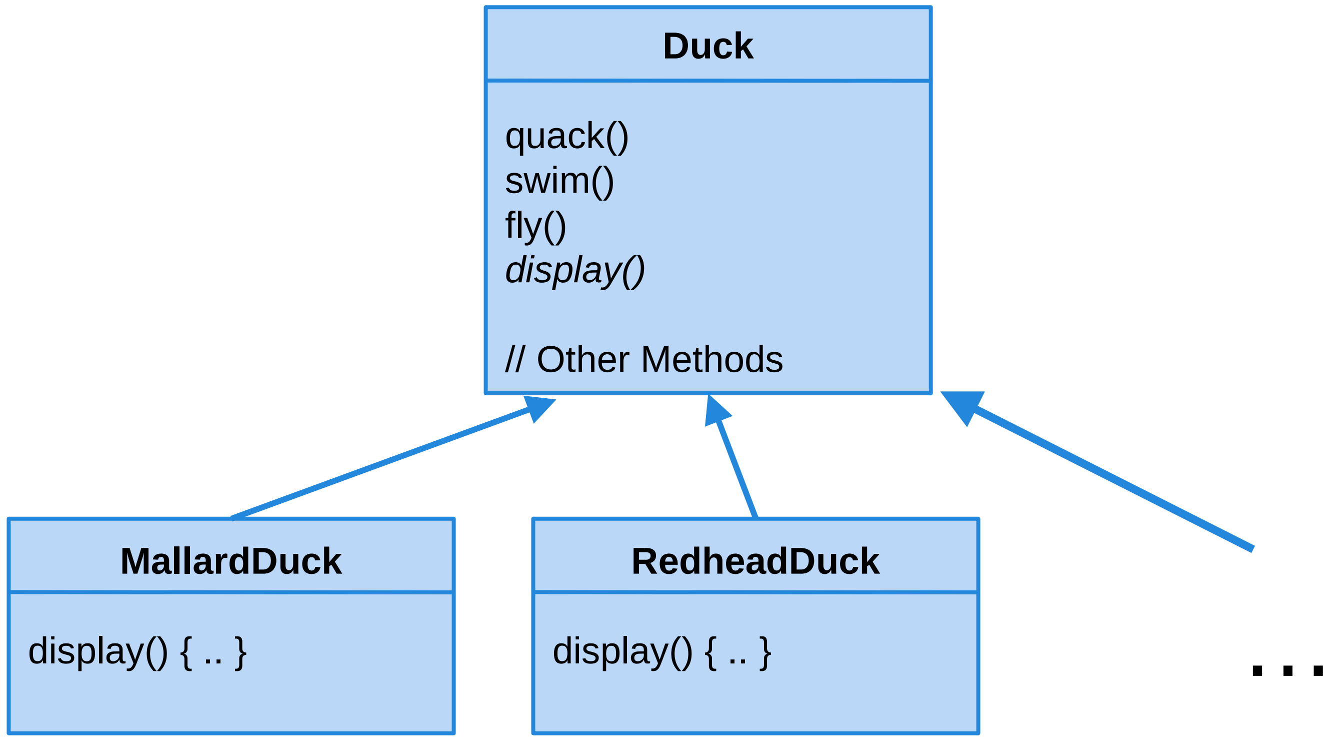
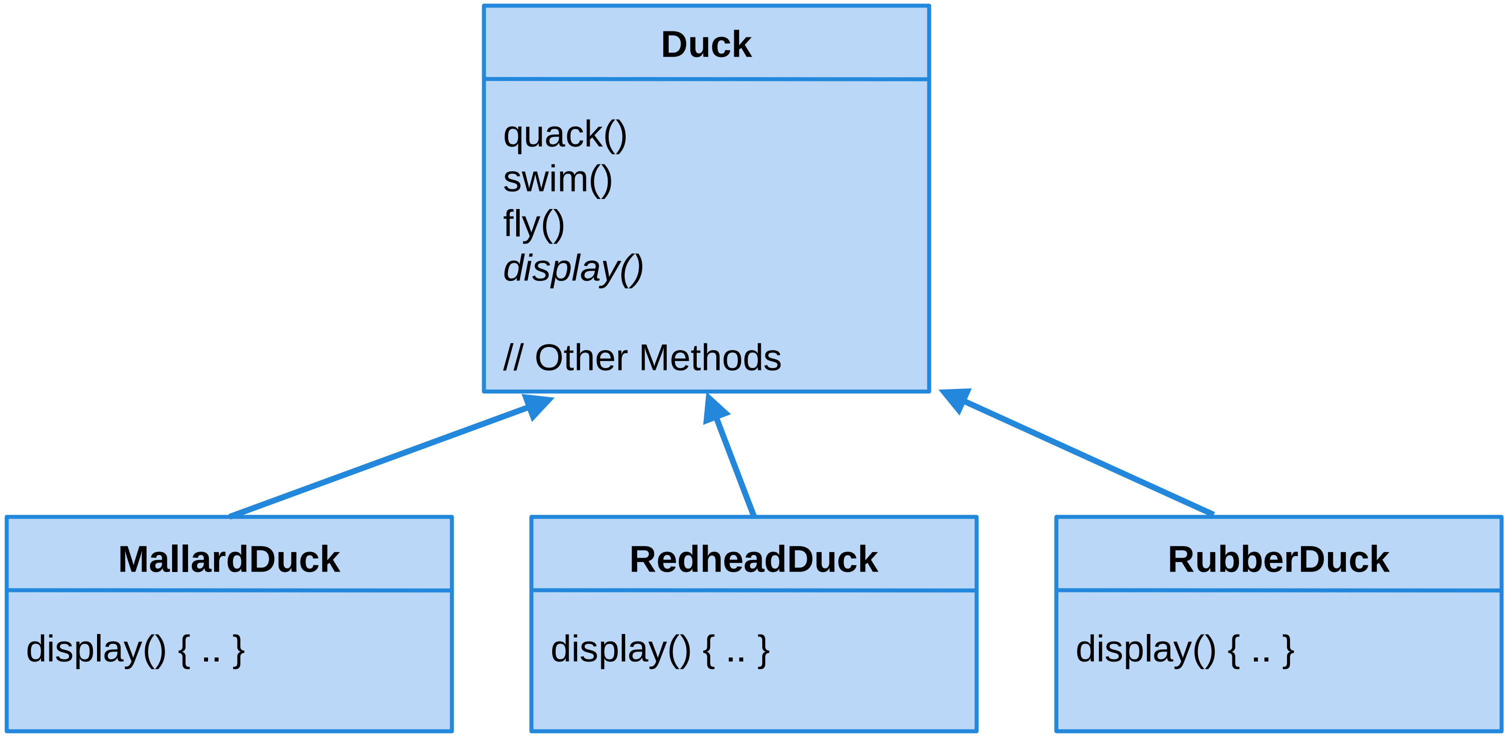
Design Patterns

Slides courtesy Gregory Gay

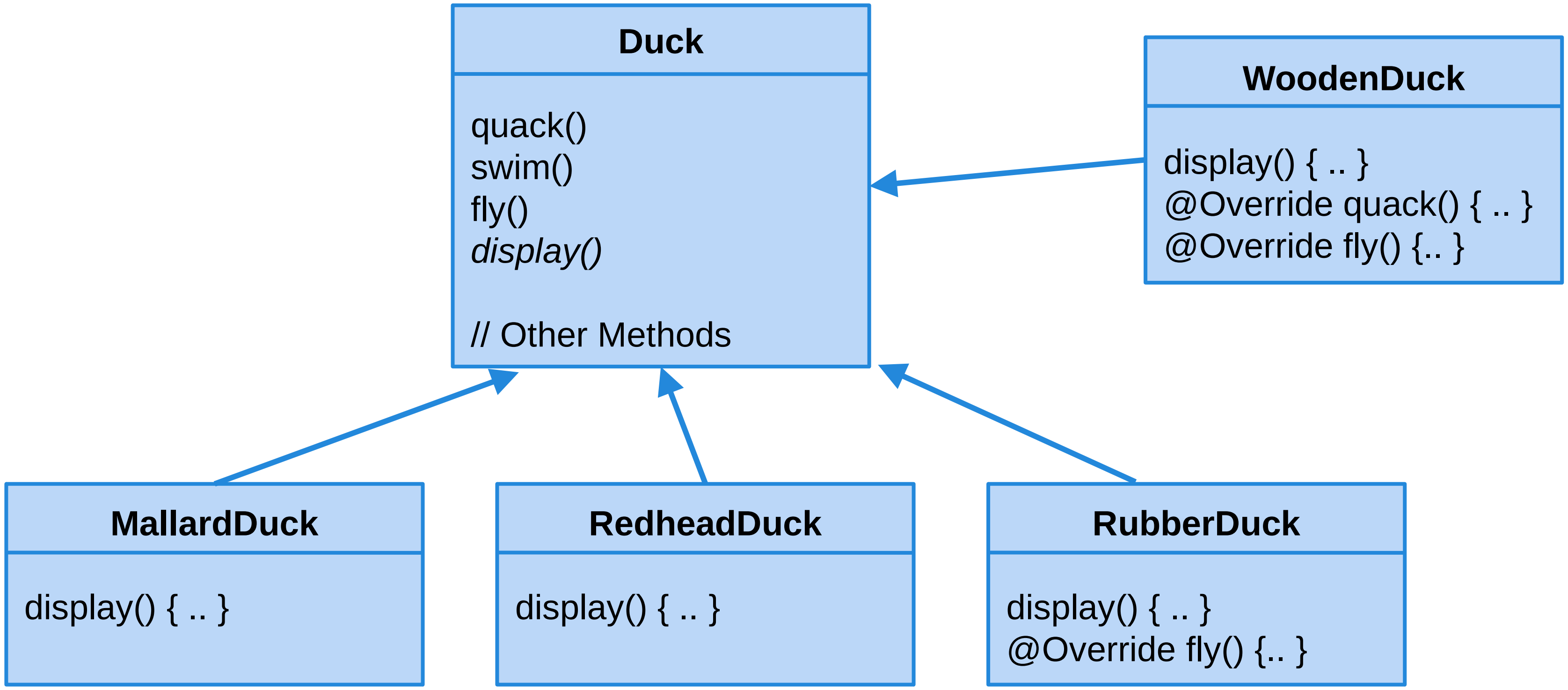
OO Design Exercise:
Building a Better Duck



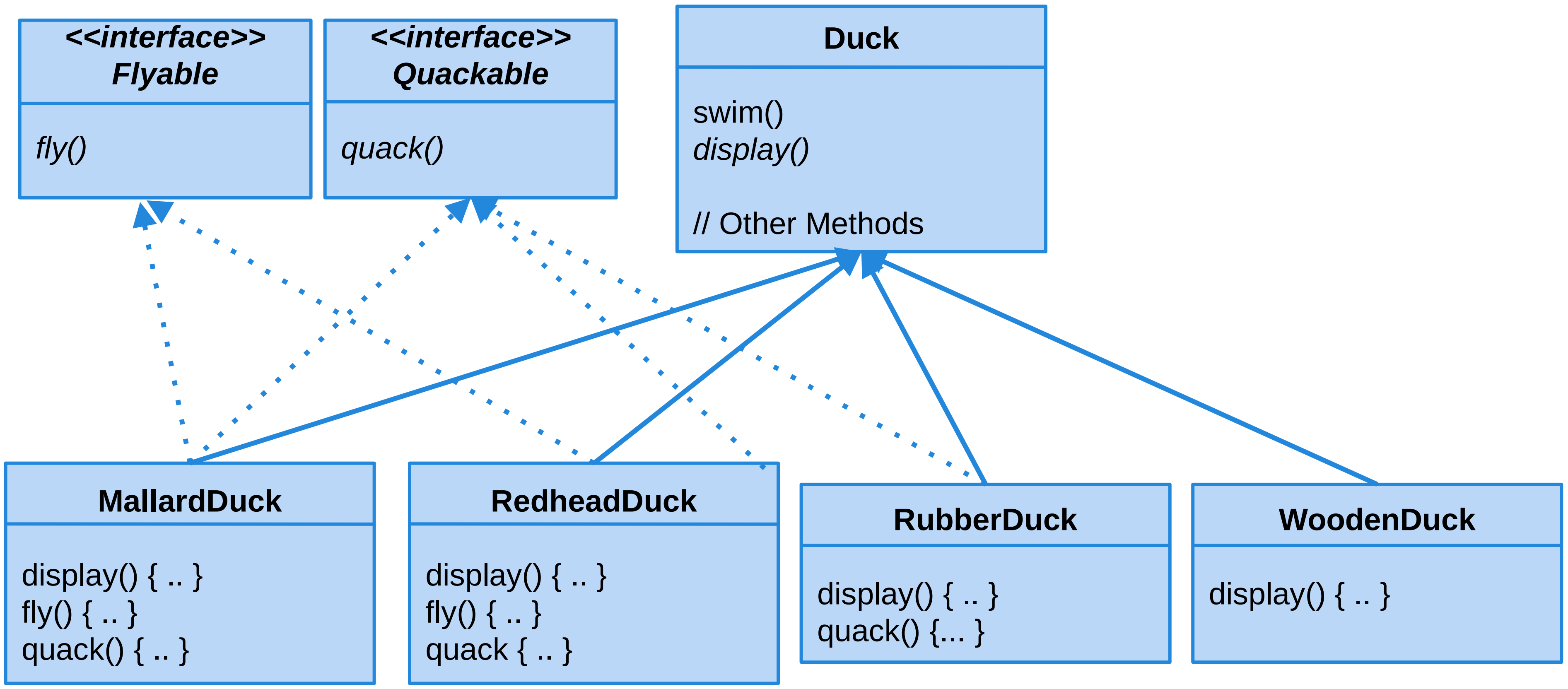
Adding new ducks



Why not override?



