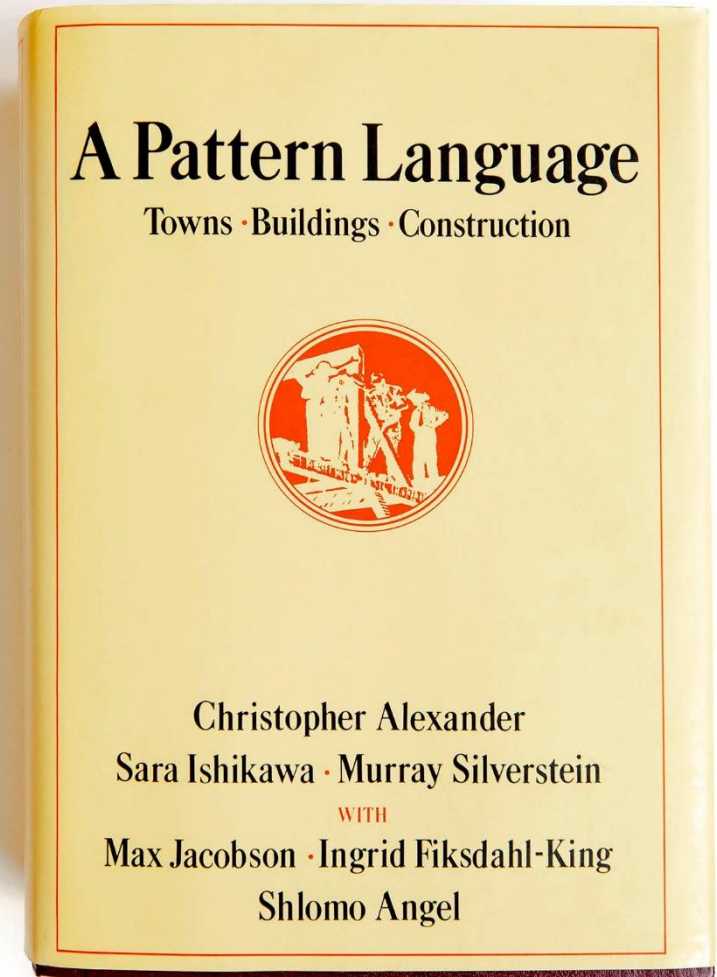


# Describing a Pattern

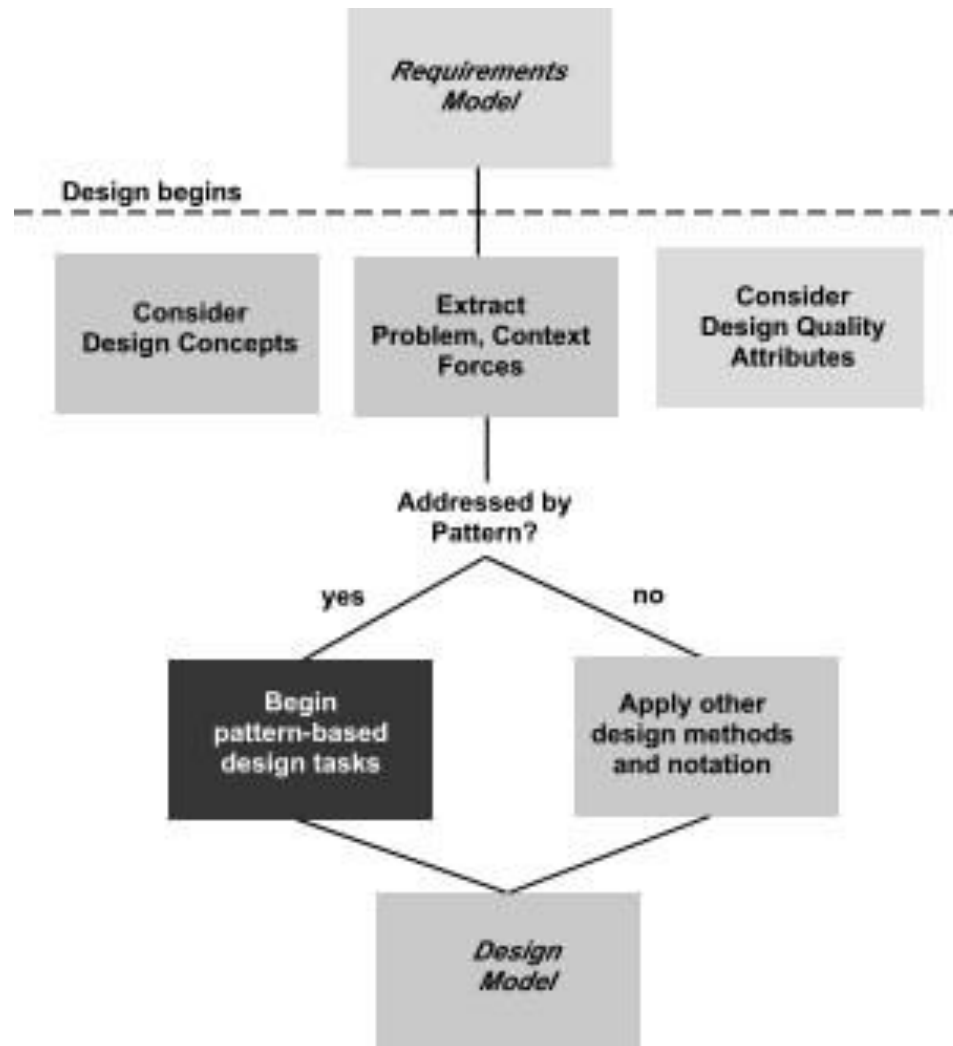
- **Pattern name**—describes the essence of the pattern in a short but expressive name
- **Problem**—describes the problem that the pattern addresses
- **Motivation**—provides an example of the problem
- **Context**—describes the environment in which the problem resides including application domain
- **Forces**—lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitation and constraints that must be considered
- **Solution**—provides a detailed description of the solution proposed for the problem
- **Intent**—describes the pattern and what it does
- **Collaborations**—describes how other patterns contribute to the solution
- **Consequences**—describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern
- **Implementation**—identifies special issues that should be considered when implementing the pattern
- **Known uses**—provides examples of actual uses of the design pattern in real applications
- **Related patterns**—cross-references related design patterns

# Pattern Languages

- A *pattern language* encompasses a collection of patterns
  - each described using a standardized template and
  - interrelated to show how these patterns collaborate to solve problems across an application domain.
- a pattern language is analogous to a hypertext instruction manual for problem solving in a specific application domain.
  - The problem domain under consideration is first described hierarchically, beginning with broad design problems associated with the domain and then refining each of the broad problems into lower levels of abstraction.



# Pattern-Based Design



# Design Patterns

## Creational

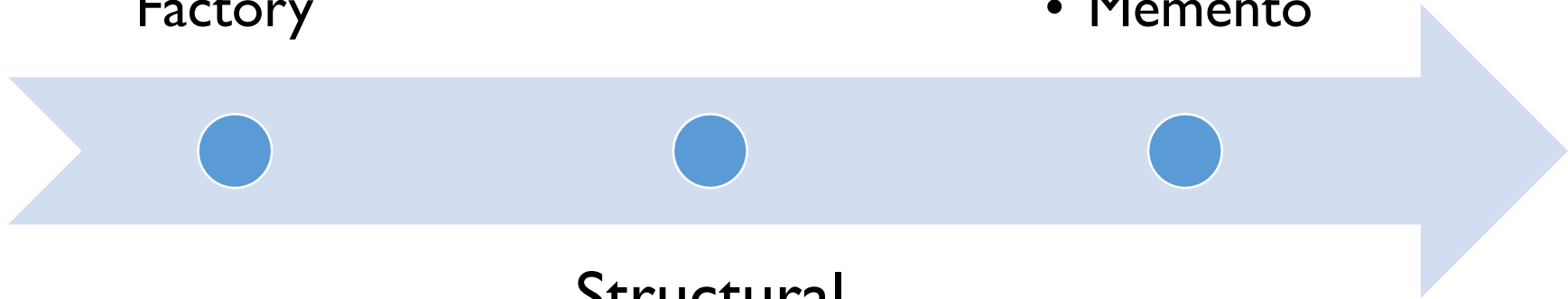
- Factory
- Abstract Factory

## Behavioral

- Memento

## Structural

- Flyweight



# Factory pattern

# 46 | Pizza Store

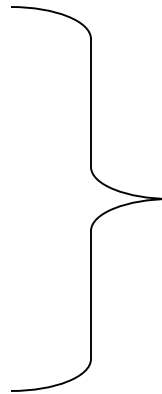
```
public class PizzaStore{  
    Pizza orderPizza() {  
  
        Pizza pizza = new Pizza();  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}
```

# If you need more than one type...

```
Pizza orderPizza(String type) {  
    Pizza pizza ;  
  
    if (type.equals("cheese")) {  
        pizza = new CheesePizza();  
    } else if (type.equals("greek")) {  
        pizza = new GreekPizza();  
    } else if (type.equals("pepperoni")) {  
        pizza = new PepperoniPizza();  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

# still more...

```
Pizza orderPizza(String type) {  
    Pizza pizza;  
    if (type.equals("cheese")) {  
        pizza = new CheesePizza();  
    } else if (type.equals("greek")) {  
        pizza = new GreekPizza();  
    } else if (type.equals("pepperoni")) {  
        pizza = new PepperoniPizza();  
    } else if (type.equals("chicken")) {  
        pizza = new ChickenPizza();  
    } else if (type.equals("veggie")) {  
        pizza = new VeggiePizza();  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```



1. Dealing with which concrete class is being instantiated

2. Changes are preventing orderPizza() from being closed for modification



# Building a simple Pizza factory

```
Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

```
public class SimplePizzaFactory {  
    public Pizza createPizza (String type) {  
        Pizza pizza = null;  
  
        if (type.equals("cheese")) {  
            pizza = new CheesePizza();  
        }  
        else if (type.equals("pepperoni")) {  
            pizza = new PepperoniPizza();  
        }  
        else if (type.equals("chicken")) {  
            pizza = new ChickenPizza();  
        }  
        else if (type.equals("veggie")) {  
            pizza = new VeggiePizza();  
        }  
    }  
}
```

# Dumb Question...is it really?

Hmmm...

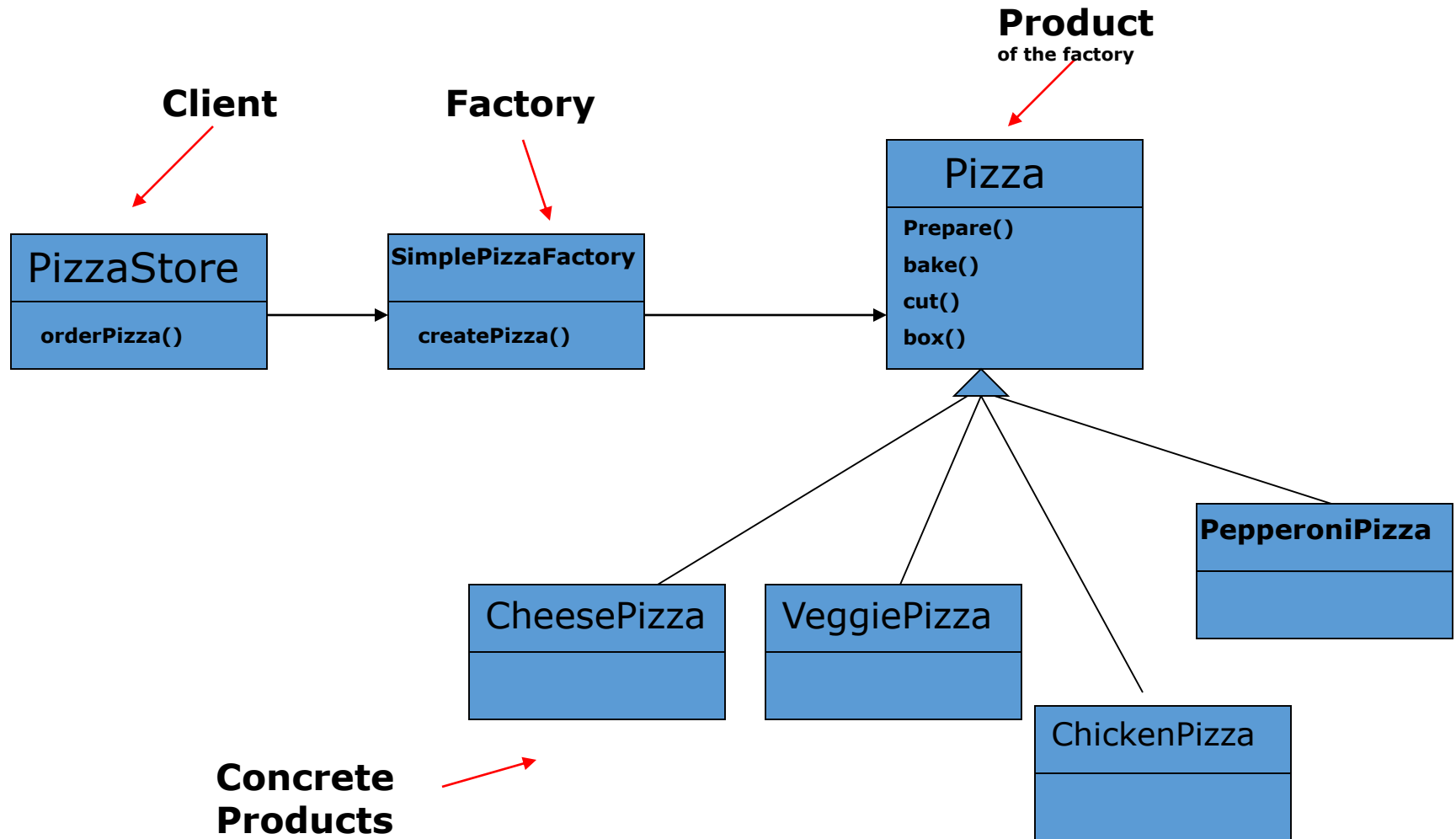
What's the advantage of this?

1. Simple pizza factory may have many clients that use the creation in different ways
2. Easy to remove concrete instantiations from the client code

# Changing the client class