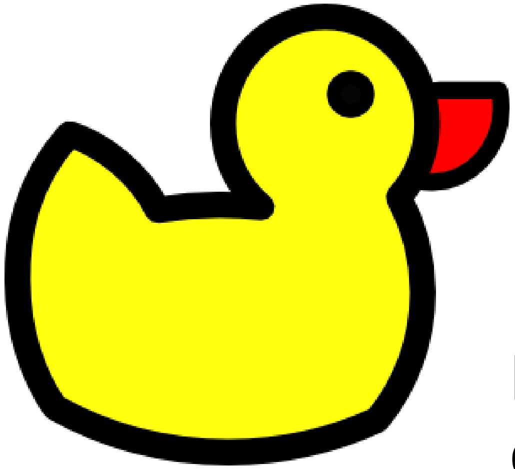
Why not interfaces?



How do we fix this mess?

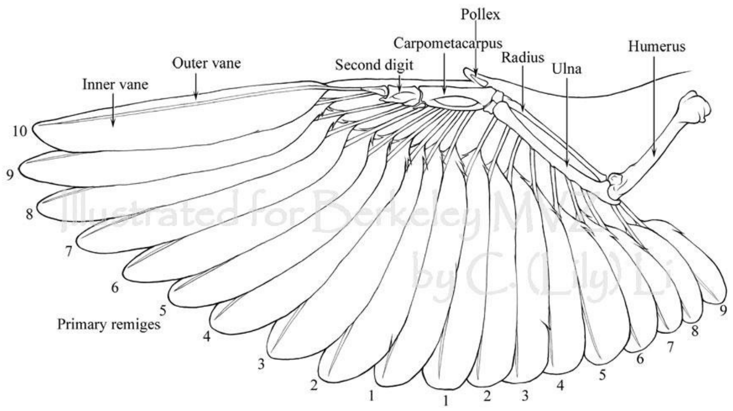
Apply good OO design principles!

Step 1: Identify the aspects that vary and encapsulate them.



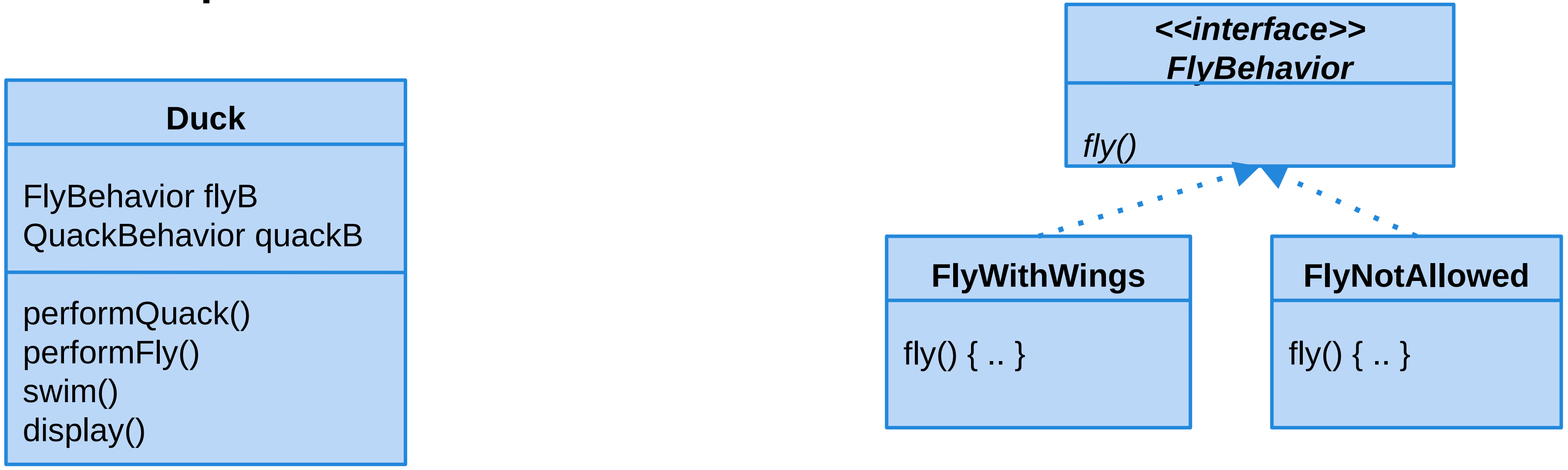
Duck class

Flying behaviors



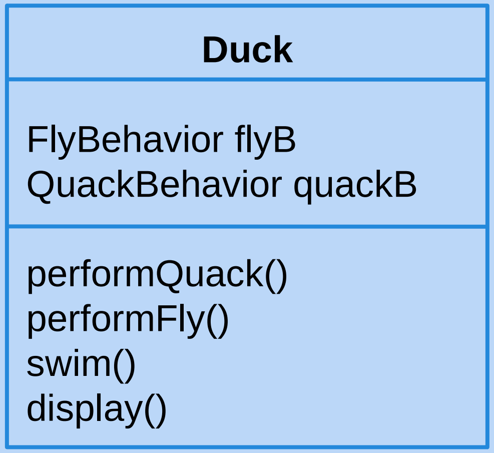
Step 2:
Implement behaviors as classes

Principle - Program to an interface, not an implementation.



Step 2: Implement behaviors as classes

Principle - Program to an interface, not an implementation.



```
performFly() {
    flyB.fly();
}
```

(BAD)
Programming to an implementation:

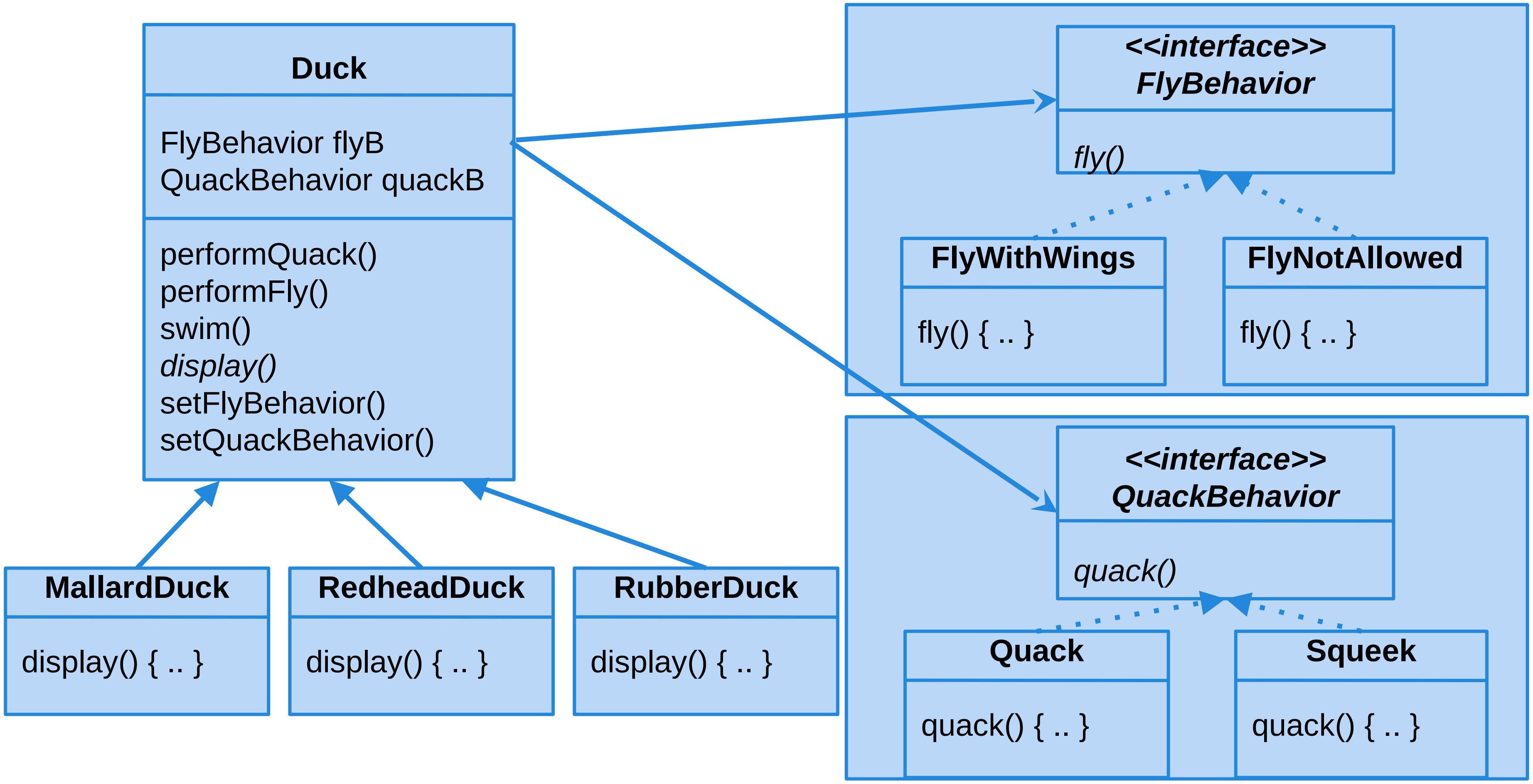
```
MallardDuck d = new MallardDuck();
d.flyWithWings();
```

(GOOD)
Programming to an interface:

```
Duck d = new MallardDuck()
d.performFly();
```

HAS-A can be better than IS-A

Principle: Favor composition over inheritance.



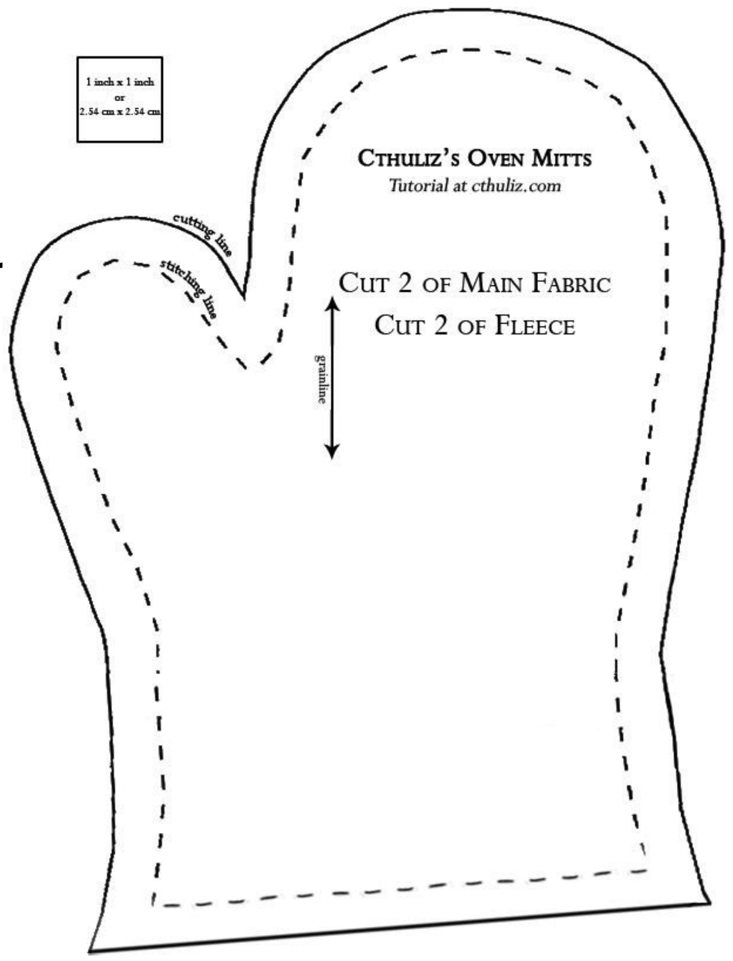
Challenge - Duck Call

A duck call is a device that hunters use to mimic the sound of a duck. How would you implement a duck call in this framework?

Enter... Design patterns

Don’t describe *classes*, describe ***problems***.

Patterns prescribe design guidelines for problem classes.



Guidelines, not solutions

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

- Christopher Alexander

Categories of design patterns

1. Creational

   Decouple a client from objects it instantiates.

2. Structural

   Clean organization into subsystems.

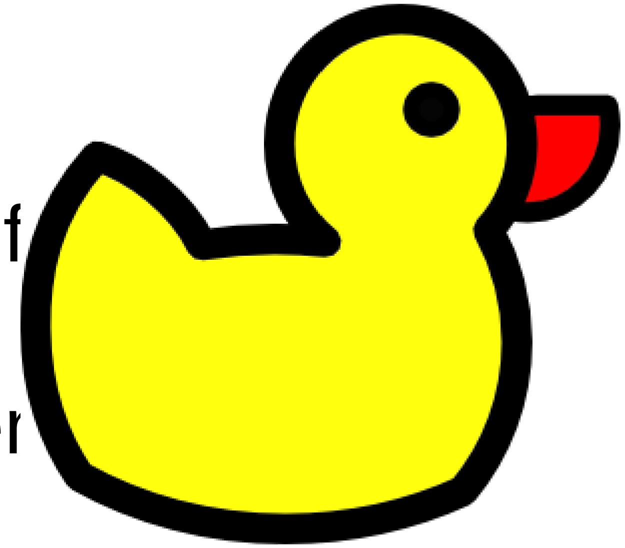
3. Behavioral

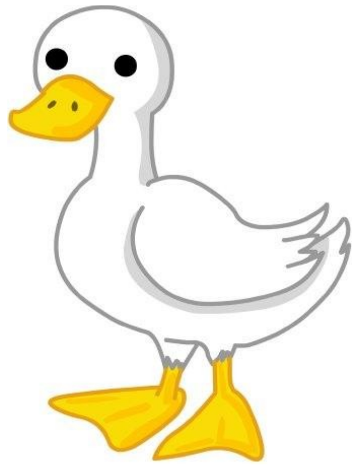
   Describe how objects interact.

You already applied one pattern

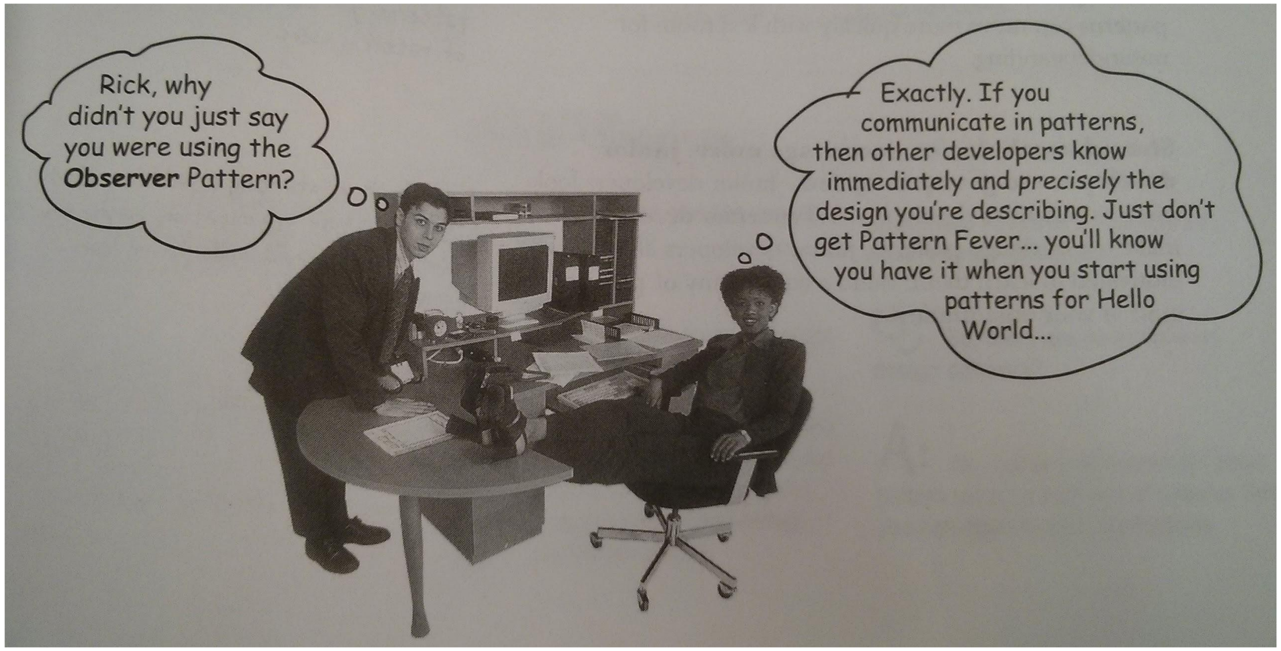
Strategy Pattern

Defines a family of algorithms, encapsulates ther makes them interchangeable.



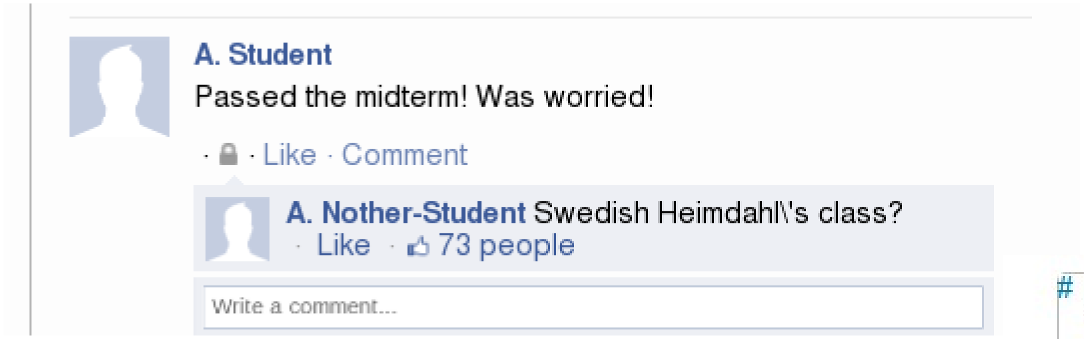


Why use design patterns?

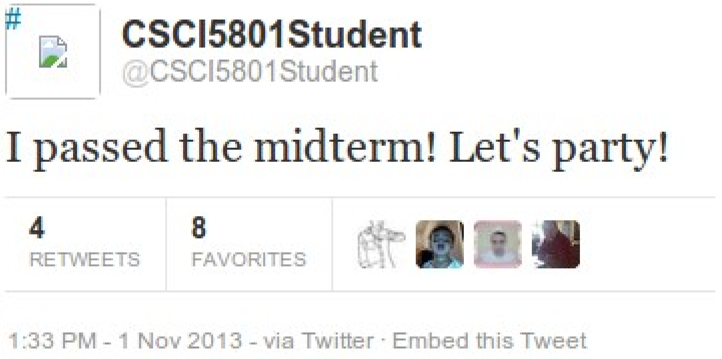

Rick, why didn't you just say you were using the **Observer** Pattern?
Exactly. If you communicate in patterns, then other developers know immediately and *precisely* the design you're describing. Just don't get Pattern Fever... you'll know you have it when you start using patterns for Hello World...

1. Good examples of OO principles.
2. Faster design phase.
3. Evidence that system will support change.
4. Offers shared vocabulary between designers.

Observer Pattern - Motivation

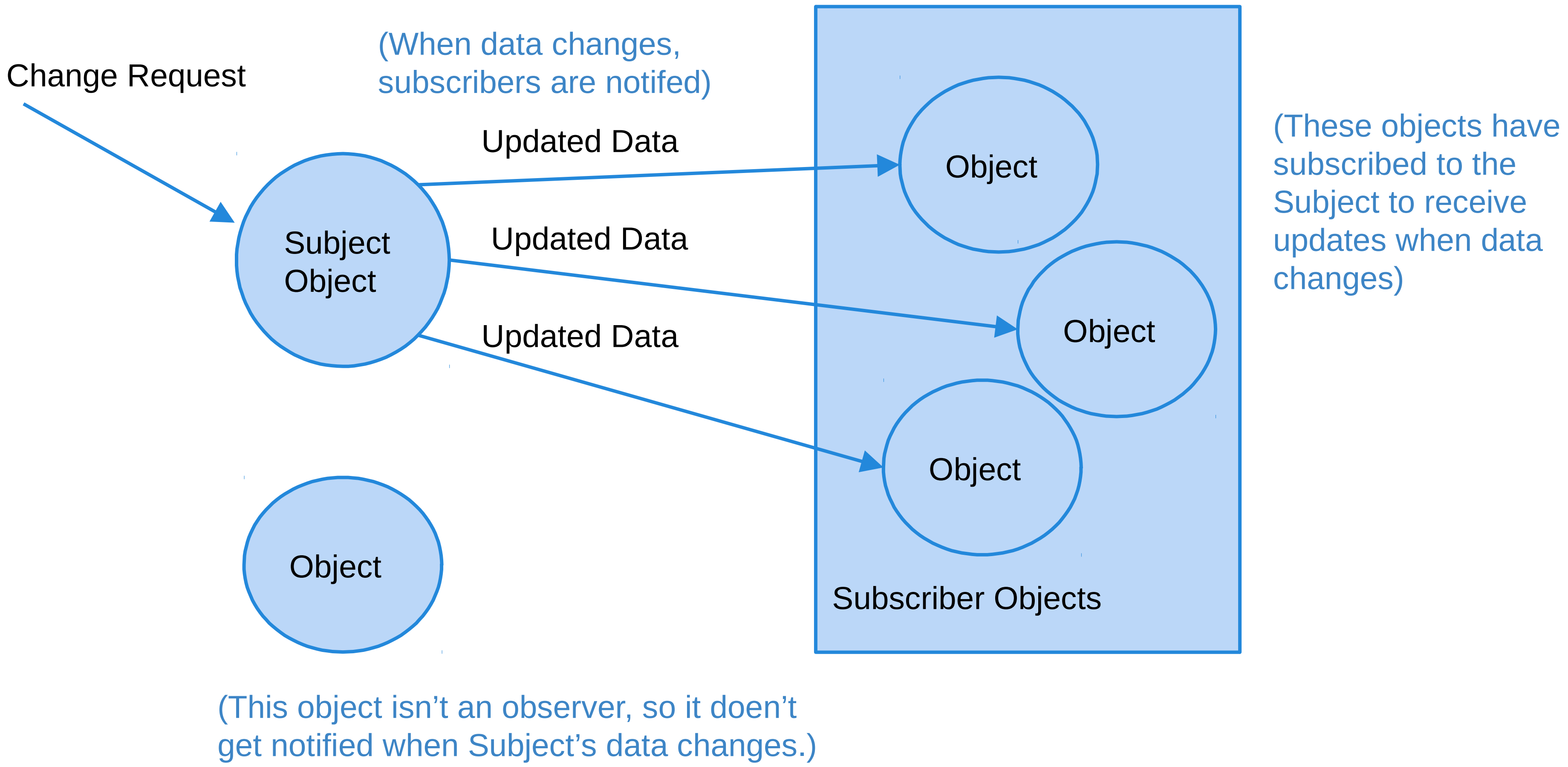


A. Student
Passed the midterm! Was worried!
· · Like · Comment
A. Nother-Student Swedish Heimdahl\'s class?
· Like · 73 people
Write a comment...

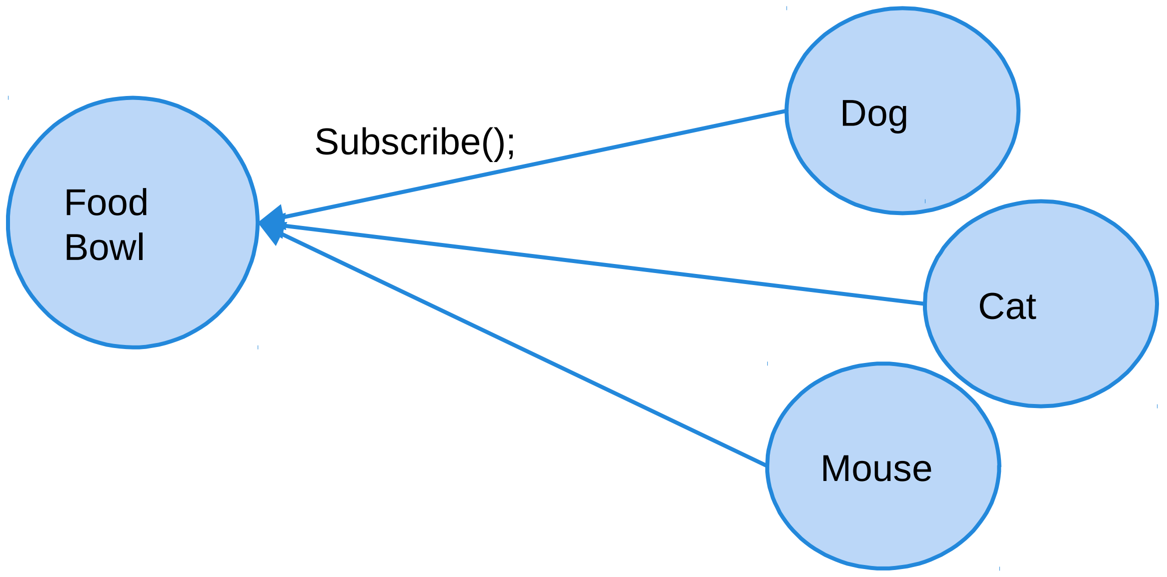


CSCI5801Student
@CSCI5801Student
I passed the midterm! Let's party!
4 RETWEETS 8 FAVORITES
1:33 PM - 1 Nov 2013 - via Twitter · Embed this Tweet

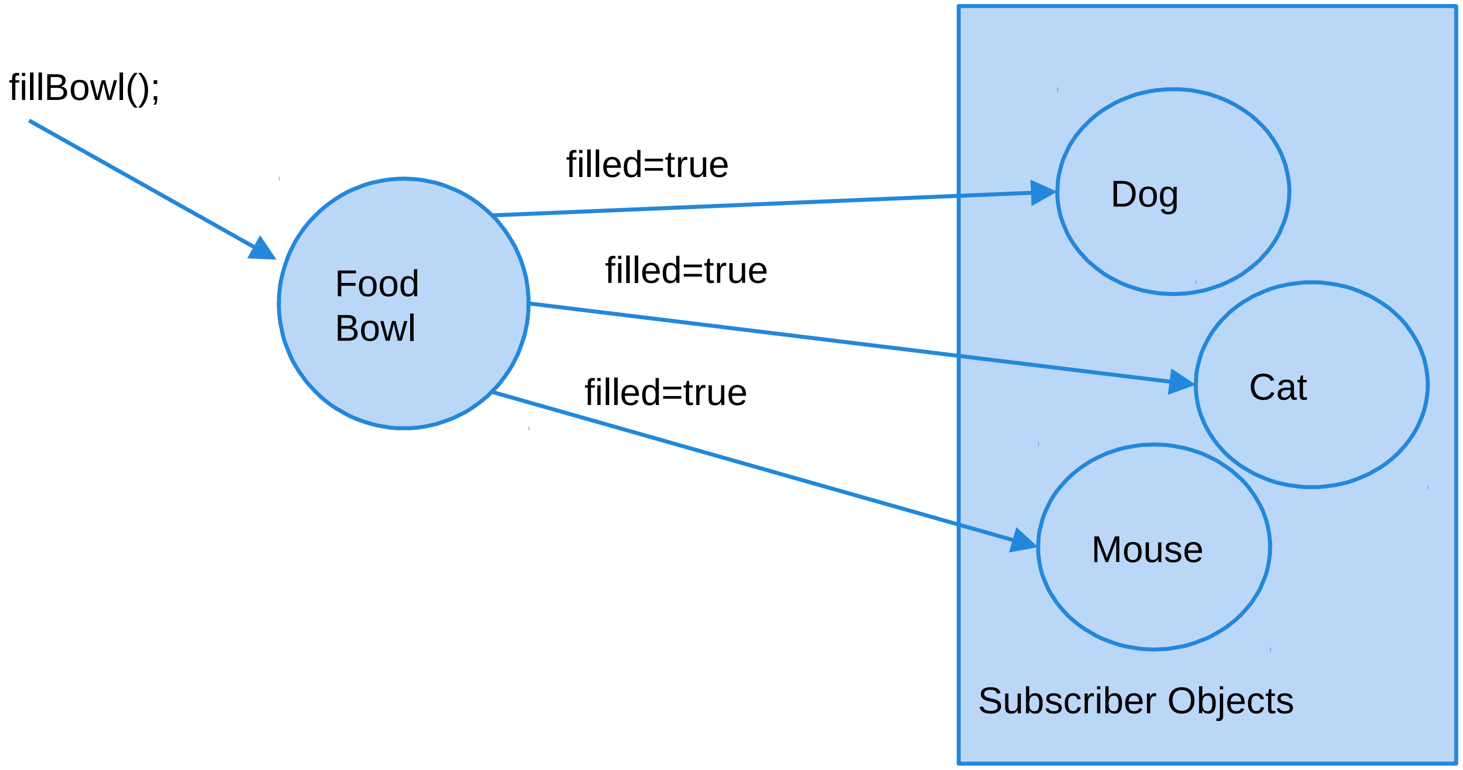
Observer Pattern - Definition


Change Request
(When data changes, subscribers are notifed)
Updated Data
Updated Data
Updated Data
Subject Object
Object
Object
Object
Object
Subscriber Objects
(These objects have subscribed to the Subject to receive updates when data changes)
(This object isn't an observer, so it doen't get notified when Subject's data changes.)