```
public class PizzaStore {  
    SimplePizzaFactory factory;  
  
    public PizzaStore (SimplePizzaFactory factory) {  
        this.factory = factory;  
    }  
  
    public Pizza orderPizza (String type) {  
        Pizza pizza;  
  
        pizza = factory.createPizza(type);  
  
        pizza.prepare()  
        pizza.bake()  
        pizza.cut()  
        pizza.box()  
        return pizza;  
    }  
}
```

# Simple Factory



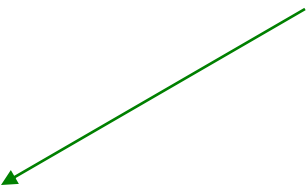
# Revisiting - Pizza Store

```
public class PizzaStore{  
    Pizza orderPizza() {  
  
        Pizza pizza = new Pizza();  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}
```

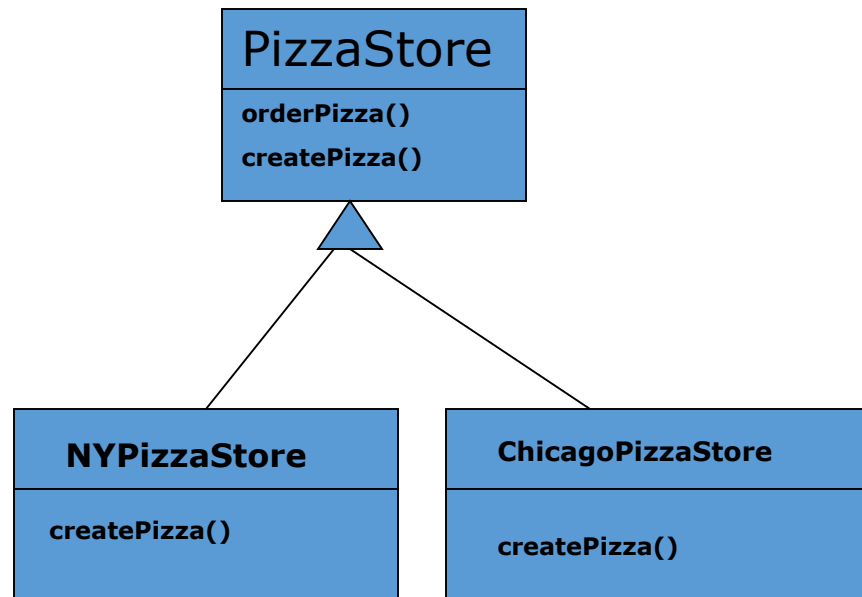
# A framework for the pizza store

```
public abstract class PizzaStore {  
  
    public Pizza orderPizza (String type) {  
        Pizza pizza;  
  
        pizza = createPizza(type);  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
  