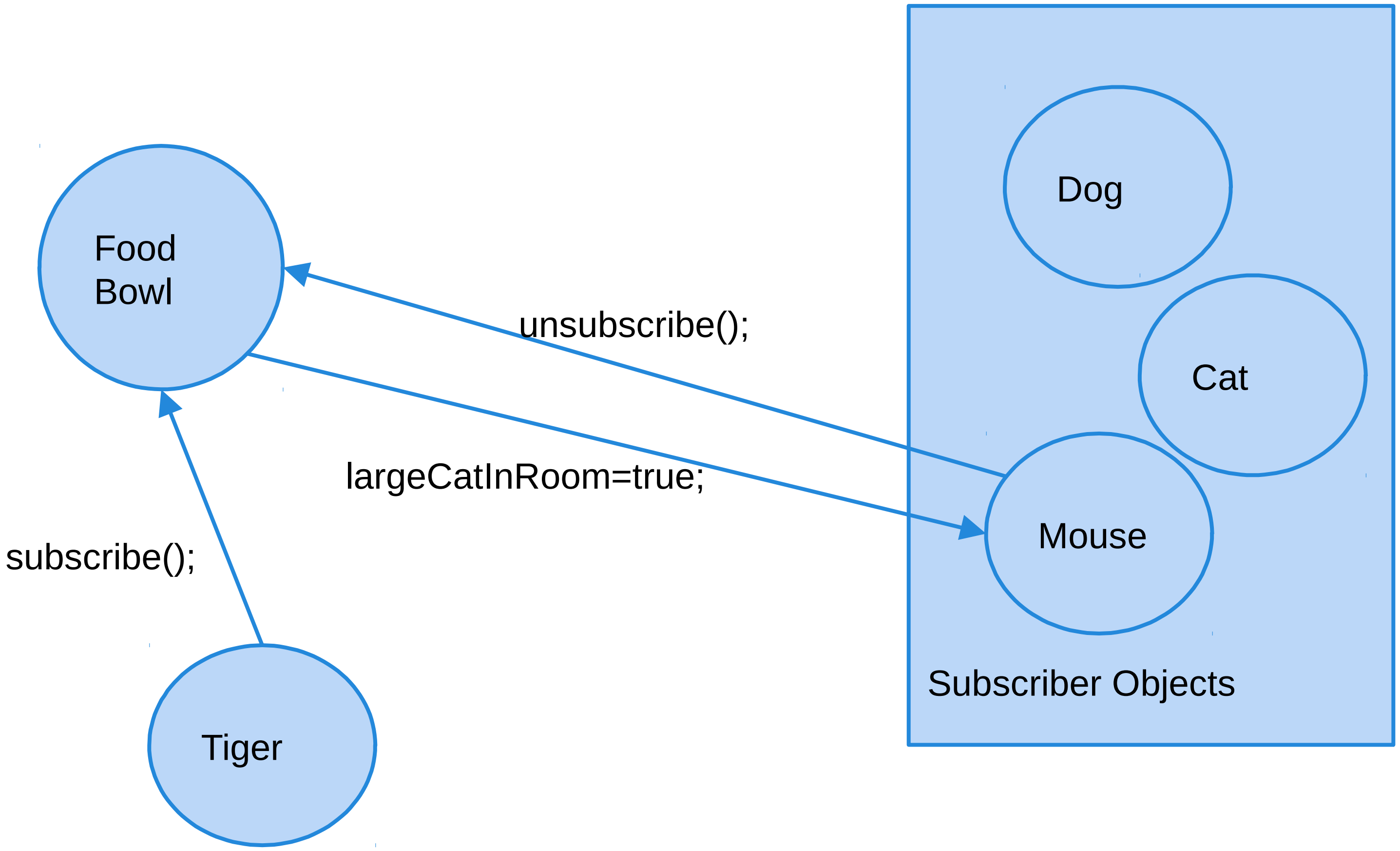
Example - Subscription



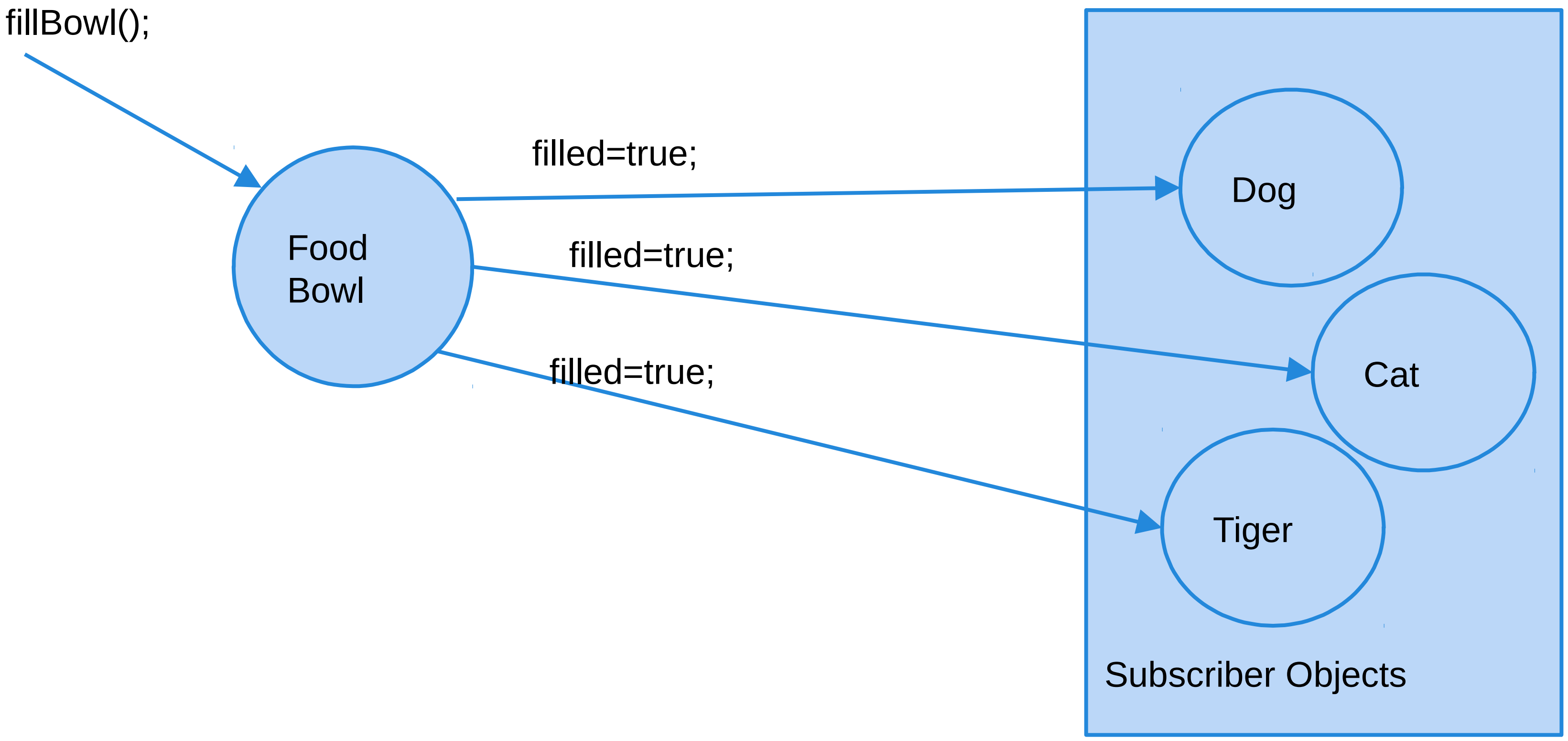
Example - Data Update



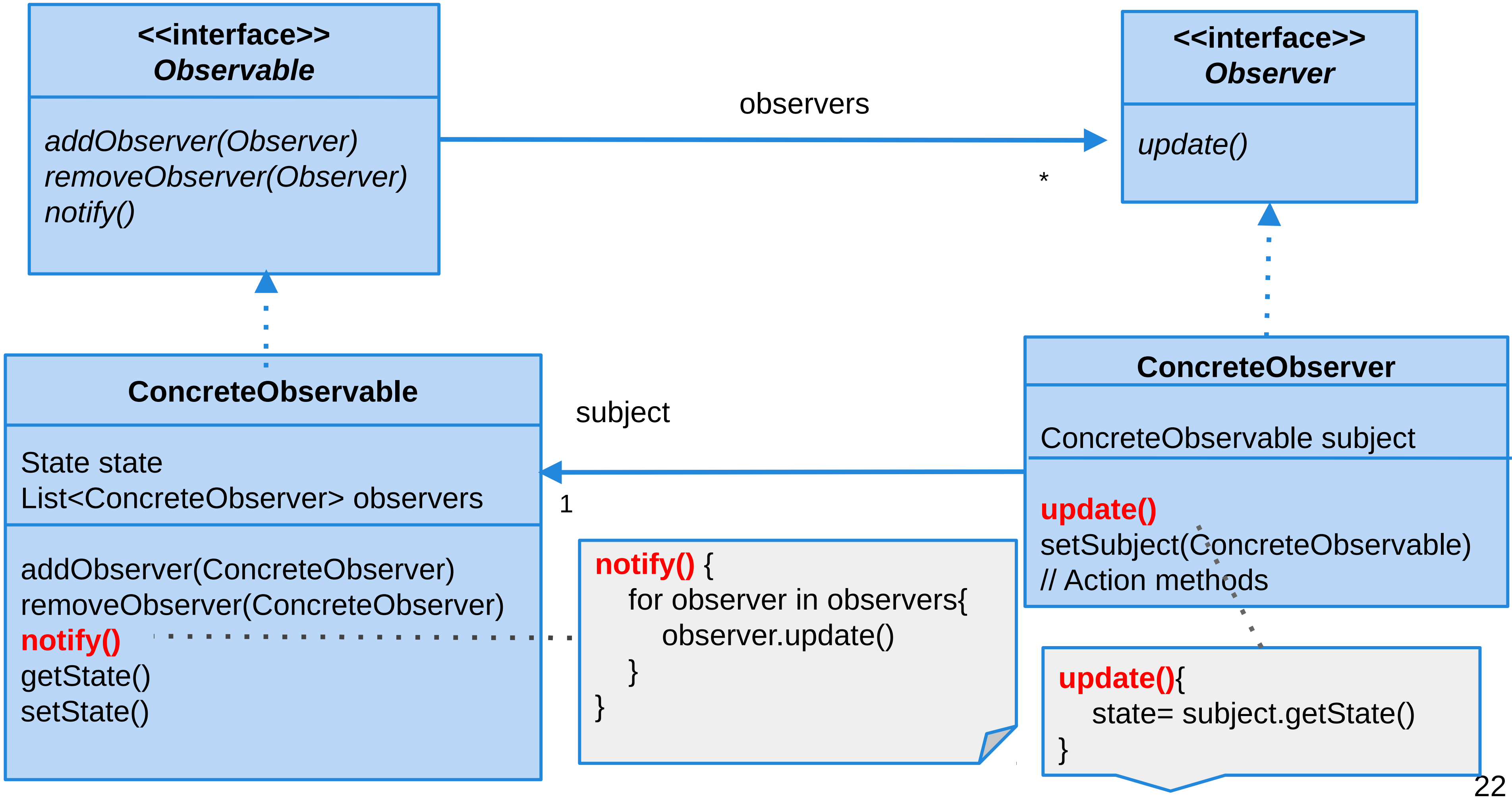
Example - Unsubscribe



Example - Watching for Updates



Observer Pattern - In Practice



```
notify() {
    for observer in observers{
        observer.update()
    }
}
```

```
update(){
    state= subject.getState()
}
```

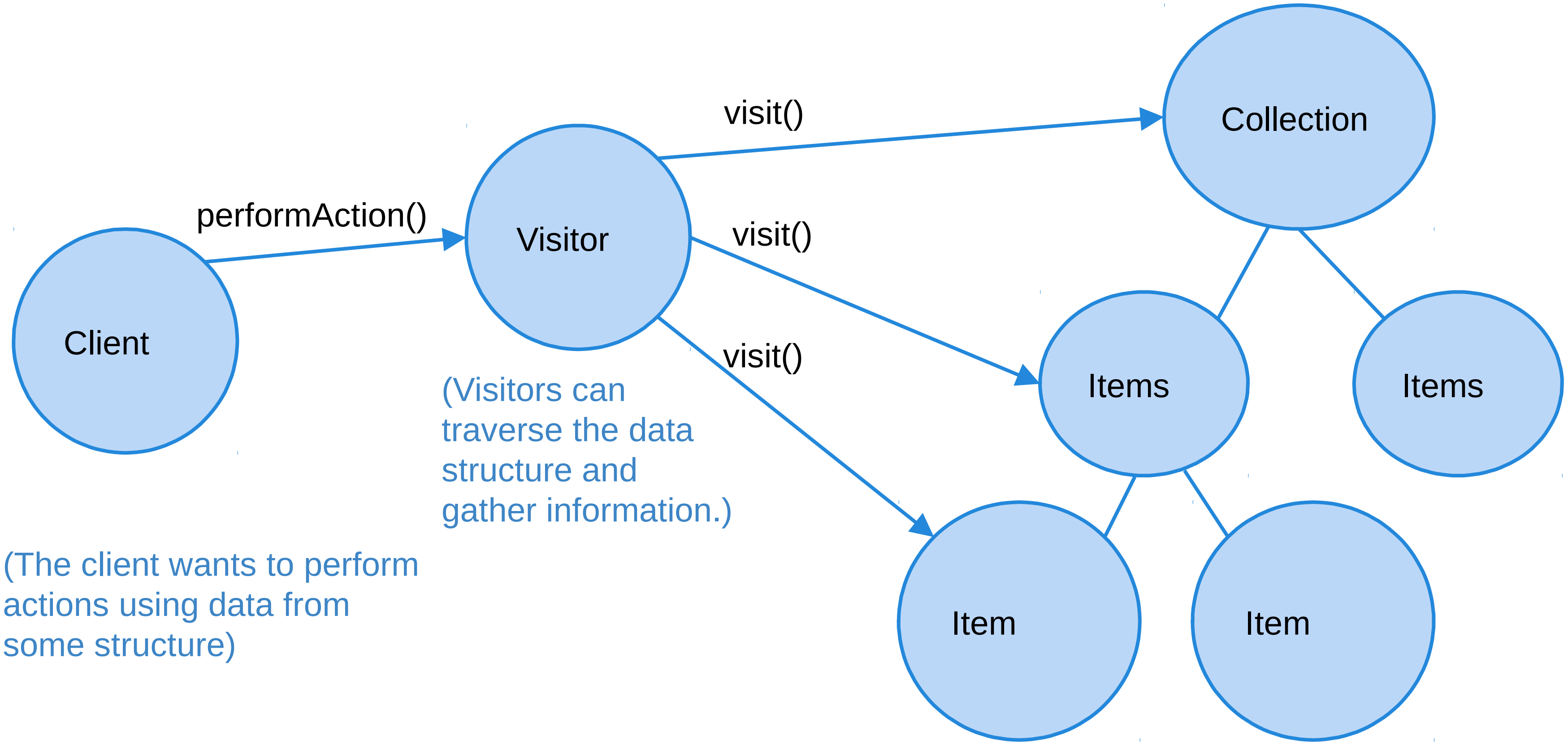
Benefit - Loose Coupling

When objects are loosely coupled, they can interact while lacking knowledge of each other.

1. Can add new observers at any time.
2. Never need to modify subject.
3. Easy code reuse.
4. Easy change.

Visitor Pattern - Motivation

Visitor Pattern - Definition

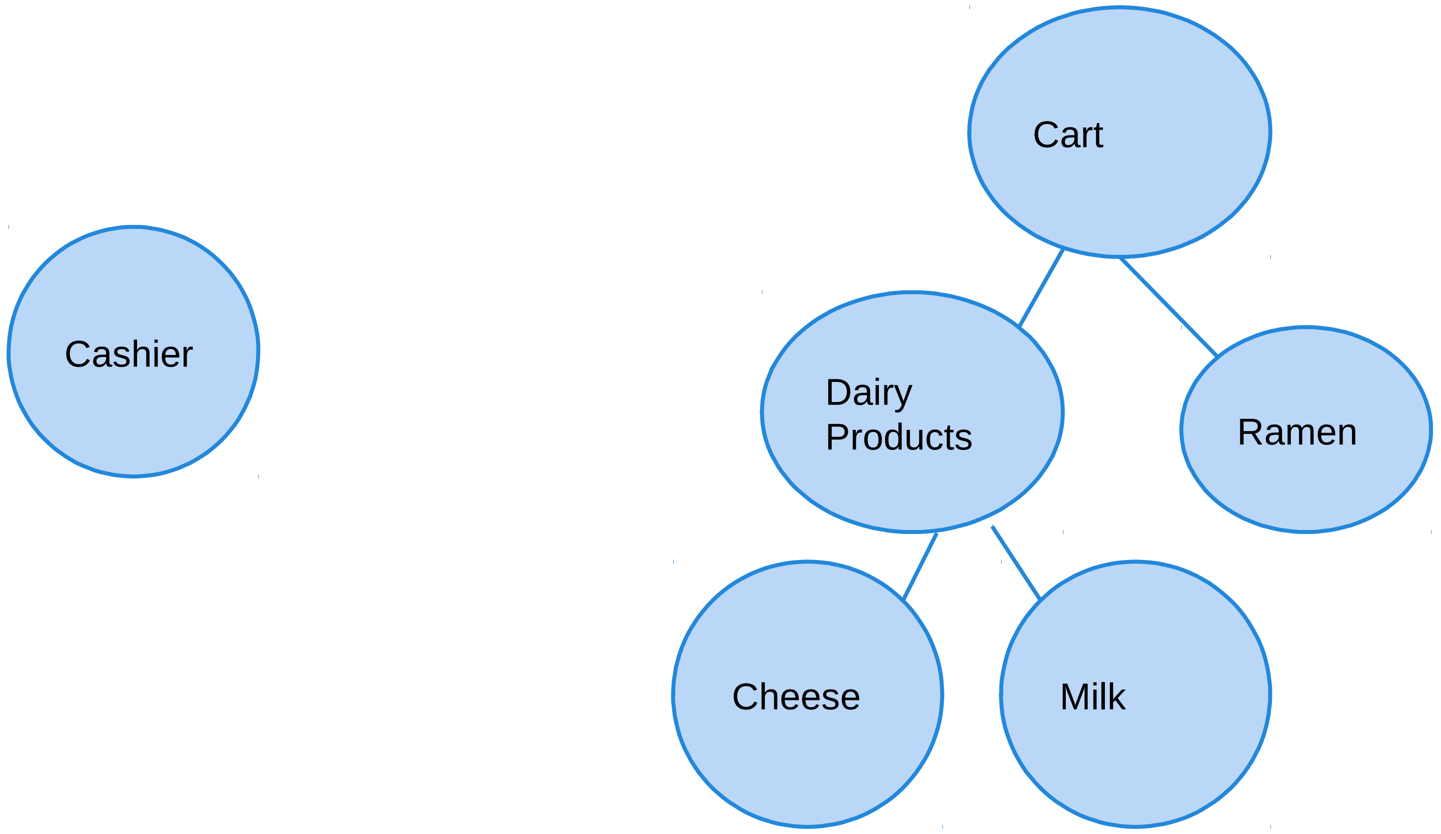


(Visitors can traverse the data structure and gather information.)

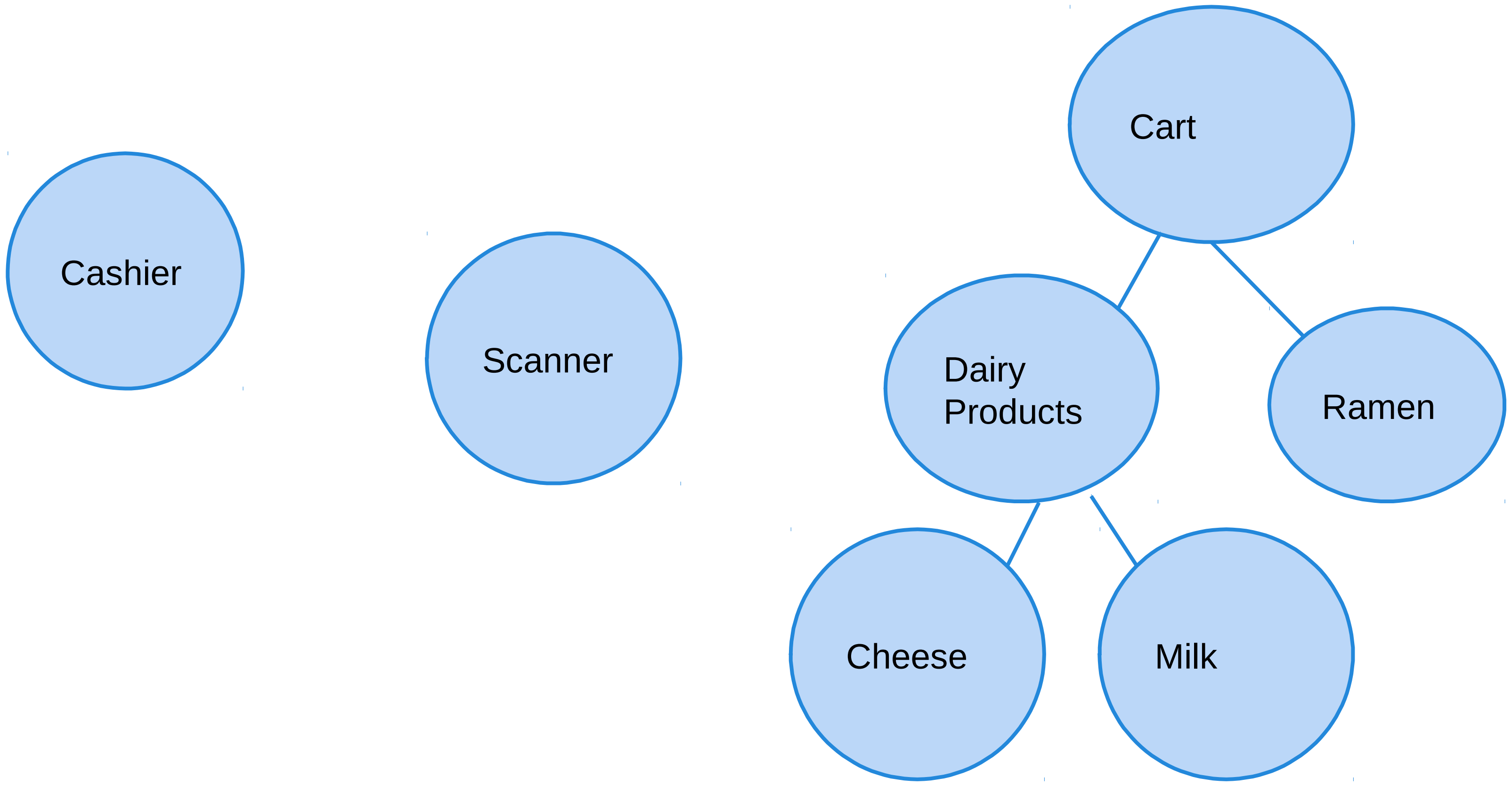
(The client wants to perform actions using data from some structure)

(Composite classes simply need a method to return info.)