        abstract Pizza createPizza (String type);  
    }  
}
```

Factory  
method is  
now  
abstract



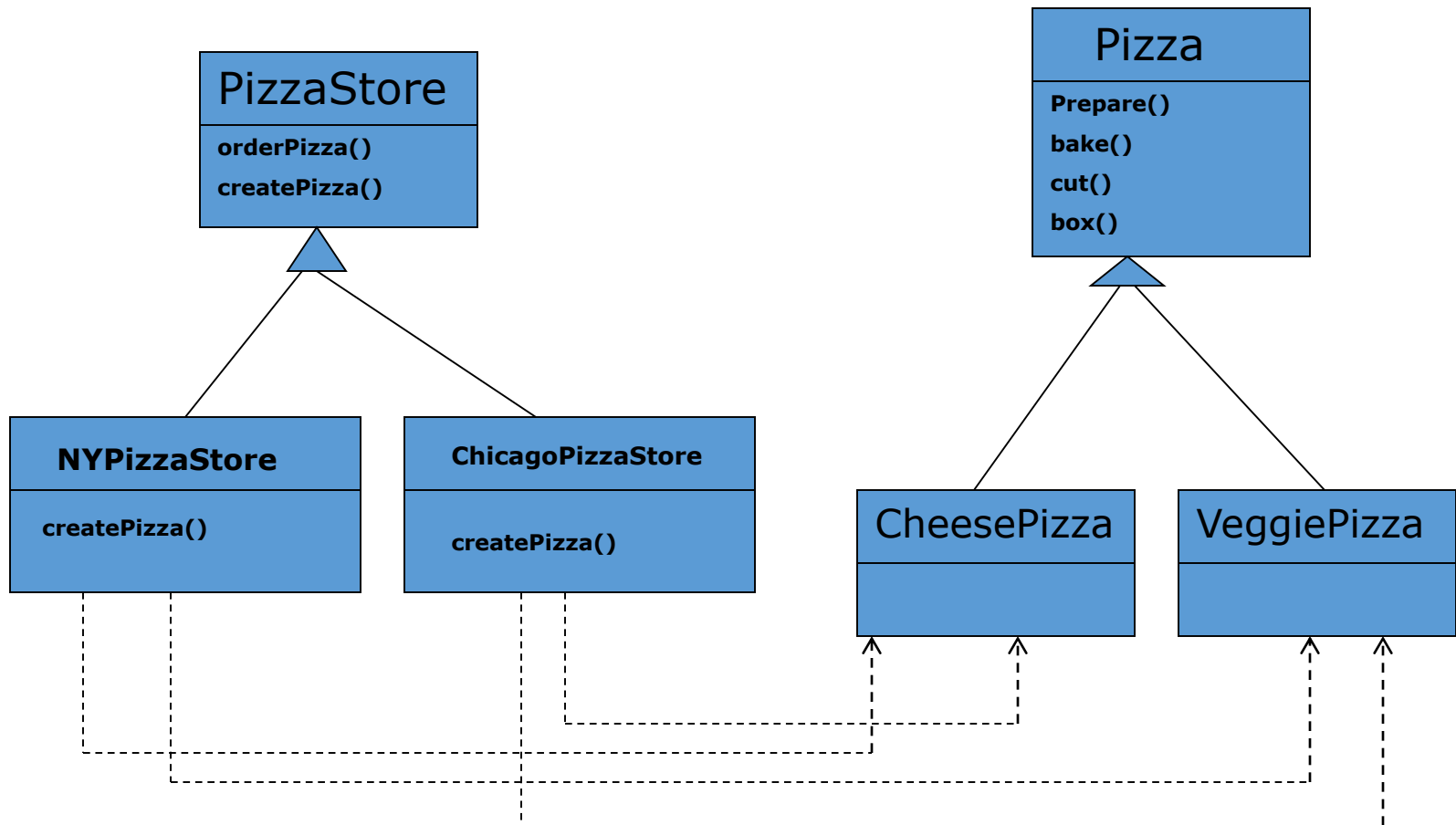
# Allowing the subclasses to decide (creator classes)



# Factory method

Creator classes

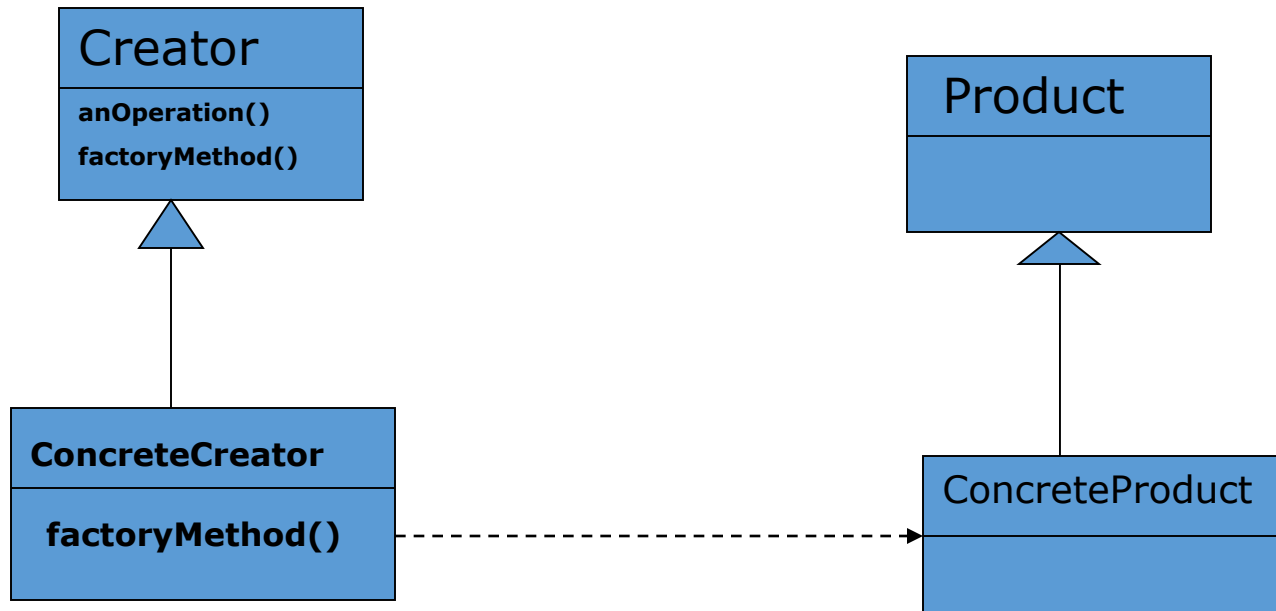
Product classes





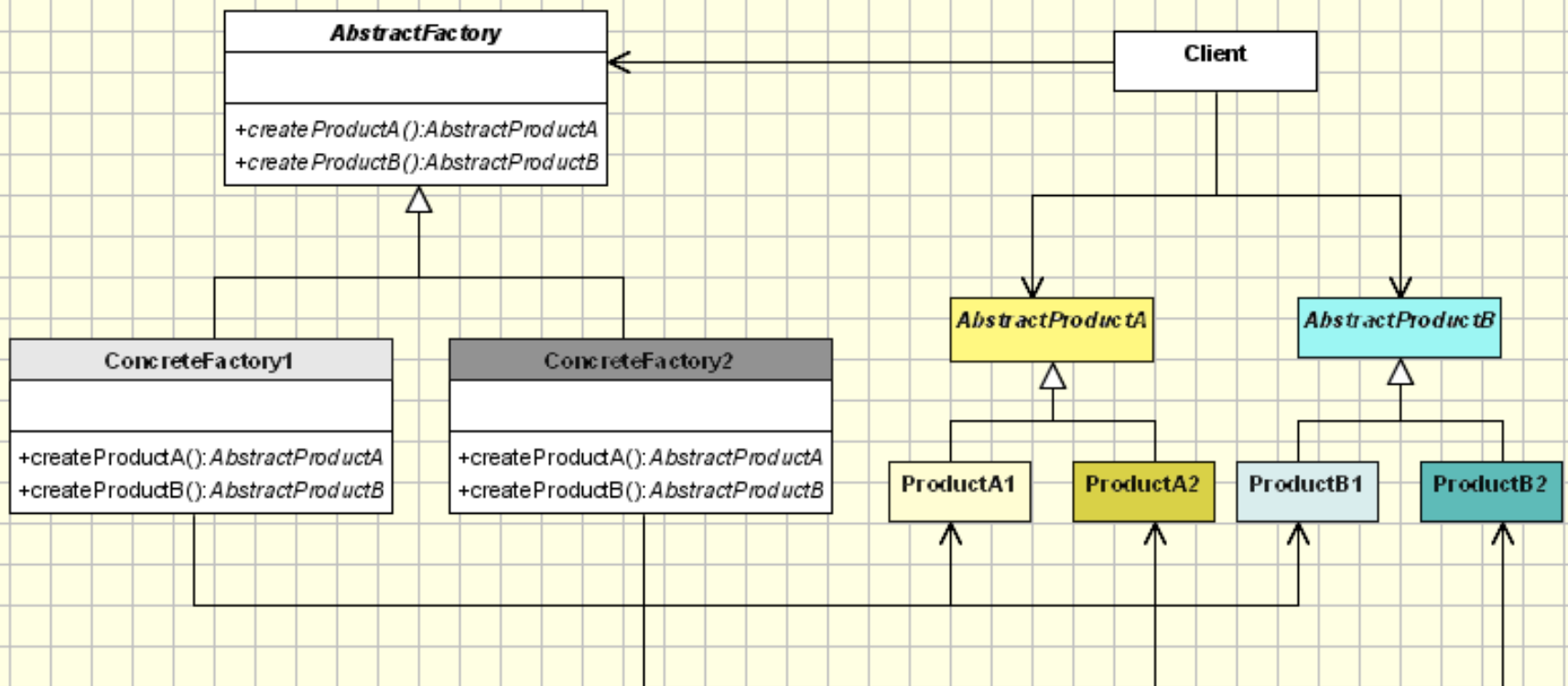
# Factory method defined

- Factory method pattern defines an interface for creating an object, but lets the subclasses decide which class to instantiate. Factory method lets a class *defer instantiation to subclasses*



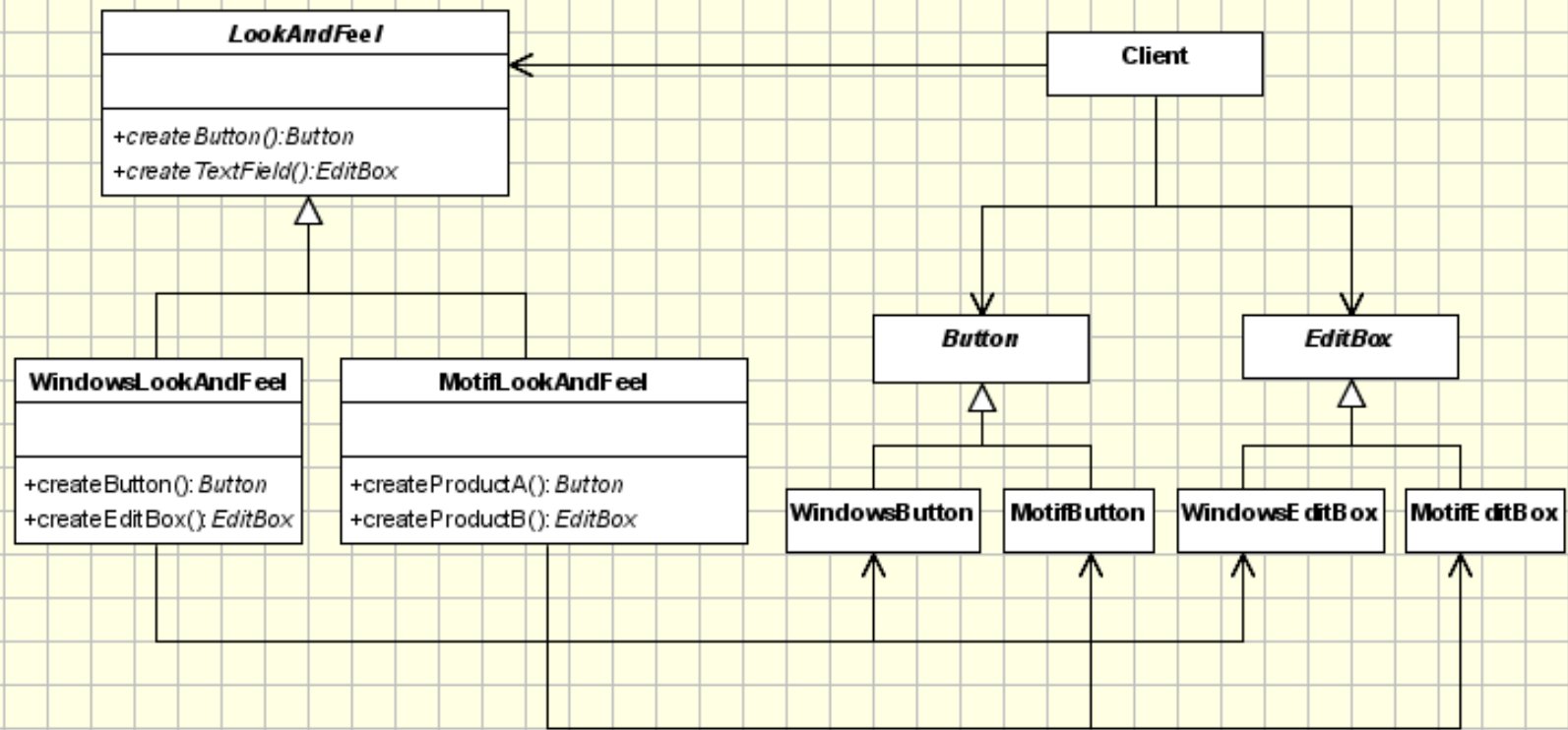
# Abstract Factory defined

cd: Abstract Factory Implementation - UML Class Diagram



# Look & Feel Example

cd: Abstract Factory - Look & Feel Example - UML Class Diagram



# Applicability

- Examples
  - Create different cars
  - Create different file types
  - Create different t-shirts
  - Create different database connections
- Core Idea: Factory patterns separate and defer object creation process

# How does factory pattern improve our design?

- Dependency Inversion Principle - “Depend upon abstractions. Do not depend upon concrete classes.”
  - No variable should hold a reference to a concrete class
  - No class should derive from a concrete class
  - No method should override an implemented method of its base classes

# Limitations

- *Refactoring* an existing class to use factories breaks existing clients.
- *Re-implementation* of factory methods enforces same *signatures*
- The factory method pattern relies on inheritance via sub classes
- Whereas Abstract Factory pattern delegates the responsibility to another object via composition

# Creational Patterns

- Defer instantiation of objects
- Creational patterns abstract the object instantiation process.
- They hide how objects are created and help make the overall system independent of how its objects are created and composed.

# Memento Pattern



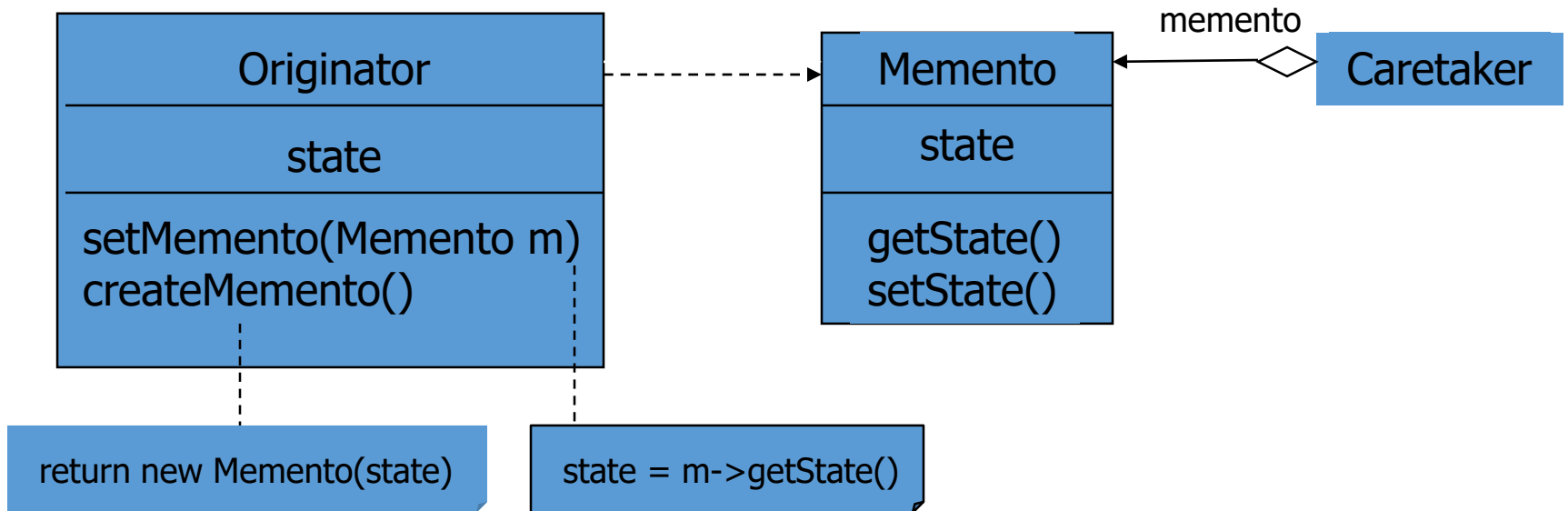
# The Problem

- Problem
  - Sometimes state of an object must be restored to a previous state by a client
- Desire
  - Preserve encapsulation of state privacy
- Solution
  - Have object create a **Memento** of its current state for later restoration

# Applicability

- You need to checkpoint the state of an object
- You do not want to expose the internal structure of the object
  - **Memento** is opaque to client
- Mainly in cases of error/failure to do an Undo operation

# Pattern Structure



# Participants

- Memento
  - Stores part or all of internal state
  - Ideally protects access by non-originator objects
- Originator
  - Creates snapshot of its state in Memento
  - Restores its state from Memento
- Caretaker
  - Holds Memento for later restoration of state of originator

# Consequences

- Encapsulation is maintained
  - Clients manage state w/o knowing details
  - Simplifies originator
- Mementos can be expensive / large
  - How much state must be saved?
    - Full state?
    - A difference / delta from a base state?
  - If not cheap to save state is it appropriate?
- Caretaker has to manage memento adding additional overhead

# Flyweight pattern

# When do we use it?

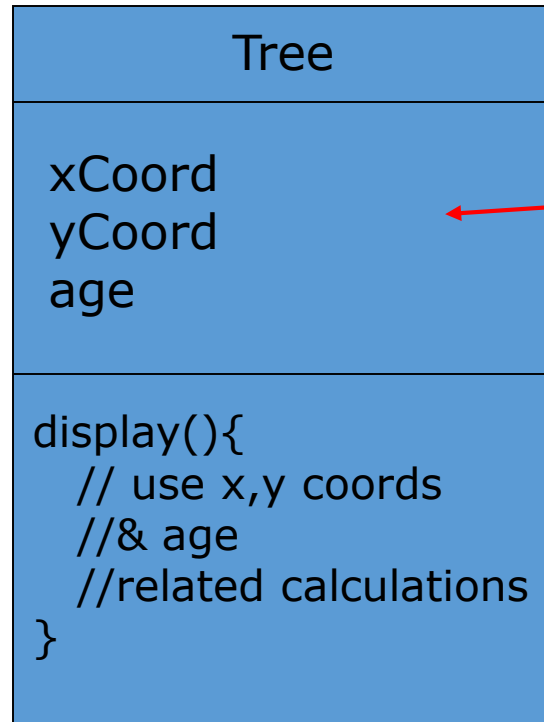
- When one instance of a class can be used to provide many “virtual instances”

# Example scenario

- Wish to add trees as objects in a landscape design
- They just contain x,y location and draw themselves dynamically depending on the age of the tree
- User may wish to have lots of trees in a particular landscape design



# Tree class

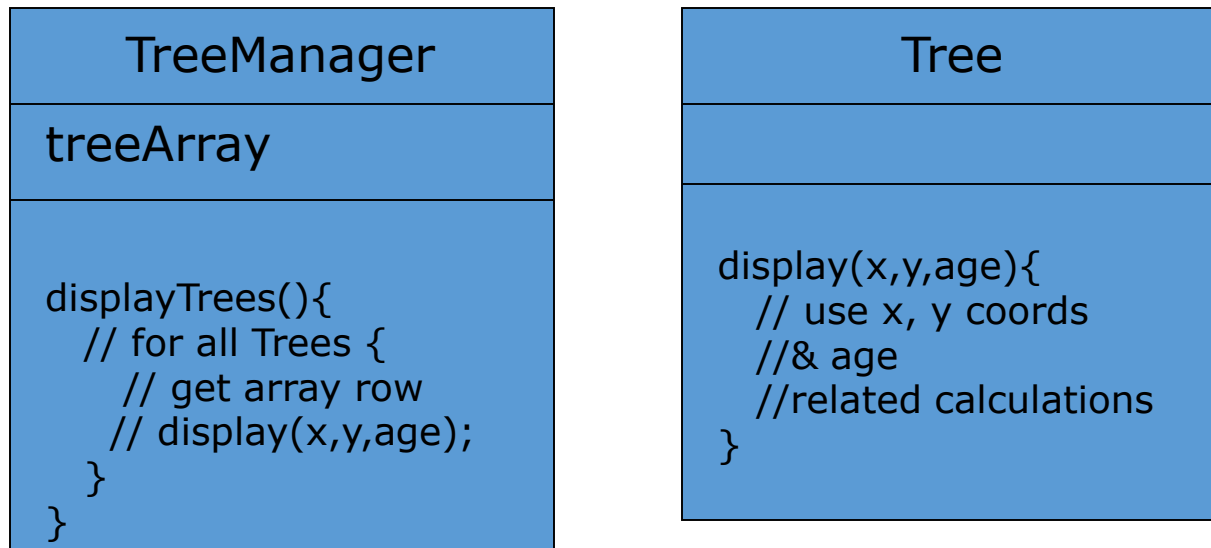


*Each Tree instance maintains its own state*

What happens if 100000 tree objects are created?