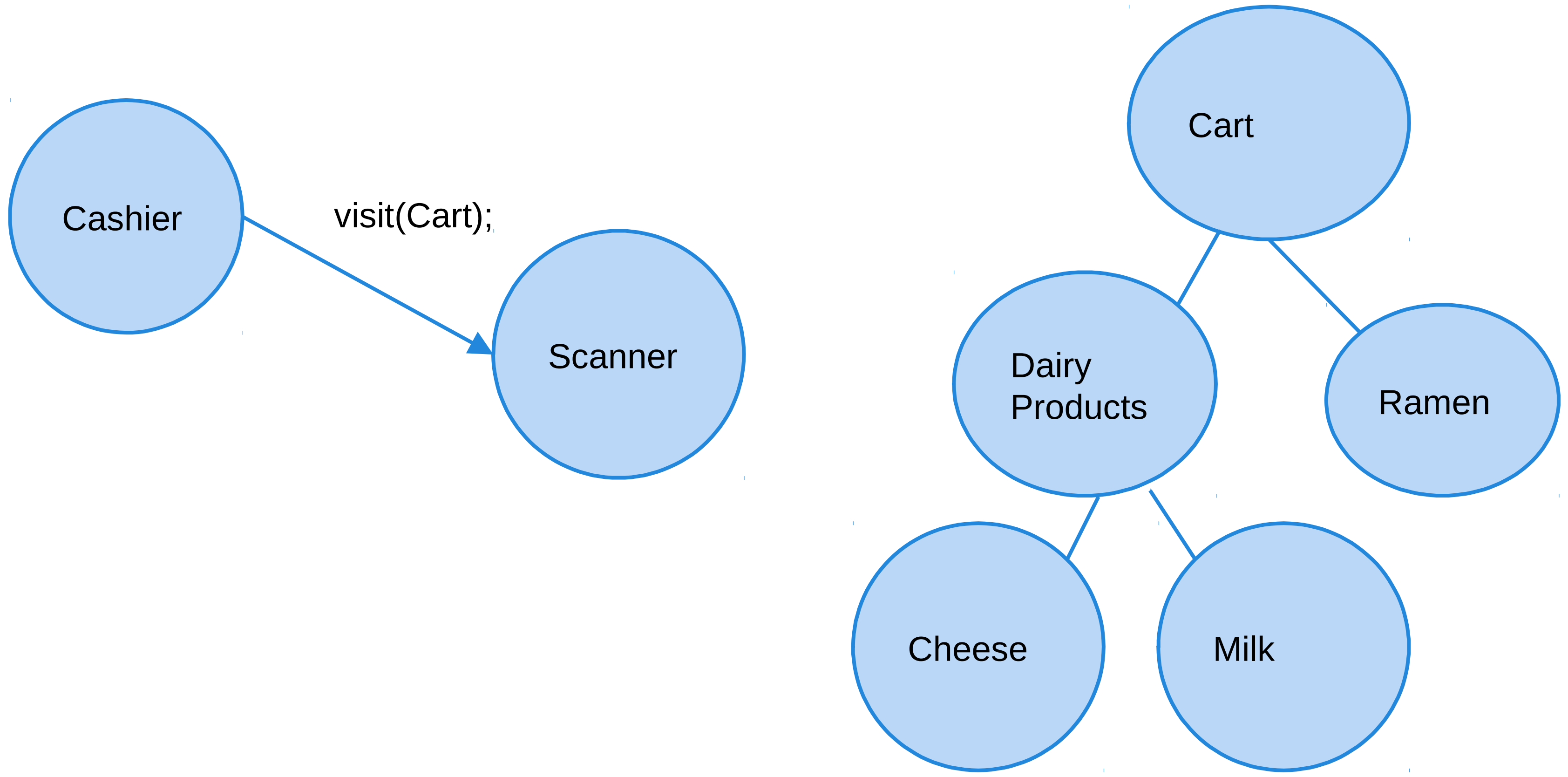
Example - Grocery Cashier



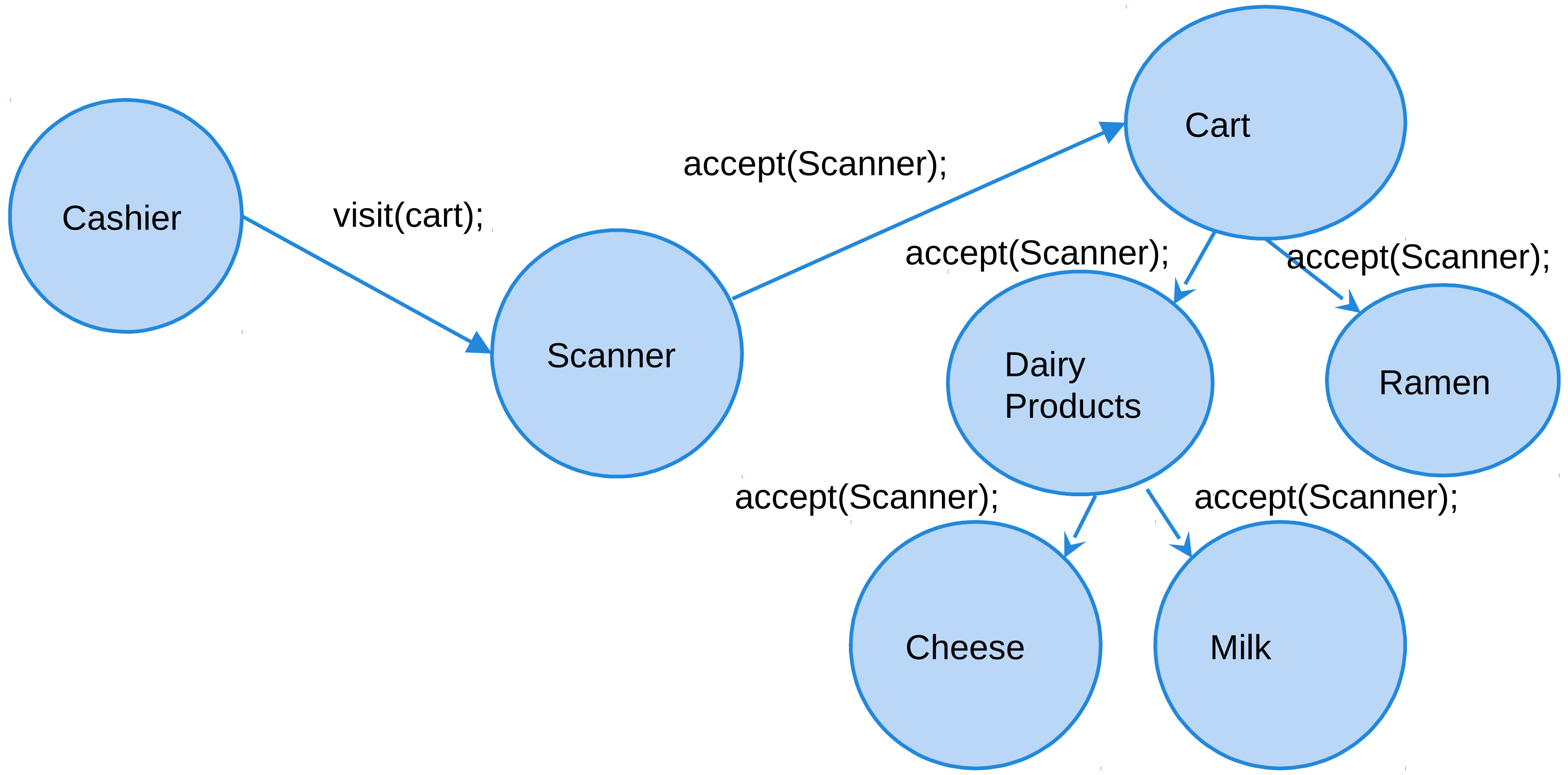
Example - The Visitor



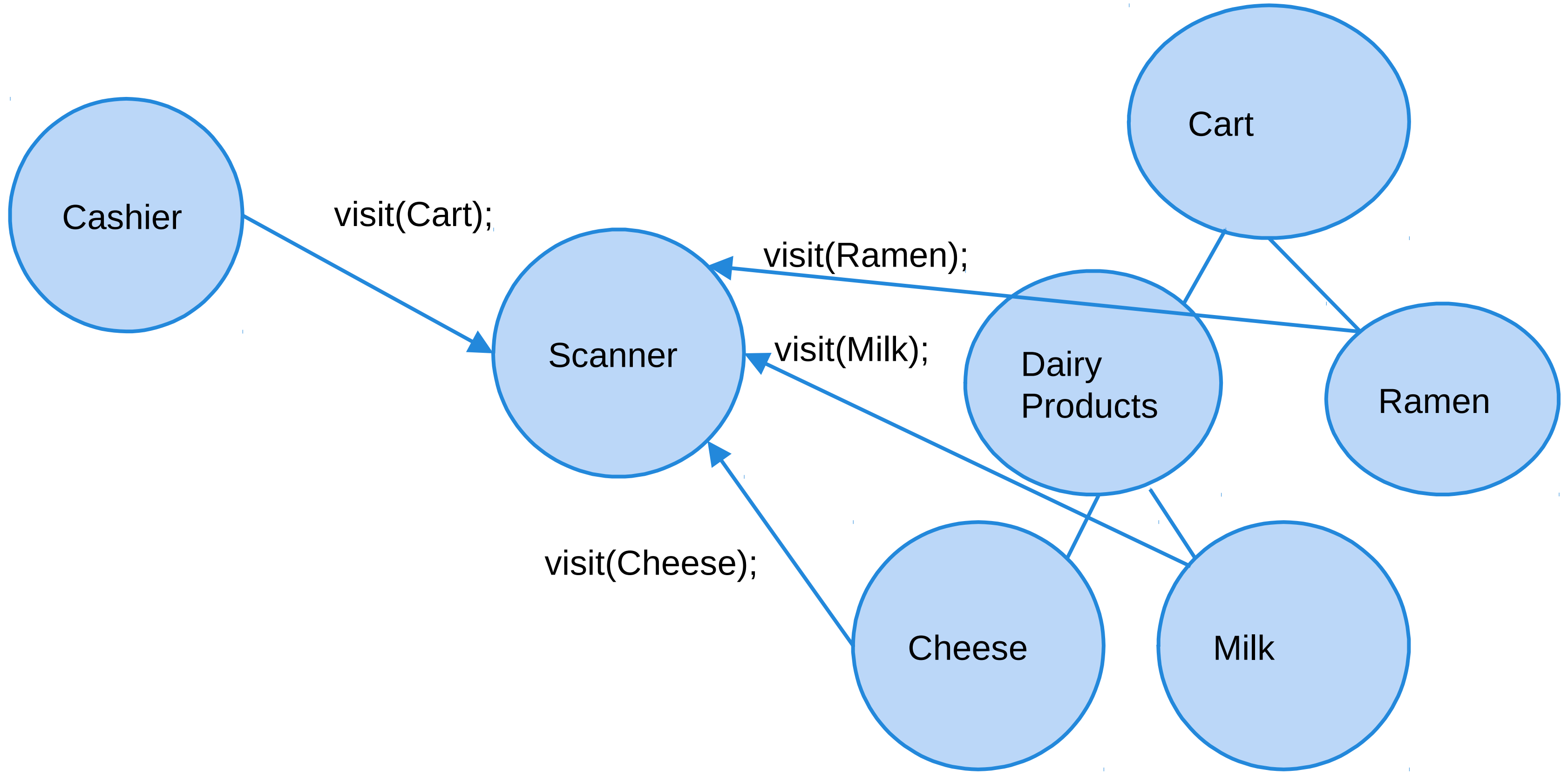
Example - Using The Visitor



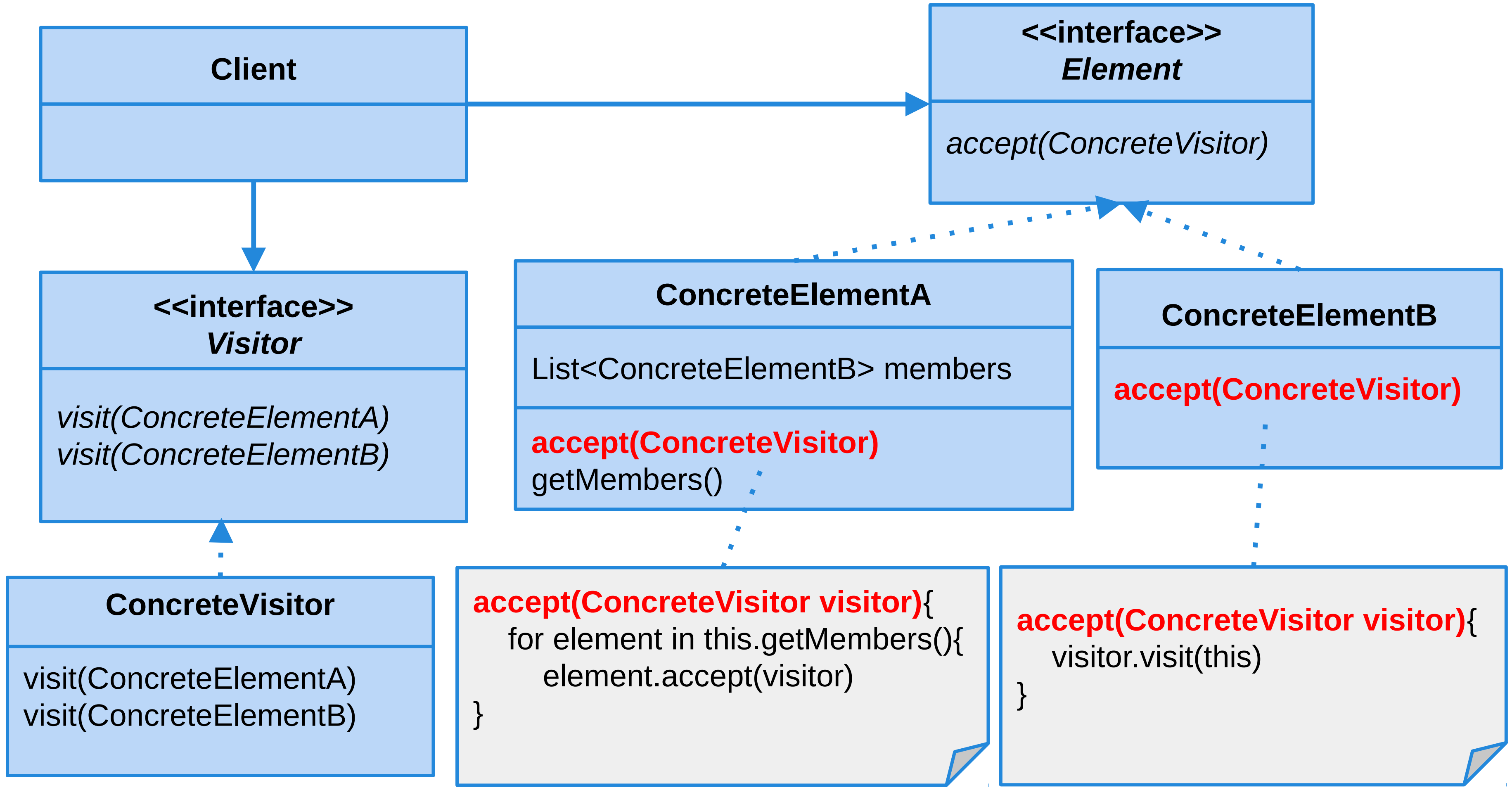
Example - Using The Visitor



Example - Returning To The Visitor



Visitor Pattern - In Practice



```
accept(ConcreteVisitor visitor){
    for element in this.getMembers(){
        element.accept(visitor)
}
```

```
accept(ConcreteVisitor visitor){
    visitor.visit(this)
}
```

Benefits of Visitor Pattern

1. Can add operations to a collection without changing the collection itself.
2. Adding new operations is easy.
3. Operation code is centralized.

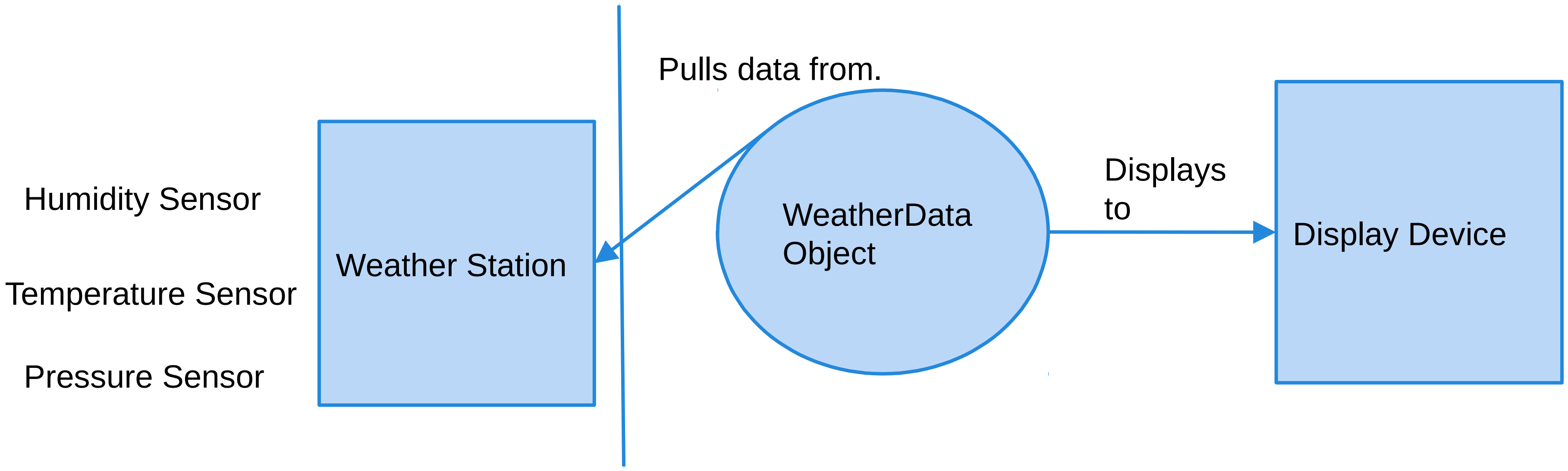
Activity

Building a weather monitoring application.