# Flyweight pattern

- If there is only one instance of Tree and client object maintains the state of ALL the Trees, then it's a **flyweight**



# Benefits & Drawbacks

## Benefits:

- Reduces the number of object instances at runtime, saving memory
- Centralizes state for many “virtual” objects into a single location

## Drawbacks:

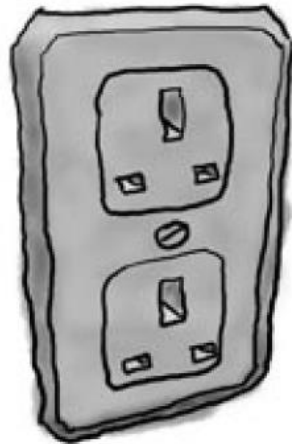
- Once a flyweight pattern is implemented, single logical instances of the class will not be able to behave independently from other instance.

# Adapter pattern

*Acknowledgement: Head-first Design patterns. Freeman & Freeman*

# Example Scenario

**European Wall Outlet**



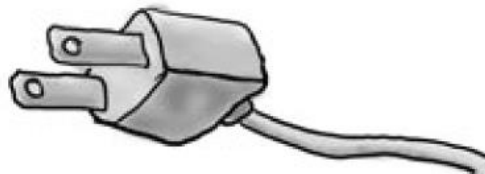
*The European wall outlet exposes one interface for getting power*

**AC Power Adapter**



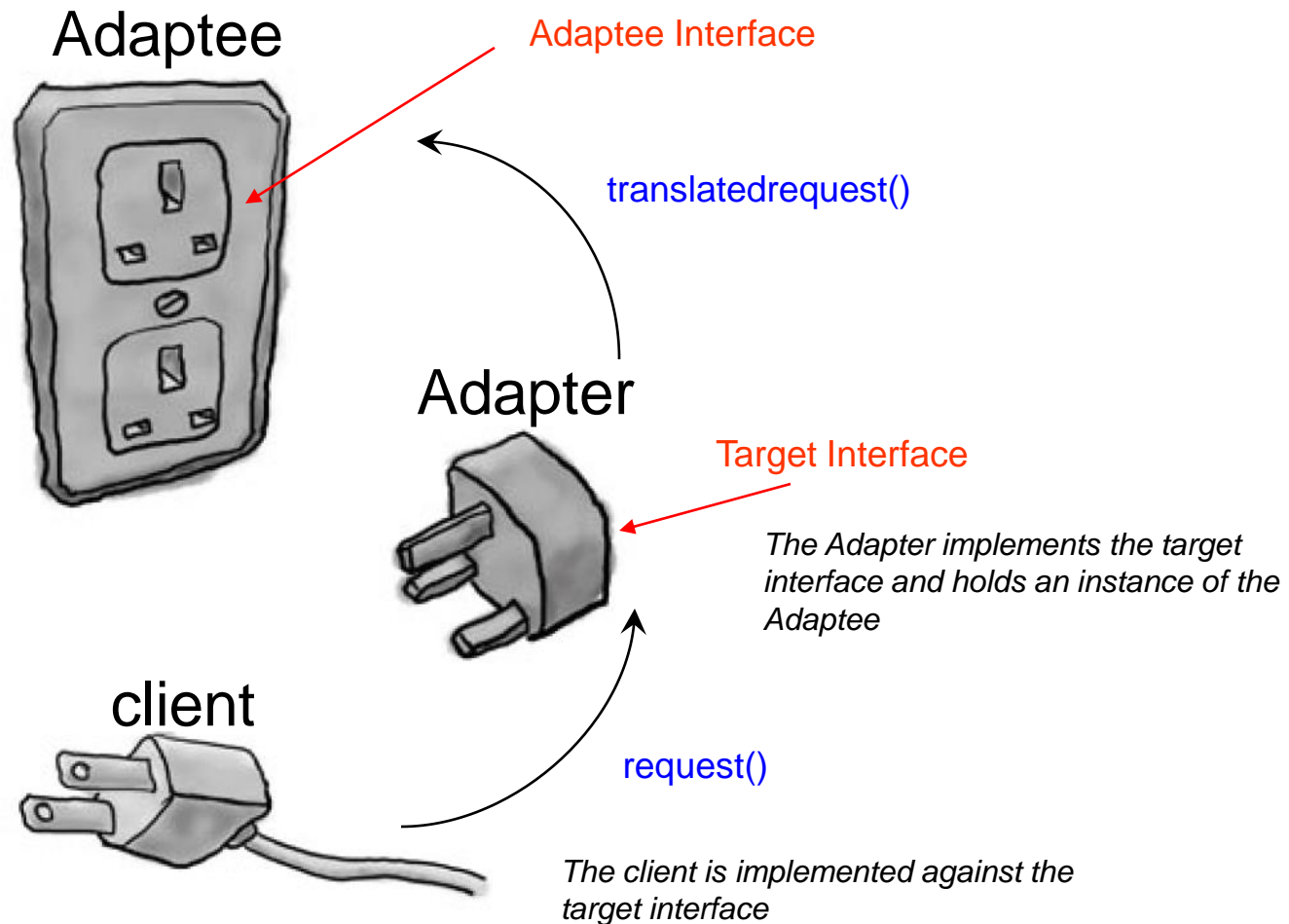
*The adapter converts one interface into another*

**Standard AC Plug**



*The US laptop expects another interface*

# Adapter Pattern Explained



# Adapter Pattern

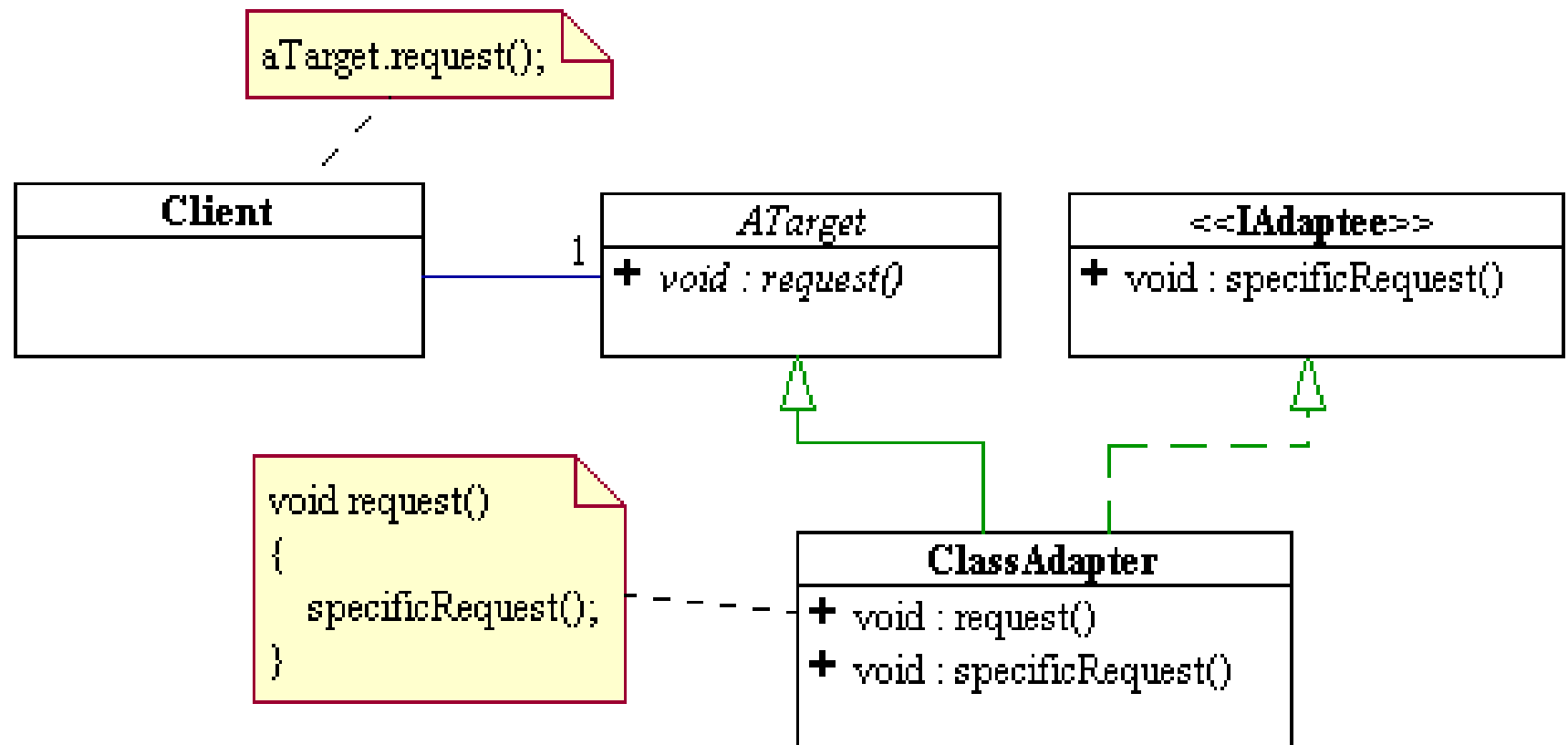
- An adapter pattern converts the interface of a class into an interface that a client expects
- Adapters allow incompatible classes to work together
- Adapters can extend the functionality of the adapted class
- Commonly called “glue” or “wrapper”

# When to Use

- Need to adapt the interface of an existing class to satisfy client interface requirements
  - Adapting Legacy Software
  - Adapting 3<sup>rd</sup> Party Software



# Class Adapter Pattern



# When to Use Adapters

- Concrete adapter

When using a class whose interface does not match what you need.

- Abstract adapter

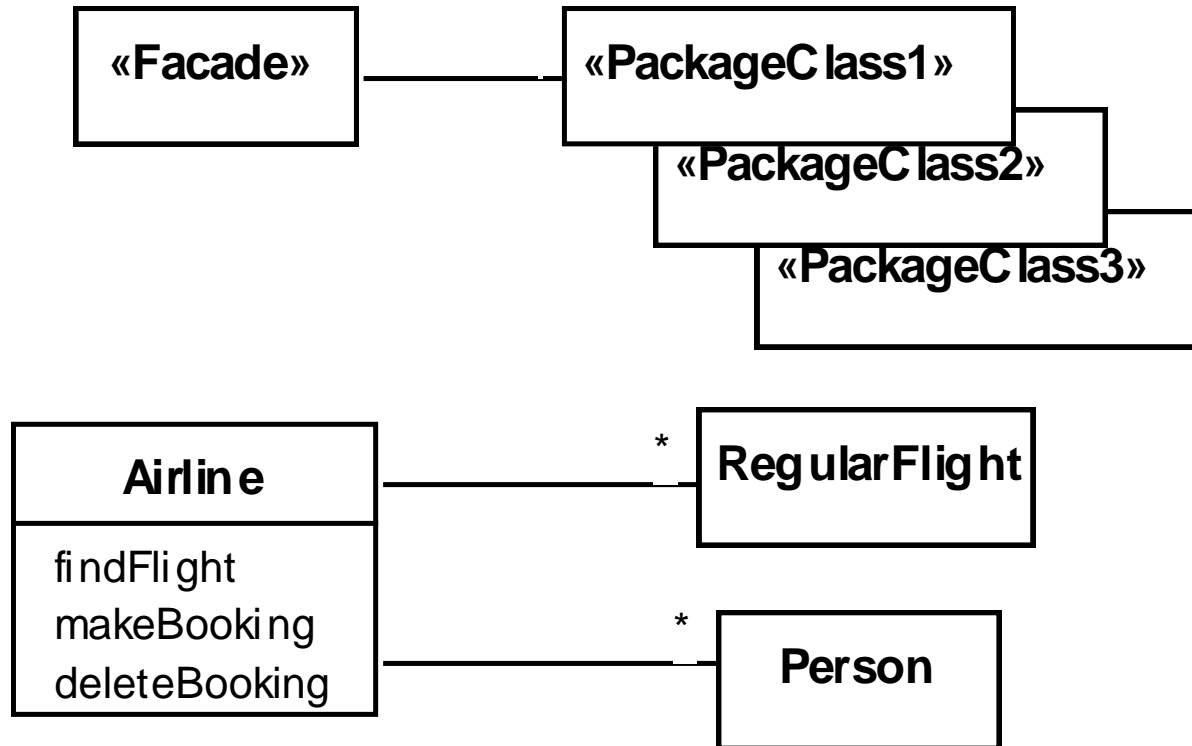
When creating a reusable class that cooperates with unknown future classes.

# The Controller Façade Pattern

- **Context:**
  - Often, an application contains several complex packages.
  - A programmer working with such packages has to manipulate many different classes
- **Problem:**
  - How do you simplify the view that programmers have of a complex package?
- **Forces:**
  - It is hard for a programmer to understand and use an entire subsystem
  - If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

# Façade

- **Solution:**



# Proxies/Adapters/Facades

- Proxies and Adapters both place a stand-in object between the client and the real object
- Adapters do so to change the real object's interface
- Proxies do so to optimize access to the object via the same interface.
- Facades ease the use of sub-systems of objects

# Evaluating Designs

- The application of “well-known” design patterns that promote loosely coupled, highly cohesive designs.
- Conversely, identify the existence of recurring *negative* solutions – AntiPatterns
- AntiPattern : use of a pattern in an inappropriate context.
- Refactoring : changing, migrating an existing solution (antipattern) to another by improving the structure of the solution.

# What we learnt today?

- On Patterns
- Factory/Abstract Factory
  - Virtual Constructor - Defer instantiation
- Flyweight – Virtual Instances
- Patterns encapsulate guidelines, best practices and experience
- Patterns of Patterns
- High level patterns
- Trade offs are critical
- How do patterns fit together? – A Pattern Language

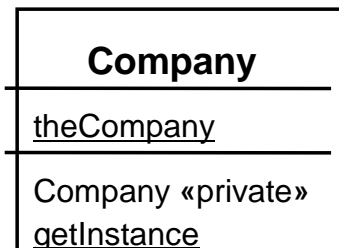
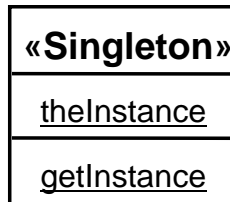
# The Singleton Pattern

- **Context:**
  - It is very common to find classes for which only one instance should exist (*singleton*)
- **Problem:**
  - How do you ensure that it is never possible to create more than one instance of a singleton class?
- **Forces:**
  - The use of a public constructor cannot guarantee that no more than one instance will be created.
  - The singleton instance must also be accessible to all classes that require it



# Singleton

- ***Solution:***



```
if (theCompany==null)
    theCompany= new Company();

return theCompany;
```

# Proxy pattern

*Acknowledgement: Freeman & Freeman*

# The Problems

- Expensive & inexpensive pieces of state
  - Example: Large image
  - Inexpensive: size & location of drawing
  - Expensive: load & display
- Remote objects (e.g., another system)
  - Want to access it as if it were local
  - Want to hide all the required communications
  - Example: Java RMI
- Object with varying access rights
  - Some clients can access anything
  - Other clients have subset of functionality available

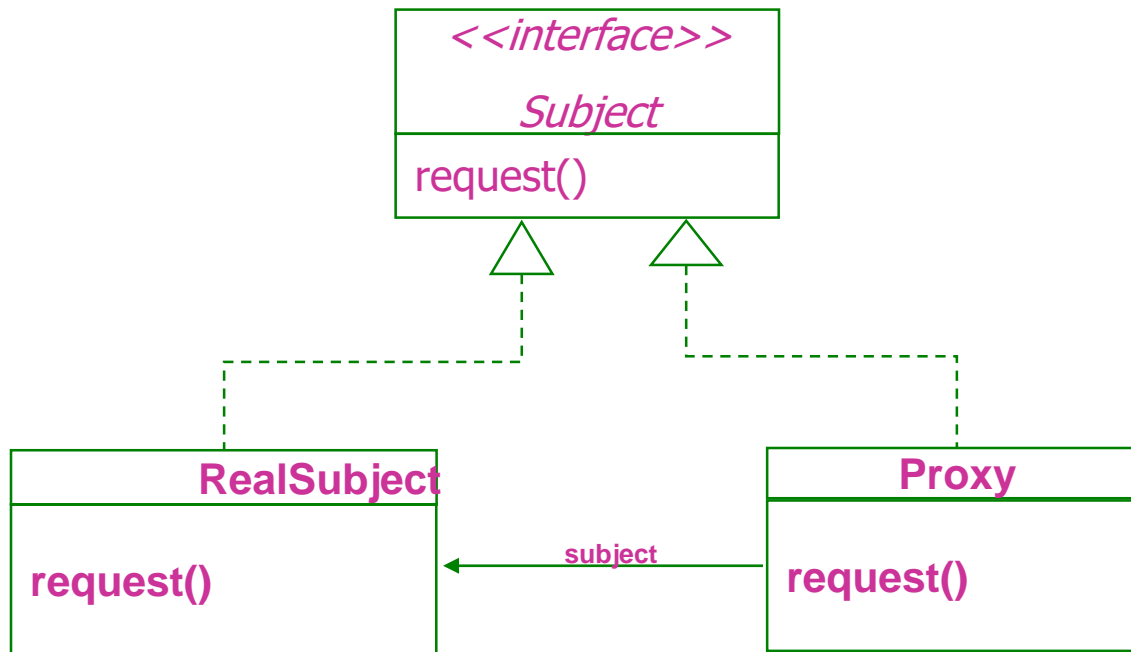
# The Design Goal

- In all these cases desire access to object as if it is directly available
- For efficiency, simplicity, or security, put a *proxy* object in front of the real object
- We have a stand-in for the real object to control how the real object behaves

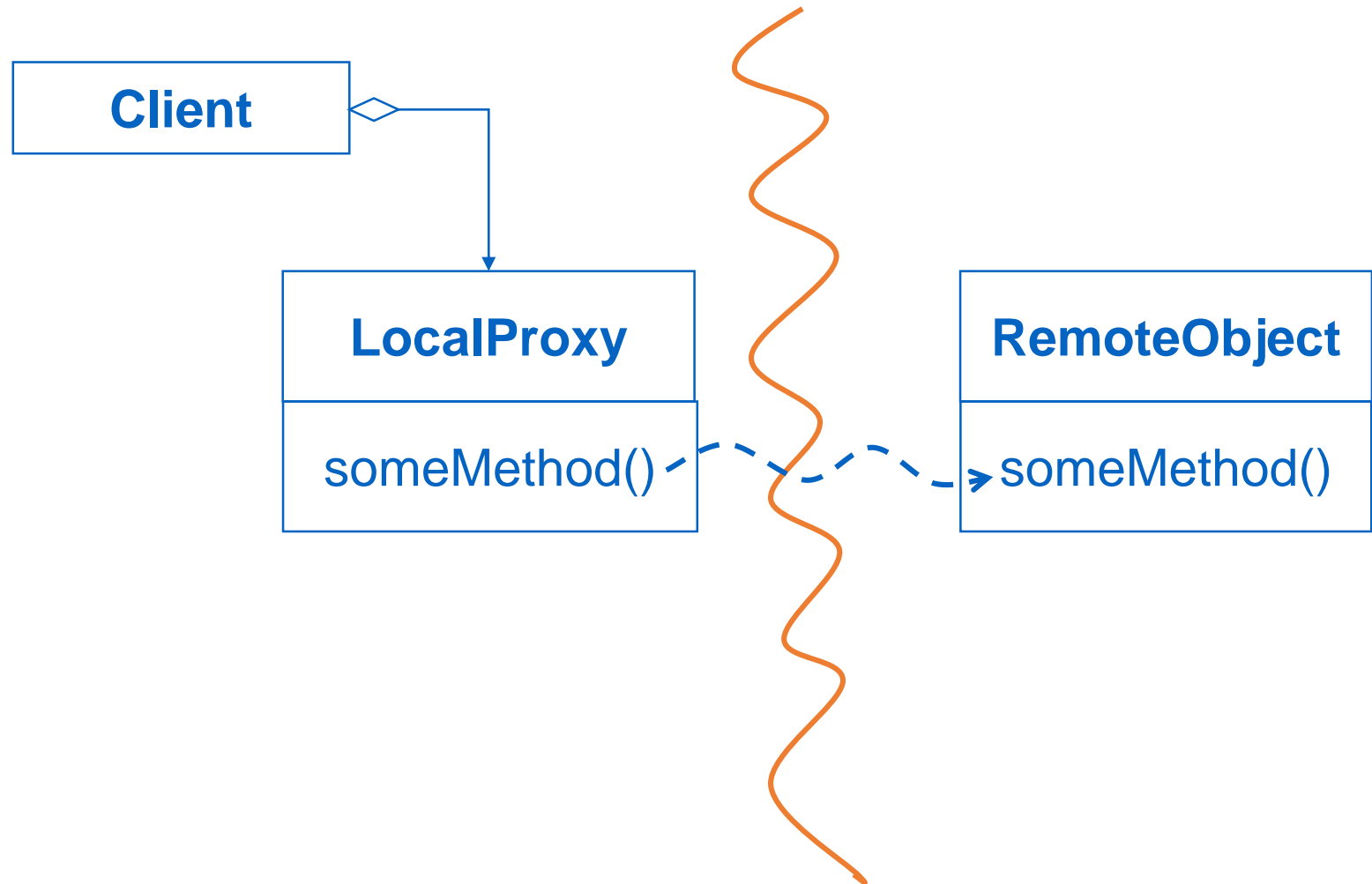
# Proxy pattern Defined

- Proxy patterns provides a surrogate or placeholder for another object to control access to it
  - **Remote proxy** controls access to a remote object
  - **Virtual proxy** controls access to a resource that is expensive to create
  - **Protection proxy** controls access to a resource based on access rights

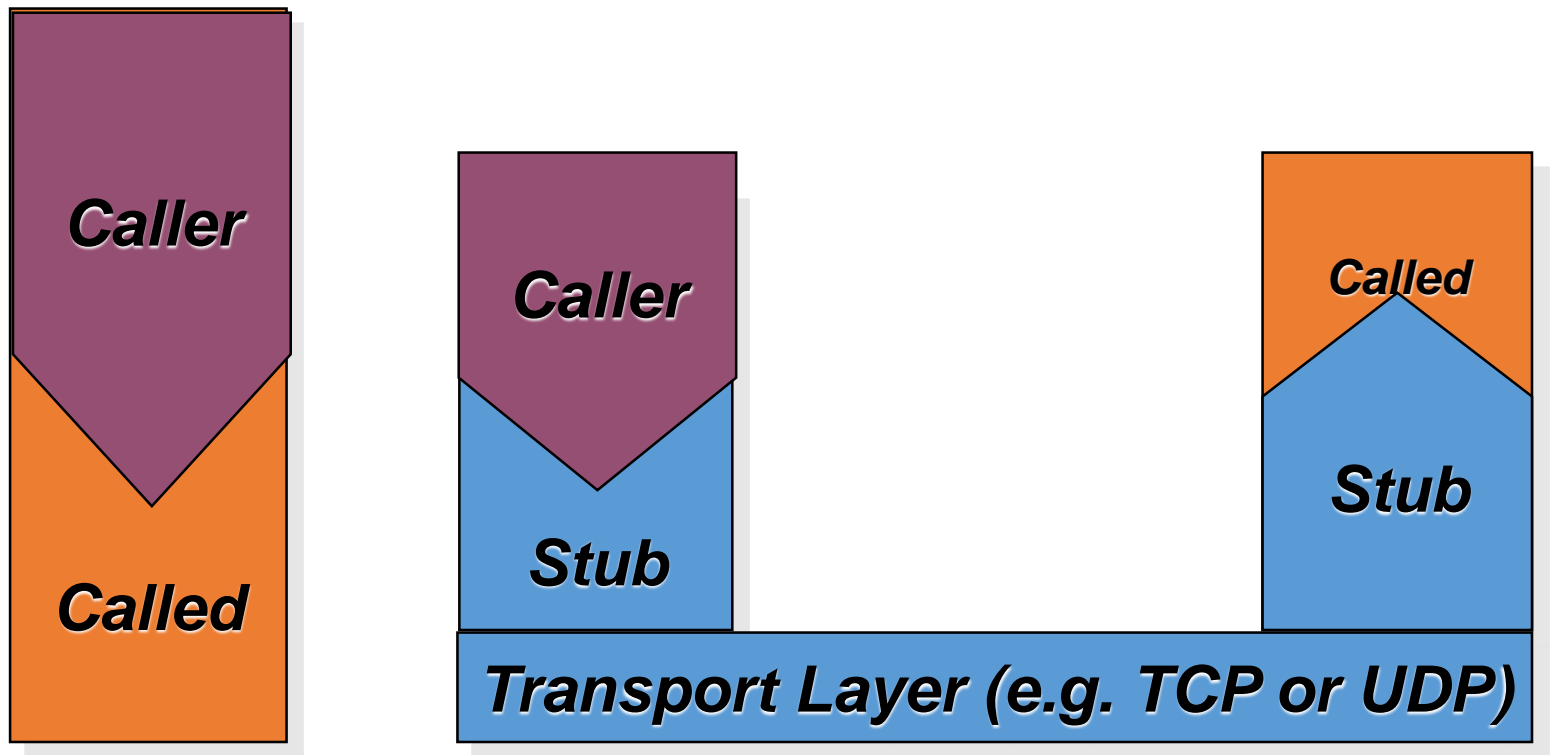
# Proxy pattern structure



# Example: Accessing Remote Object



# Java RMI, the big picture





# Categories of Proxies

- **Remote proxy** - as above
  - Local representative for something in a different address space
  - Java RMI tools help set these up automatically
  - Object brokers handle remote objects (CORBA or DCOM)
- **Virtual proxy**
  - Stand-in for an object that is expensive to implement or completely access
  - Example – image over the net
  - May be able to access some state (e.g., geometry) at low cost
  - Defer other high costs until it must be incurred
- **Protection proxy**
  - Control access to the "real" object
  - Different proxies provide different rights to different clients
  - For simple tasks, can do via multiple interfaces available to clients
  - For more dynamic checking, need a front-end such as a proxy

# Visitor Pattern

# The Problem

- Situation:
  - ❑ Operations to be performed on the elements of an object structure
- Desire:
  - ❑ Add new operations without changing the classes
  - ❑ Perform operation in a type-safe manner