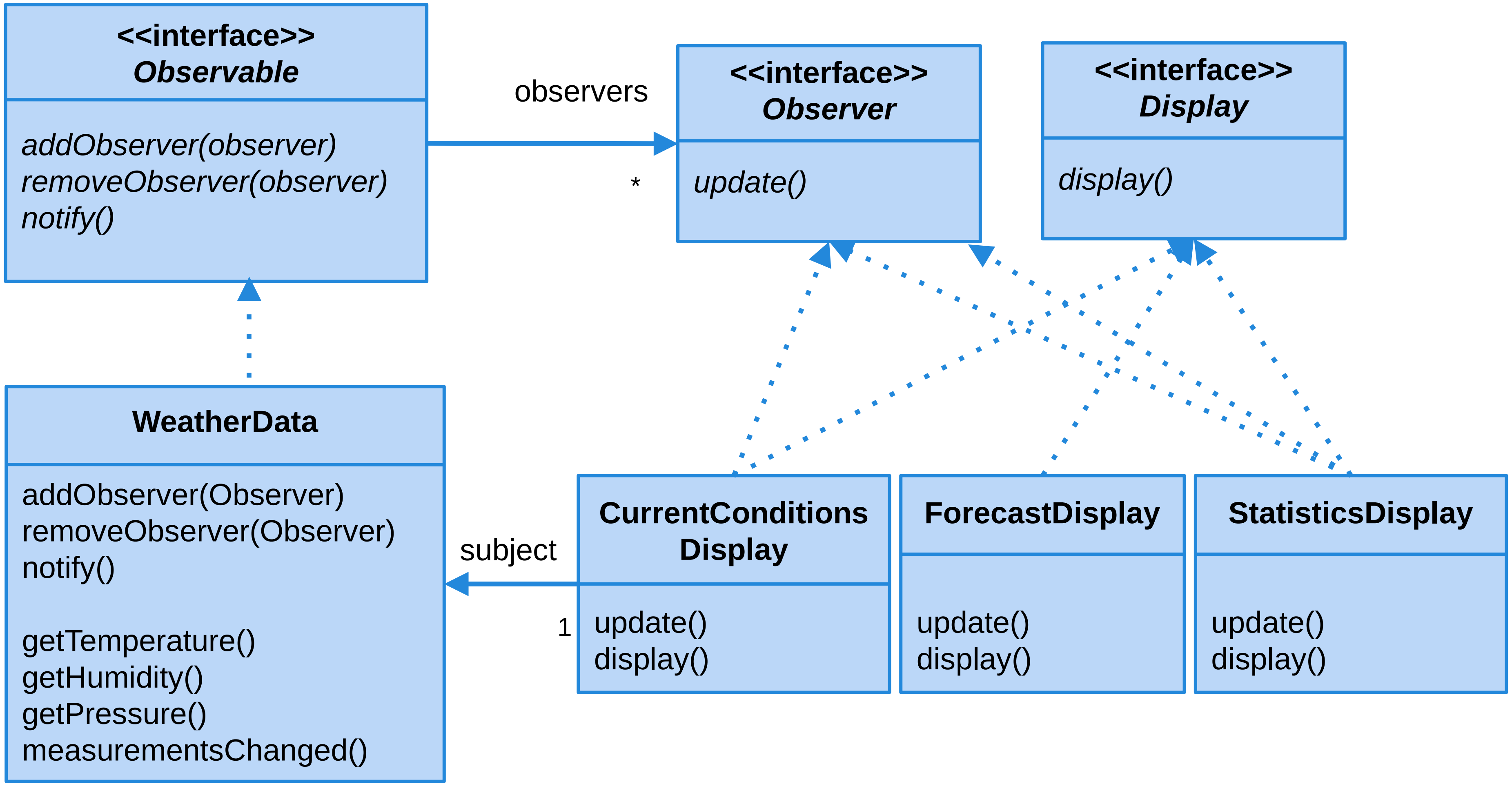
Generates three displays: current conditions, weather statistics, simple forecast.

Design system using either visitor or observer pattern.

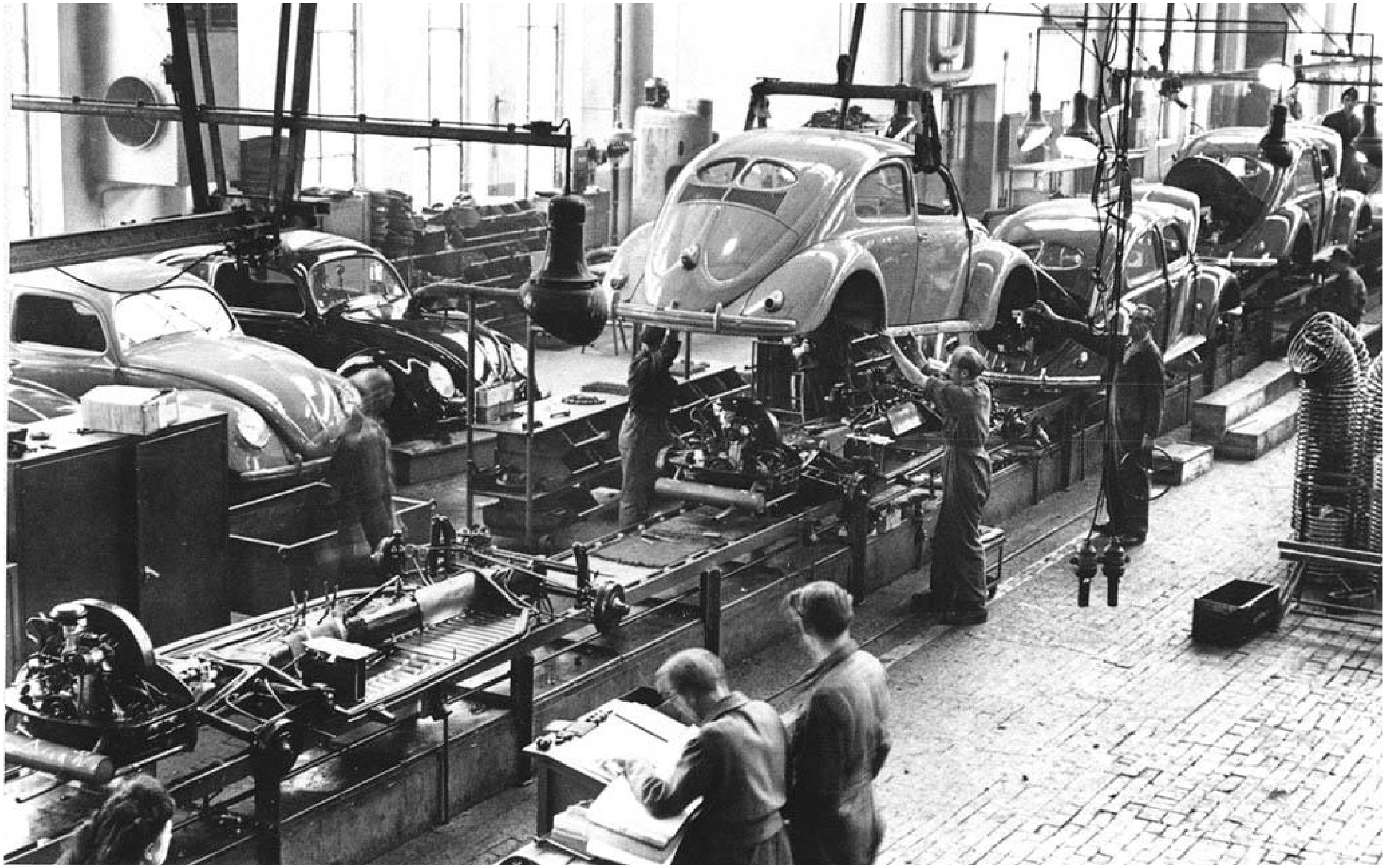
Provided: To Implement:



Activity Solution - Observer Pattern



Factory Pattern - Motivation



Factory Pattern - Motivation

```
Pizza orderPizza(){
    Pizza pizza = new Pizza();

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Factory Pattern - Motivation

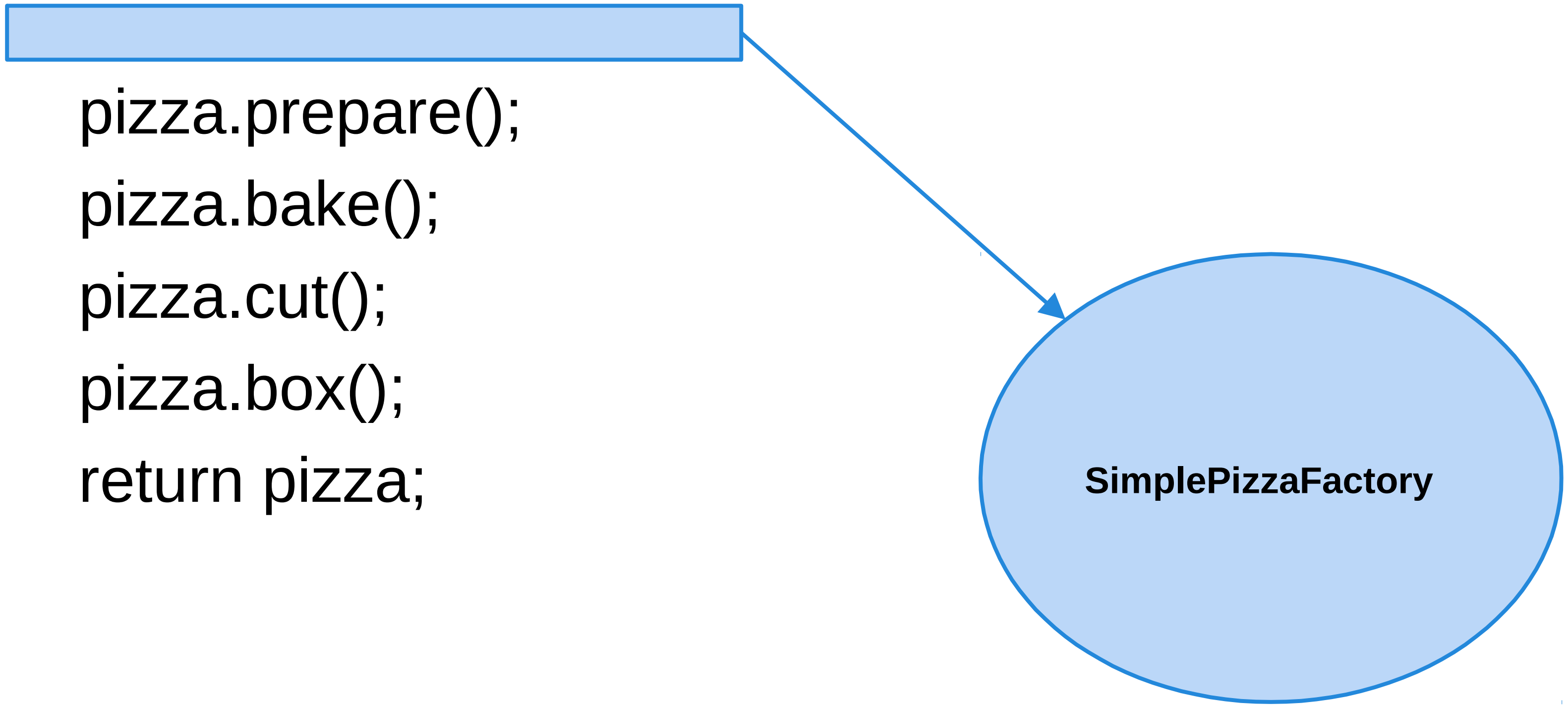
```
Pizza orderPizza(String type){
    Pizza pizza;
    if (type.equals(“cheese”)){
        pizza = new CheesePizza();
    else if(type.equals(“pepperoni”)){
        pizza = new PepperoniPizza();
    }
    // Prep methods
}
```

Factory Pattern - Motivation

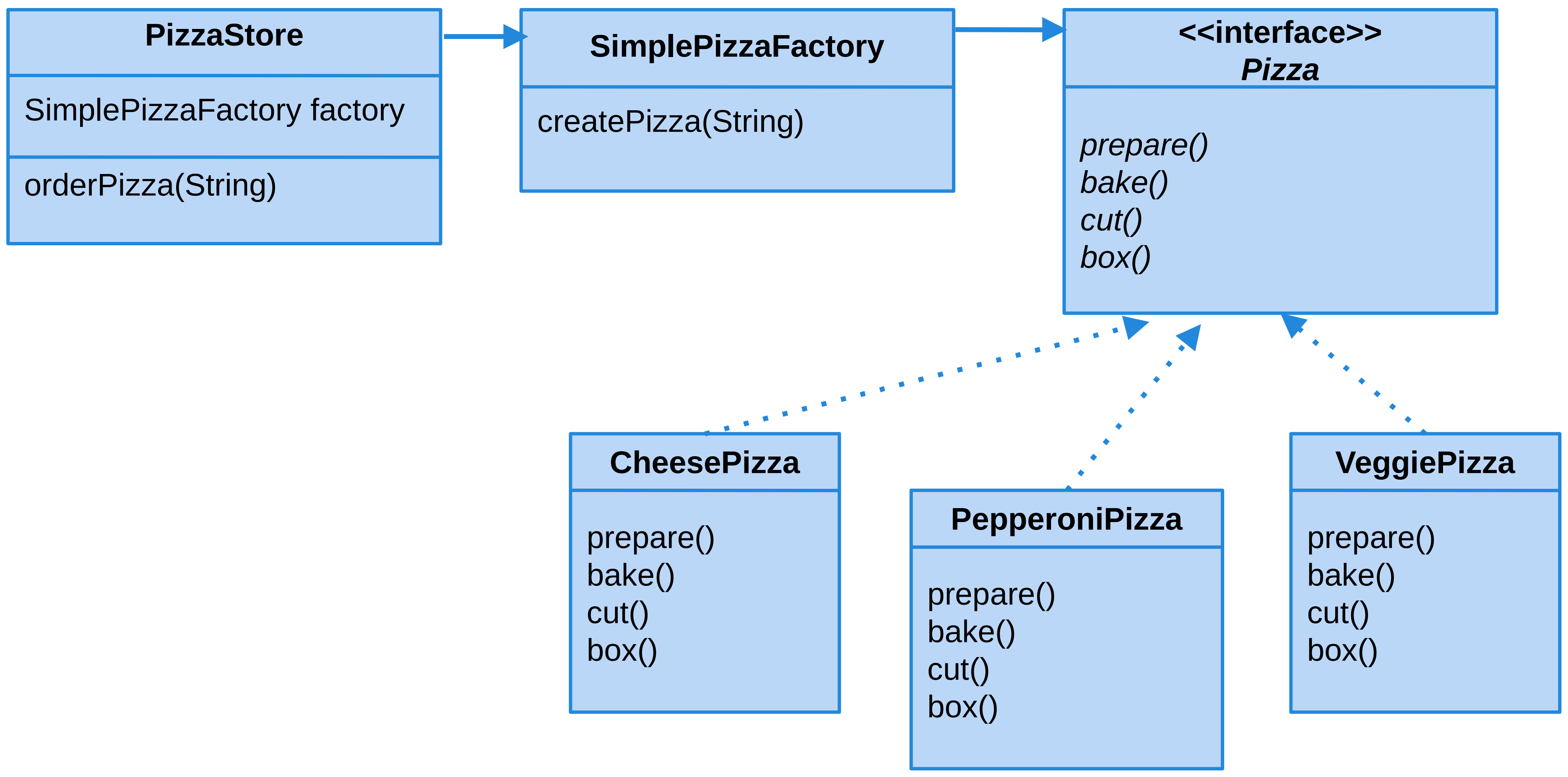
```
Pizza orderPizza(String type){
        Pizza pizza;
        if (type.equals(“cheese”)){
        pizza = new CheesePizza();
        ~~else if(type.equals(“pepperoni”)){~~
        ~~pizza = new PepperoniPizza();~~
        } else if(type.equals(“veggie”)){
        pizza = new VeggiePizza();
        }
        // Prep methods
}
```

Factory Pattern - Motivation

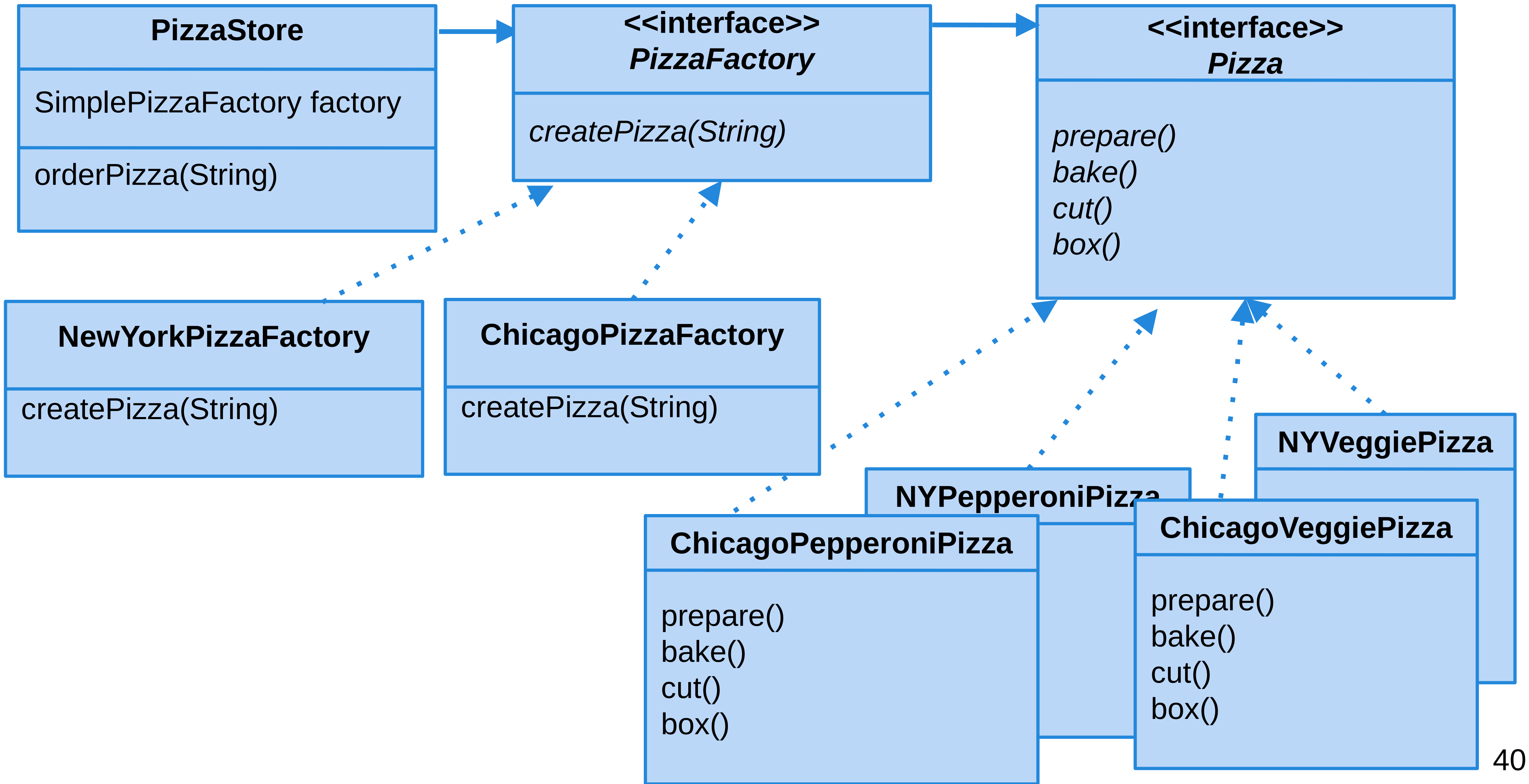
```
Pizza orderPizza(String type){
    Pizza pizza;

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```



The Simple Factory



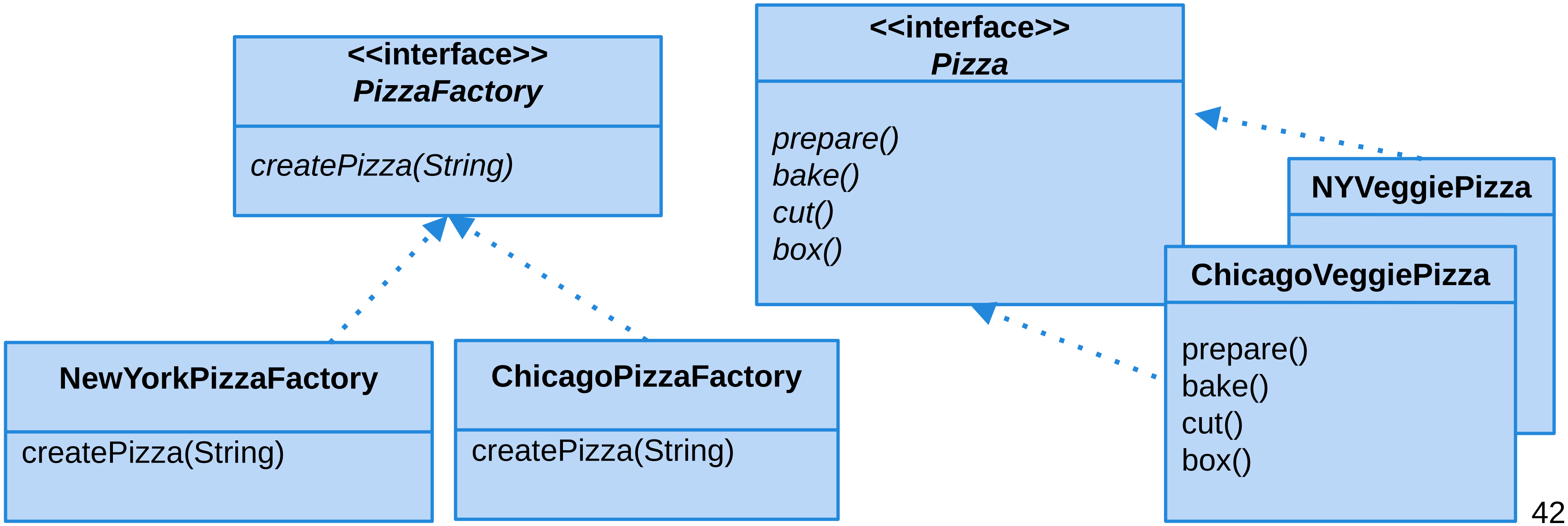
Franchising the Factory



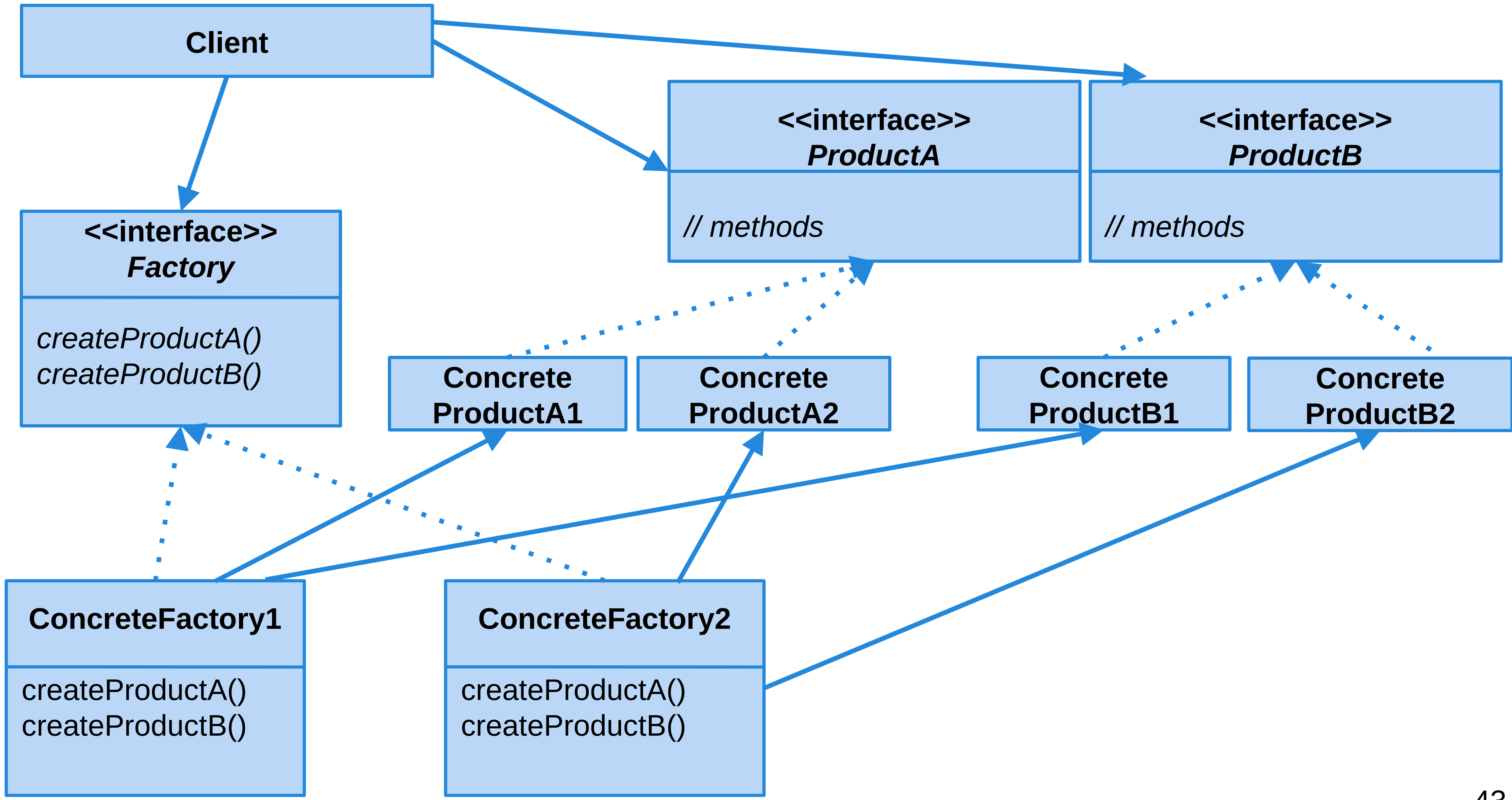
Factory Pattern - Definition

Defines an interface for creating an object, but lets subclasses decide which class to instantiate.

Reason in terms of **creators** and **products**.



Factory Pattern - In Practice



Benefits of Factory Pattern

1. Loose coupling.
2. Creation code is centralized.
3. Easy to add new classes.
4. Lowered class dependency (can depend on abstractions, not concrete classes).

Why not use a design pattern?

What are the drawbacks to using patterns?

- Potentially over-engineered solution.
- Increased system complexity.
- Design inefficiency.

How can we avoid these pitfalls?

Resources

Web:

- oodesign.com
- c2.com/cgi/wiki?PatternIndex

Book:

- Head First Design Patterns, by Eric Freeman & Elizabeth Freeman (free from UMN's Safari Tech Books subscription)
- Design Patterns: Elements of Reusable Object Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Gang of Four)

Recap

When in doubt:

1. Reason about the problem, not the objects.
2. Patterns provide templates for OO design.

Patterns come in many flavors.

Think about patterns and GRADS (hint, hint).

Next time: State Diagrams