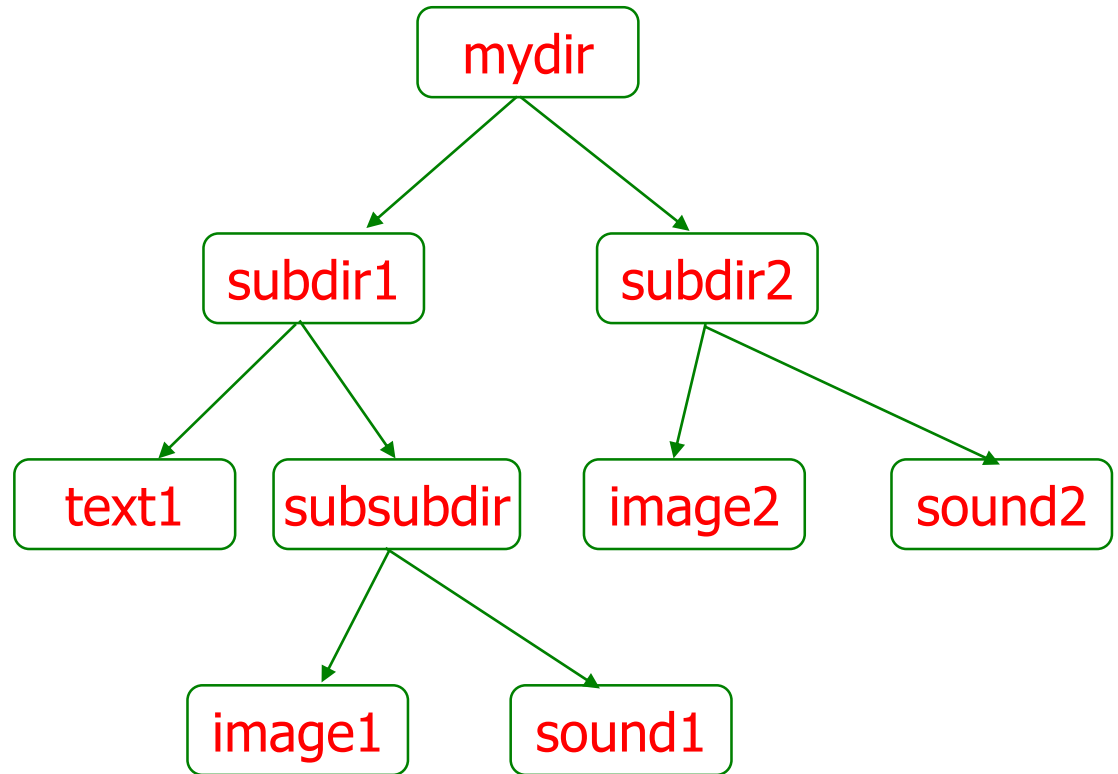
# Example – File System

- Basic Types

- Directories
- Text Files
- Image Files
- Sound Files

- Operations

- Total size
- SearchImage



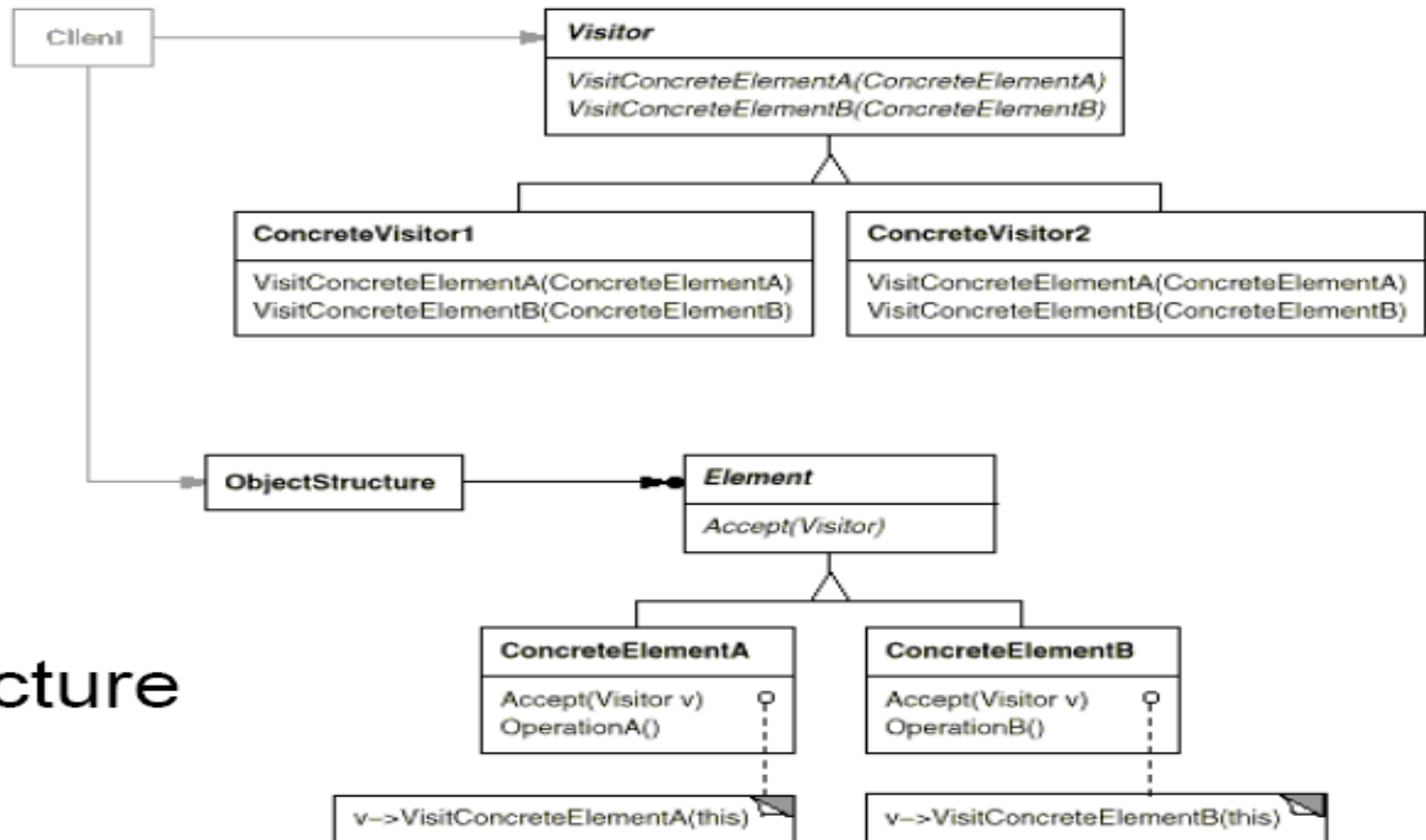
# Approach #1 – Operations in Interface

- Define all operations at topmost interface
- Override as necessary in each subclass
- Issue: adding new operations
  - Requires change to top most class
  - Requires *at least* recompilation of all subclasses
  - Typically adds code to each subclass
  - “Method bloat”

# Approach #2 - Visitor

- Define Visitor interface with one method per subclass type (e.g., visitDirectory, visitImage)
- Define **Accept** method in topmost Element interface
- **Accept** method provides a Visitor object
- Each class's **Accept** method calls that class's specific method in the Visitor object.
- Inside each visitor's method(s) we have access to all functionality of the object being visited

# Structure



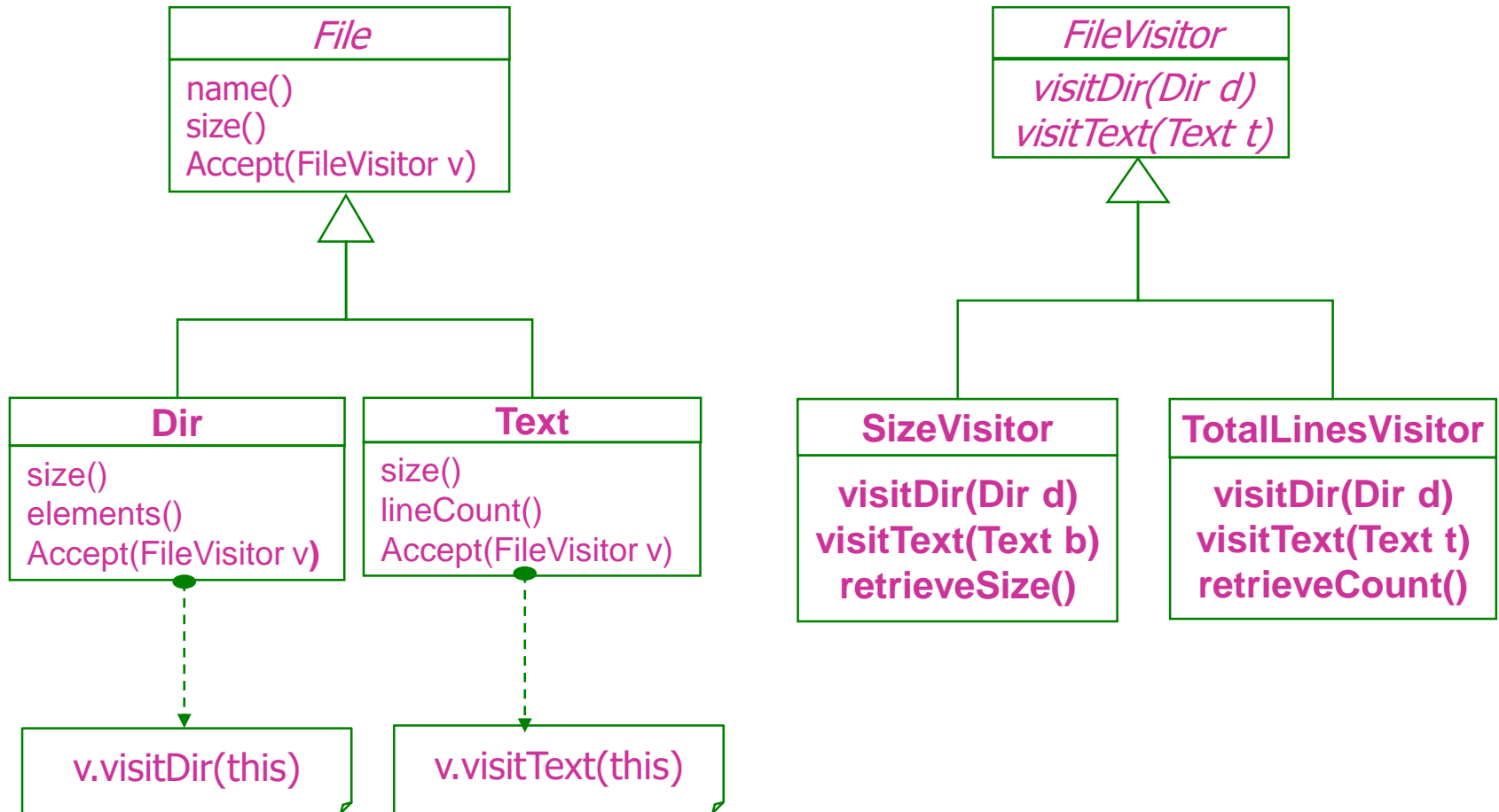
Structure

# Participants

- **Visitor** - This is an interface to declare the visit operations for all the types of visitable classes.
- **ConcreteVisitor** - Implement different operations for each visitor
- **Visitable** - is an abstraction which declares the accept operation. This is the entry point which enables an object to be "visited" by the visitor object.
- **ConcreteVisitable** - Those classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.
- **ObjectStructure** - This is a class containing all the objects that can be visited.



# Class Diagram



# Applicability

- Many distinct, unrelated operations are to be performed
- You want to minimize the common interface
  - ❑ Isolates conceptual operation in its own class
  - ❑ Supports different operations in different applications that share the overall object structure.
- Result
  - ❑ The object structure rarely change
  - ❑ The operations over the structure change frequently

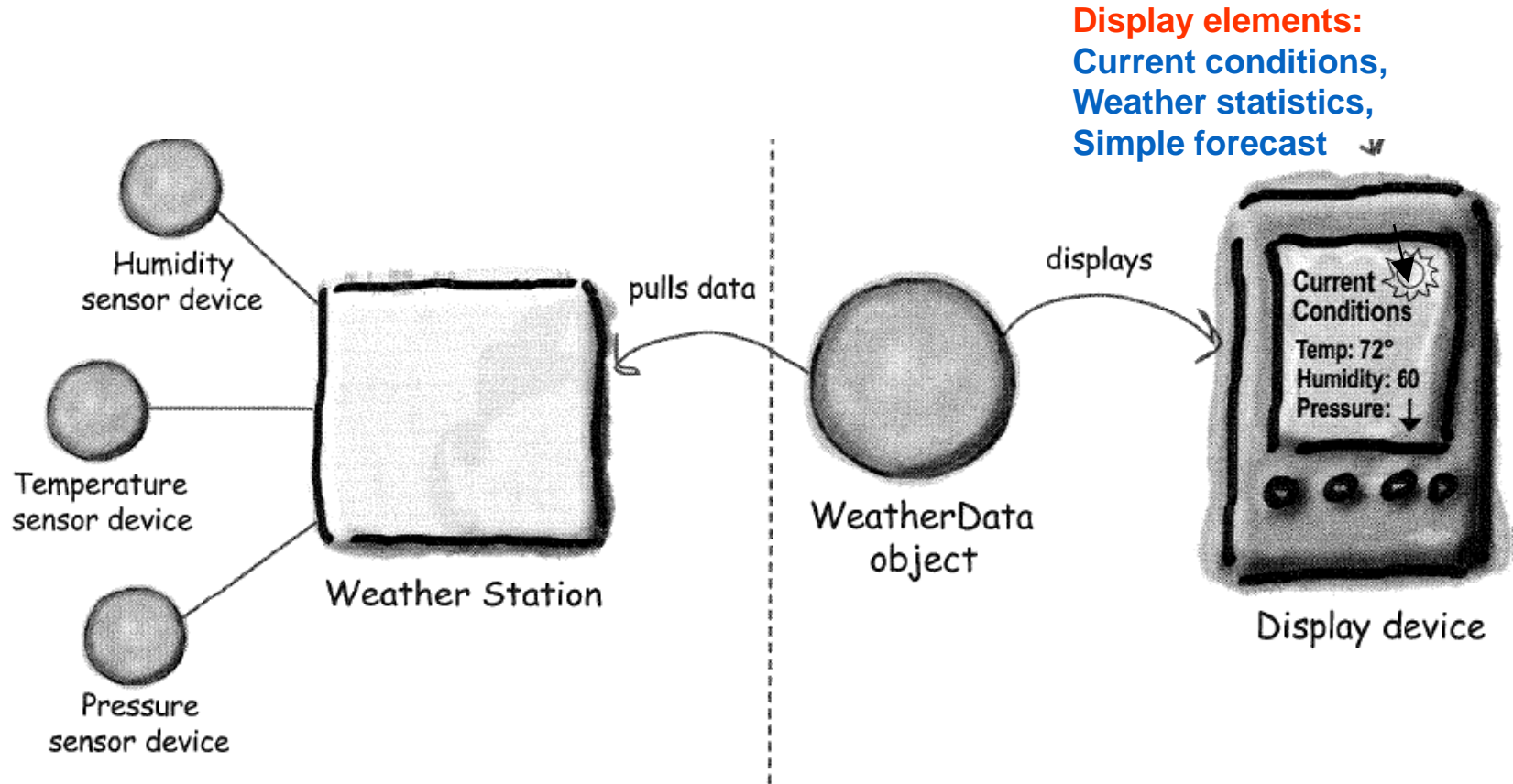
# Drawbacks

- ❑ Adding new concrete element classes is difficult:
  - ❑ All visitors must change
  - ❑ Here putting functionality in structure probably better
- ❑ Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class.

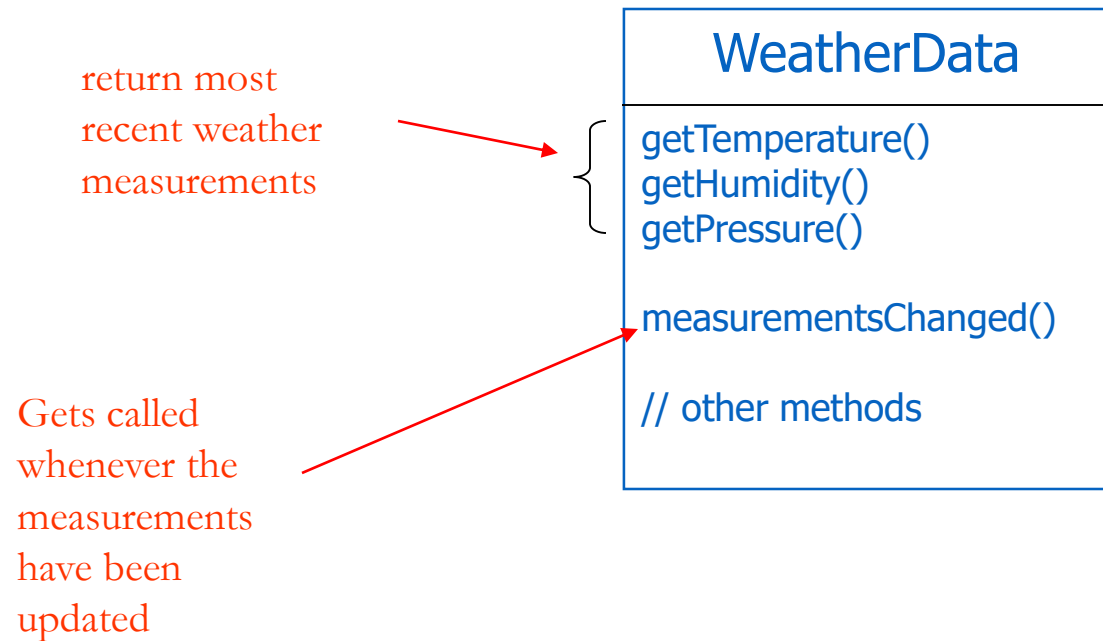
# Observer pattern

*Acknowledgement: Freeman & Freeman*

# Weather Monitoring application



# WeatherData class



# WeatherData Implementation

```
public class WeatherData {  
    // instance variable declarations  
  
    public void measurementsChanged() {  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        currentConditionsDisplay.update (temp, humidity, pressure);  
        statisticsDisplay.update (temp, humidity, pressure);  
        forecastDisplay.update (temp, humidity, pressure);  
    }  
  
    // other WeatherData methods here  
}
```

# The Problem

- Given
  - Clusters of related classes
  - Tight connections within each cluster of classes
  - Loose state dependency between clusters
- Desired
  - Keep each cluster state consistent when state changes in cluster it depends on
  - Provide isolation such that changed cluster has no knowledge of specifics of dependent clusters



# Example: UI & Application

- Application classes represent information being manipulated.
- UI provides way to view and alter application state.
- May have several views of state (charts, graphs, numeric tables).
- Views may be added at any time.
- How to tell views when application state has changed?

# Approach One – Direct Connect

- Application knows about each View object.
- On state change, call appropriate method in the View object affected
- Issues
  - Application needs to know which method to call in each View
  - Application aware of changes to UI (e.g., add/delete/change Views).

## Approach Two : Observer Pattern

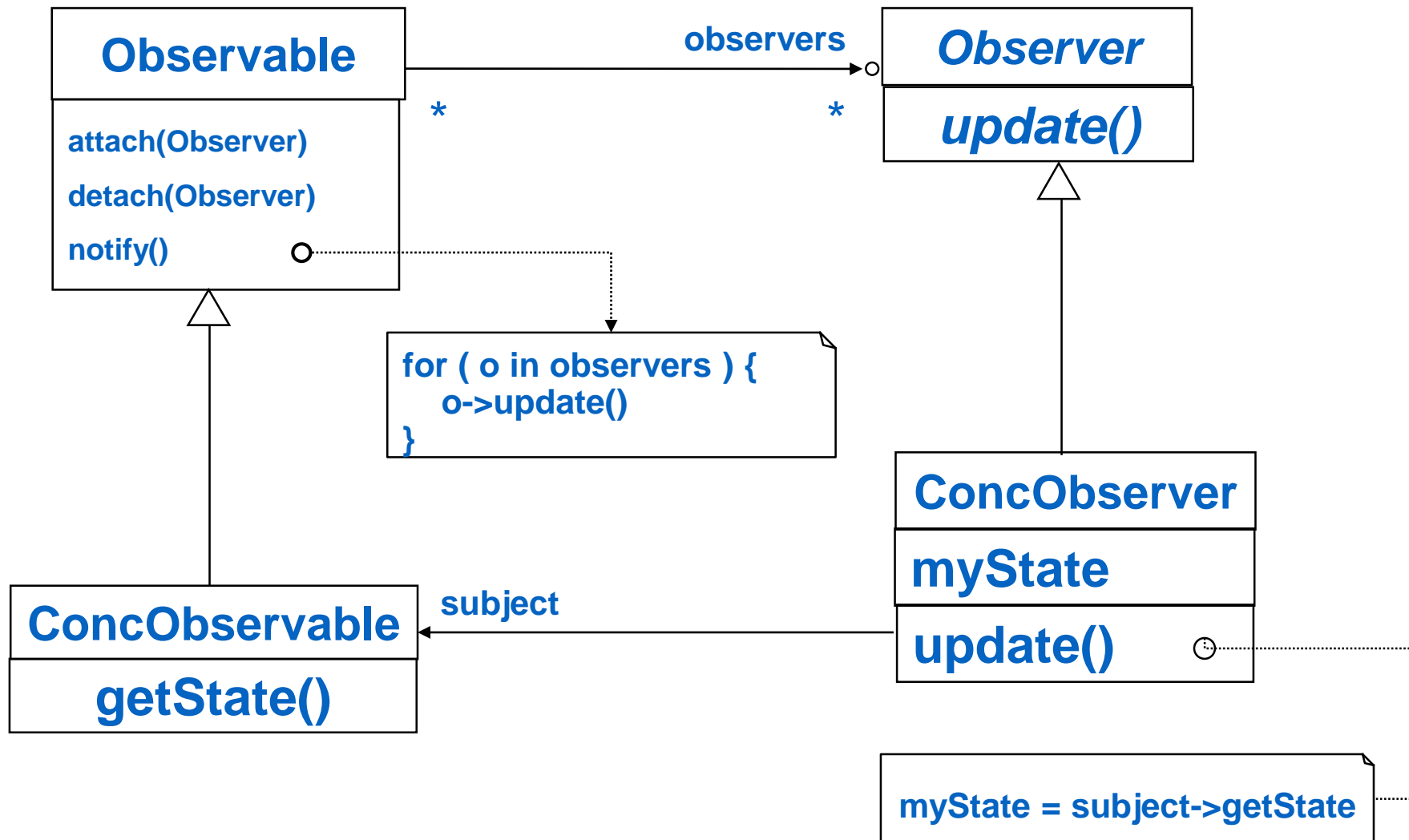
- The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically

# Approach Two – Observer/Subject

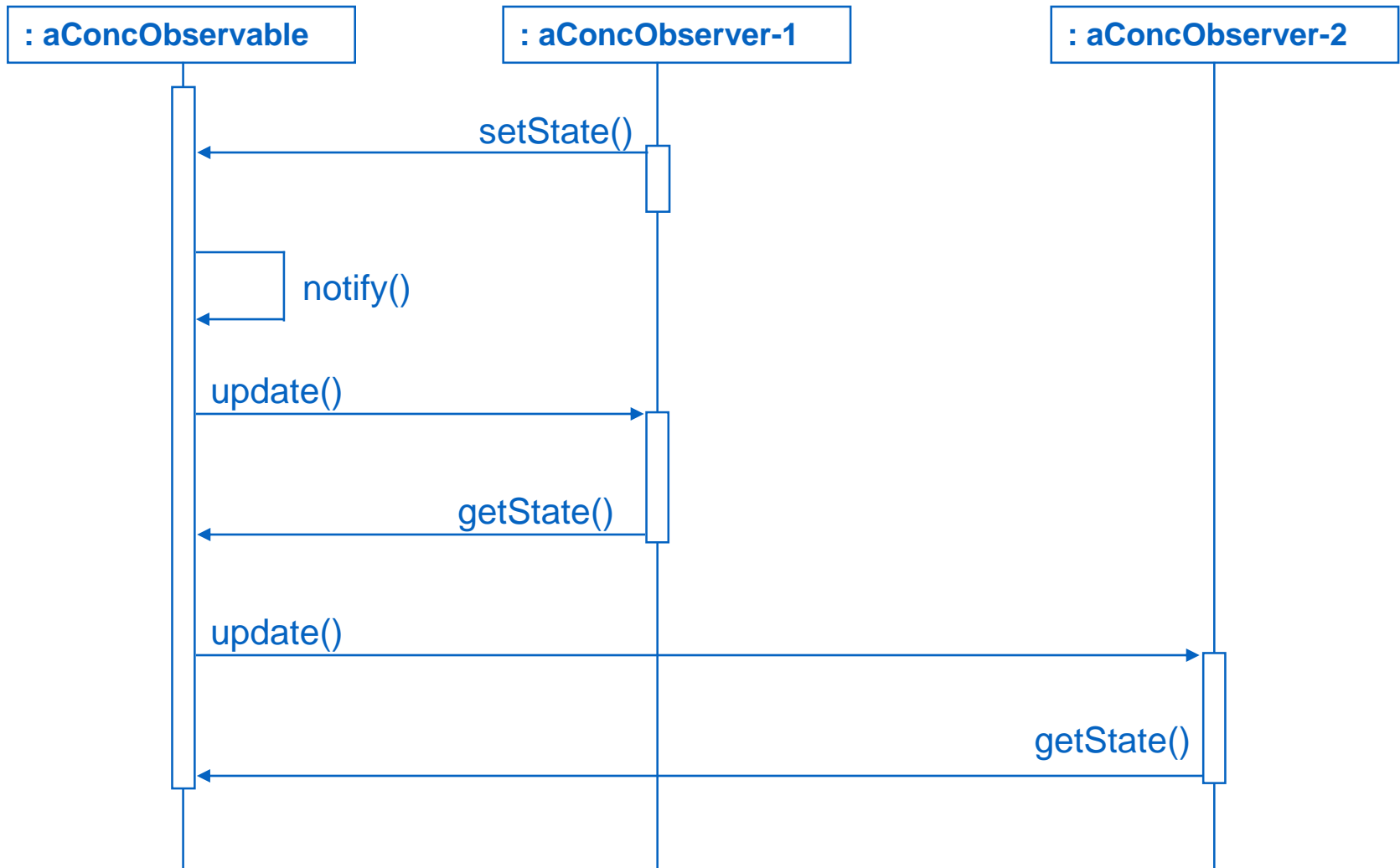
(AKA: publish/subscribe)

- **Observables**: objects with interesting state changes
- **Observers**: objects interested in state changes
- Observers *register* interest with observable objects.
- Observables *notify* all registered observers when state changes.

# Observer Pattern - Class diagram



# Interaction Diagram



# When to Use Observer

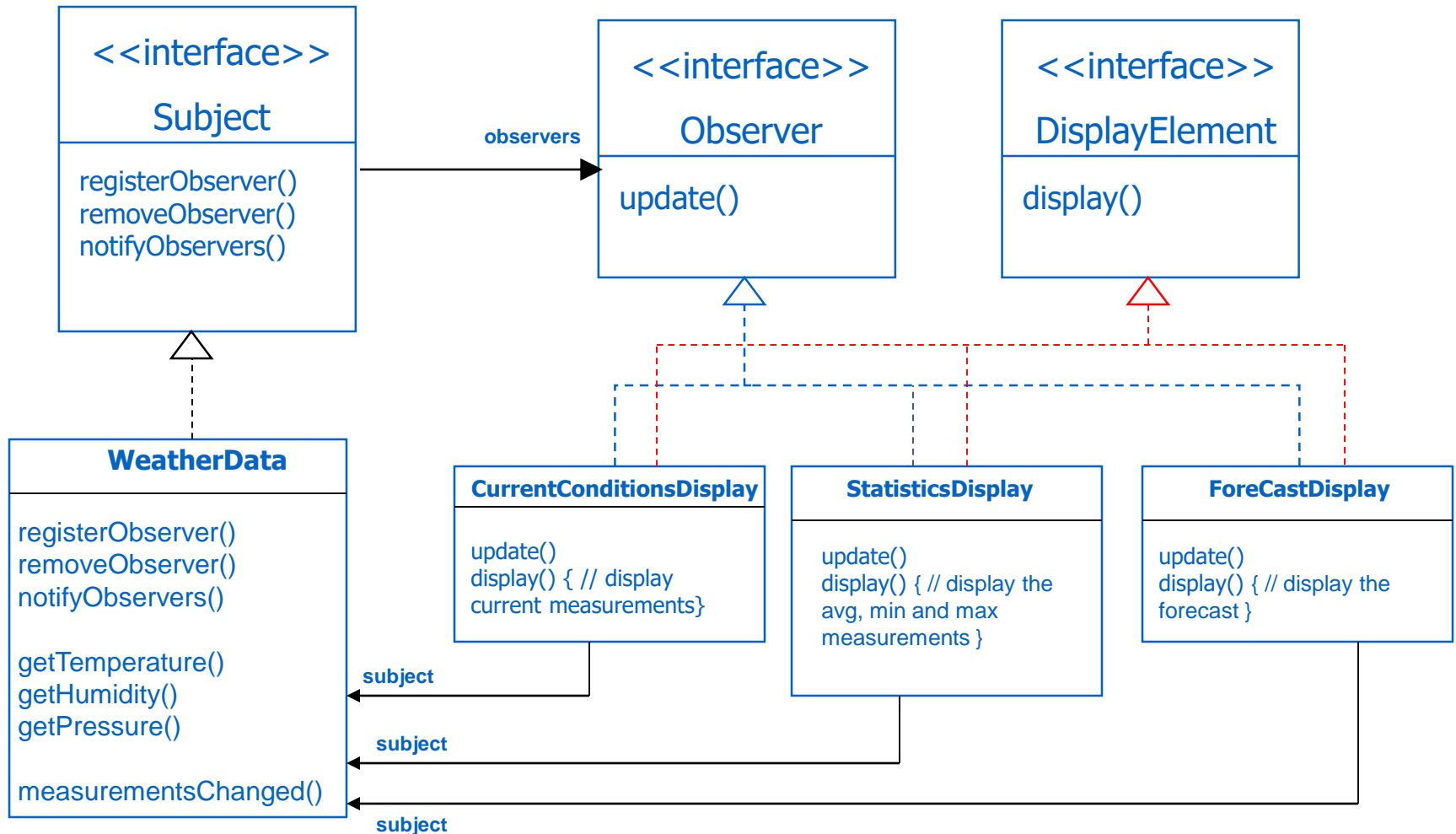
- Two subsystems evolve independently but must stay in synch.
- State change in an object requires changes in an unknown number of other objects (broadcast)
- Desire loose coupling between changeable object and those interested in the change.

# Consequences

- Subject/observer coupling (**loose coupling**)
  - Subject only knows it has a list of observers
  - The only thing the Subject knows about an Observer is that it implements a certain interface
  - Observers can be added at any time
  - Does not know any Observer concrete class
  - Subjects don't need to be modified to add new types of Observers
  - Subjects and Observers can be reused independently
- Supports broadcast communication
  - Observables know little about notify receivers
  - Changing observers is trivial
- Unexpected & cascading updates
  - change/notify/update -> change/notify/update
  - May be hard to tell *what* changed



# Designing the Weather Station



# Observer Pattern – Key points

- Observer pattern defines a one-to-many relationship between objects
- Subjects/Observables update observers using a common interface
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement observer interface
- You can **PUSH** or **PULL** data from the Observable when using the pattern (pull is considered more “correct”)
- Don’t depend on specific order of notification for your observers

# Composite pattern

*Acknowledgement: Freeman & Freeman*

# The Problem

- Problem
  - Have simple primitive component classes that collect into larger *composite* components
- Desire
  - Treat composites like primitives
  - Support composite sub-assemblies
  - Operations (usually) recurse to subassemblies
- Solution
  - Build composites from primitive elements

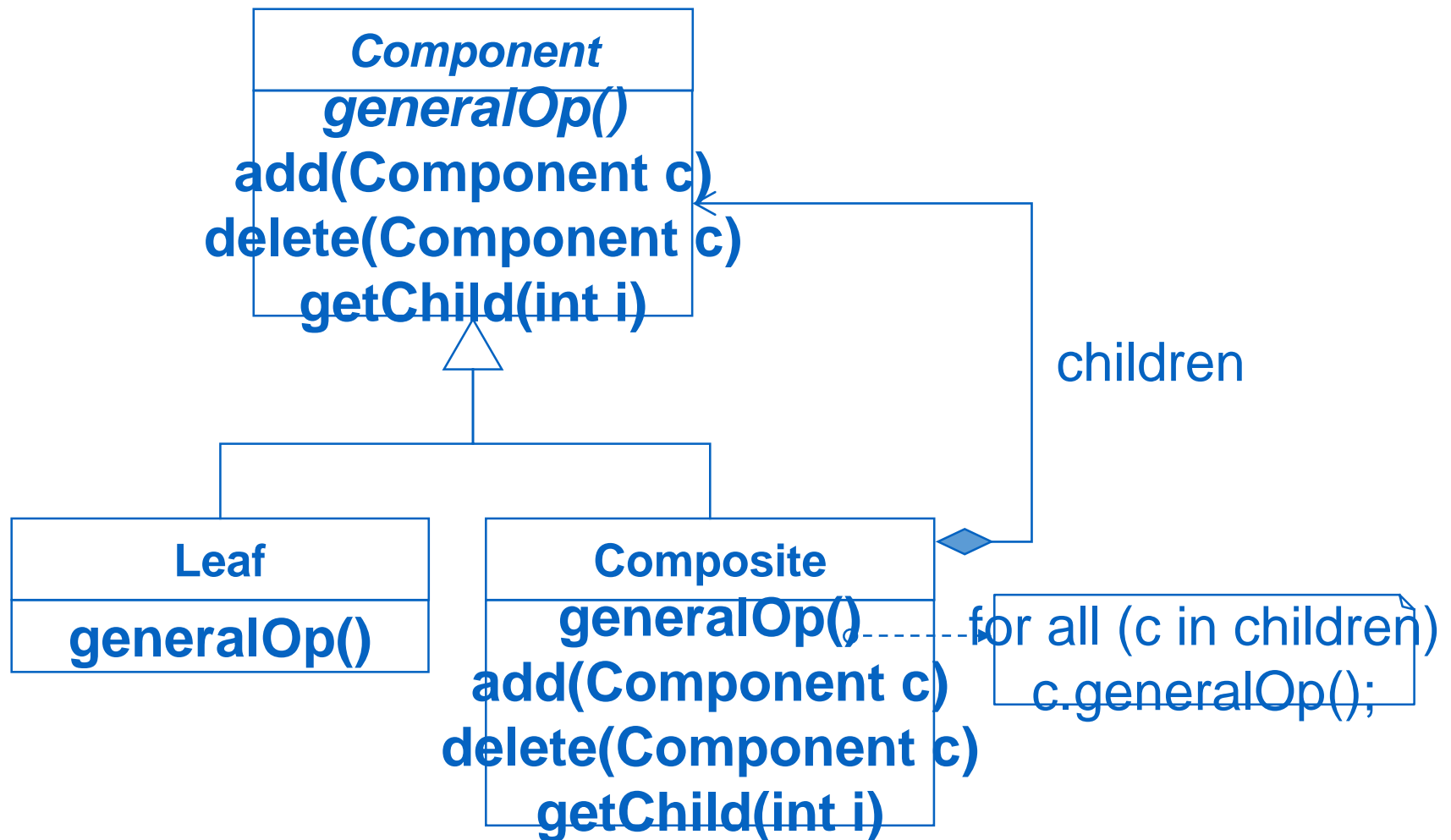
# Examples - I

- File systems
  - Primitives = text files, binary files, device files, etc.
  - Composites = directories (w/subdirectories)
- Make file dependencies
  - Primitives = leaf targets with no dependents
  - Composites = targets with dependents
- Menus
  - Primitives = menu entries
  - Composites = menus (w/submenus)

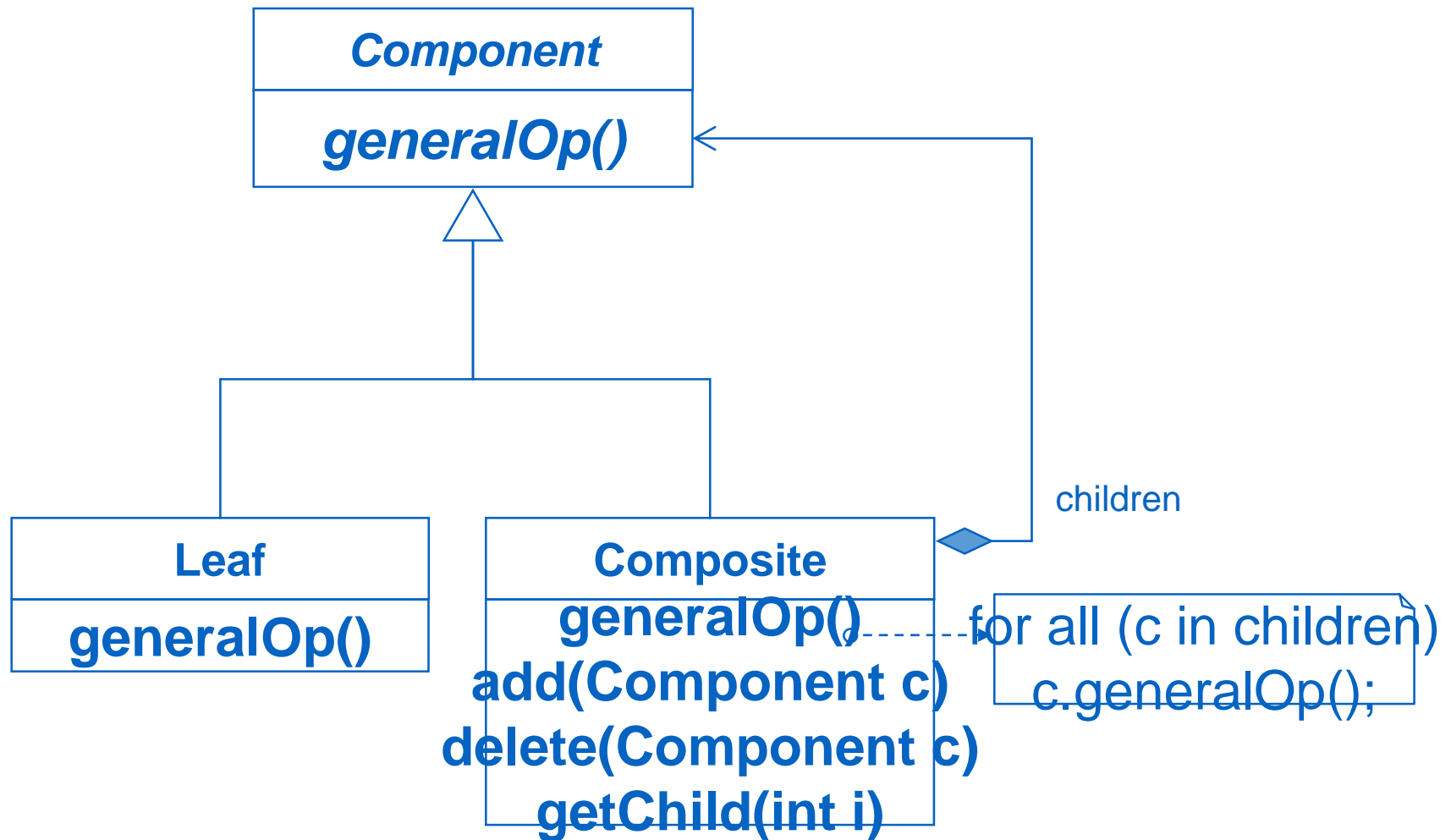
# Examples - 2

- GUI Toolkits
  - Primitives = basic components (buttons, textareas, listboxes, etc).
  - Composites = frames, dialogs, panels.
- Drawing Applications
  - Primitives = lines, strings, polygons, etc.
  - Composites = groupings treated as unit.
- HTML/XML
  - Pages as composites of links (hypertext)
  - Pages as collections of paragraphs (with subparagraphs for lists, etc.)

# Class Diagram (Alternative 1)

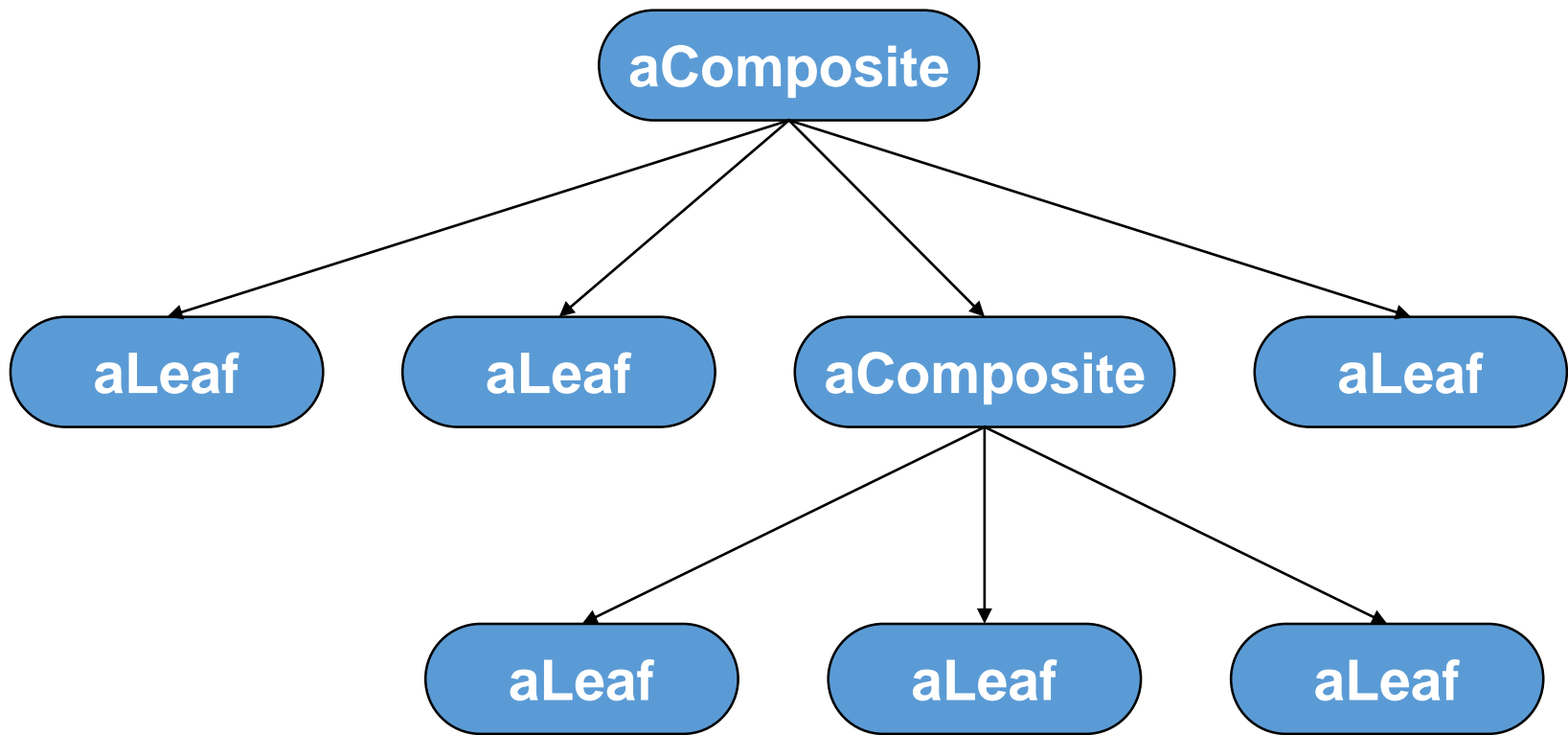


# Class Diagram (Alternative 2)

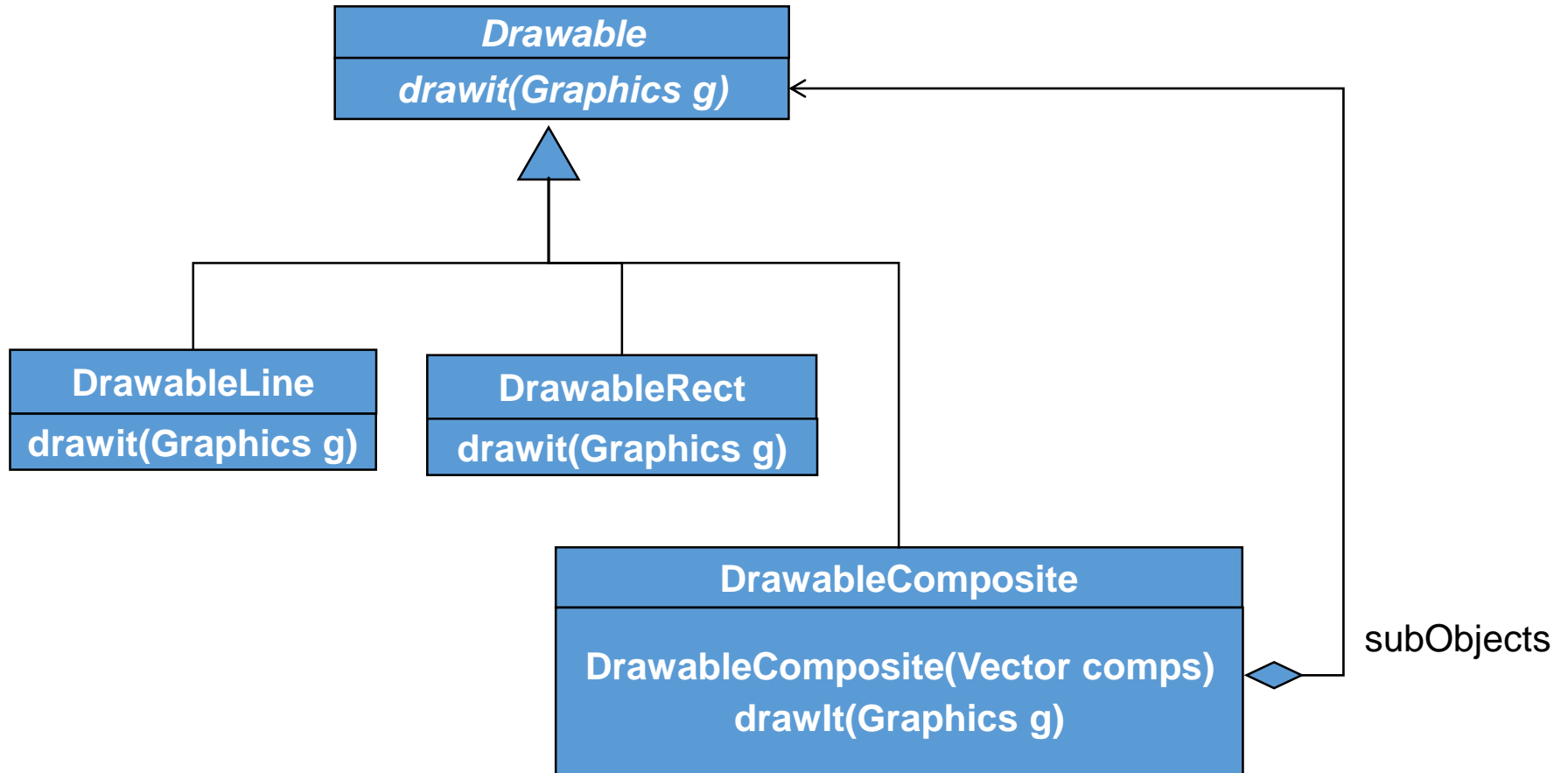




# Example Object Structure



# Example: Drawable Figures



# Discussion

- Clients (usually) ignore differences between primitives & composites
- Clients access (most) components via the generic interfaces
  - Primitives implement request directly
  - Composites can handle directly or forward
- Arbitrary composition to indefinite depth
  - Tree structure – no sharing of nodes
  - General digraph – supports sharing, multiple parents – be careful!
- Eases addition of new components
  - Almost automatic

# Evaluating Designs

- The application of “well-known” design patterns that promote loosely coupled, highly cohesive designs.
- Conversely, identify the existence of recurring *negative* solutions – AntiPatterns
- AntiPattern : use of a pattern in an inappropriate context.
- Refactoring : changing, migrating an existing solution (antipattern) to another by improving the structure of the solution.