

CS 130 SOFTWARE ENGINEERING

DESIGN PATTERNS

INTRODUCTION, STRATEGY,
OBSERVER MEDIATOR

Professor Miryung Kim
UCLA Computer Science

RECAP: IH PRINCIPLE

- ▶ Module: an independent work assignment
- ▶ Goal of modularization: parallel development
- ▶ Interface of a module should be about design decisions that are unlikely to change. (method name, input and output argument types and their names should be determined based on what is unlikely to change.)
- ▶ Implementation of an interface: should be about *secrets— design decisions that are likely to change*

- ▶ What is a good modularization? (a wrong question to ask).
- ▶ What is a better modularization between design alternatives?
- ▶ A criteria of assessing modularization:
 - ▶ Anticipate likely change scenarios
 - ▶ List which interfaces must be modified to accommodate the anticipated changes.
 - ▶ The more scattered and de-localized changes are, the worse the design is.

ON THE FLY QUIZ QUESTIONS

- ▶ Is “using a public getter method” and keeping a field private – about IH principle?
- ▶ Is “using a dynamic call instead of a static call” about IH principle?
- ▶ Is “the internal data structure layout “ being exposed through a getter interface, is it about IH principle?

ON THE FLY QUIZ QUESTIONS

- ▶ Is “using a public getter method” and keeping a field private – about IH principle? No
- ▶ Is “using a dynamic call instead of a static call” about IH principle? No
- ▶ Is “the internal data structure layout “ being exposed through a getter interface, is it about IH principle? Yes

AGENDA

- ▶ Strategy Pattern (Chapter 1 in Head First)
- ▶ Observer Pattern (Chapter 2 in Head First)
- ▶ Mediator Pattern
- ▶ Index Cards

DESIGN PATTERN, DEFINED

- ▶ “A solution to a problem in a context.”
- ▶ “A language for communication solutions with others.”
- ▶ Pattern languages exist for many problems, but we focus on design
- ▶ Best known: “Gang of Four” (Gamma, Helm, Johnson, Vlissides)
 - ▶ *Design Patterns: Elements of Reusable Object-Oriented Software*

CAVEATS

- ▶ Design patterns are not a substitute for thought
- ▶ Class names and directory structures do not equal good design
- ▶ Design patterns have tradeoffs
 - ▶ We'll talk about the *mediator* pattern. It does not remove complexity in interactions but just provides a structure for centralizing it.
- ▶ Design patterns depend on the programming language
 - ▶ Certain language restrictions may necessitate certain patterns (e.g., patterns related to object creation and destruction)

MOTIVATION FOR DESIGN PATTERNS

- ▶ They provide an abstraction of the design experience
 - ▶ Can often serve as a reusable base of experience
- ▶ They provide a common vocabulary for discussing complete system designs
- ▶ They reduce system complexity by naming abstractions
 - ▶ Thereby increasing program comprehension and reducing learning time with a new piece of code
- ▶ They provide a target for the reorganization or refactoring of class hierarchies

PIECES OF A DESIGN PATTERN

- ▶ Pattern name and classification
- ▶ Intent
- ▶ Also known as
- ▶ Justification
- ▶ Applicability
- ▶ Structure
- ▶ Participants
- ▶ Collaborations
- ▶ Consequences
- ▶ Implementation
- ▶ Sample code
- ▶ Known uses
- ▶ Related patterns

RATIONALE FOR DESIGN PATTERNS

- ▶ Shared pattern vocabularies are powerful
- ▶ Patterns allow you to say more with less
- ▶ Reusing tried and tested methods
- ▶ Focus is on developing flexible, maintainable programs

STRATEGY

SIMUDUCK

The ducks
should be
able to swim!

We should
have Mallard
ducks!

The ducks
should quack!

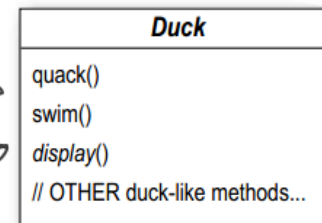
What about a
redheaded
duck?

Don't ducks
fly, too?



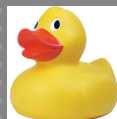
All ducks quack and swim, the
superclass takes care of the
implementation code.

Each duck subtype
is responsible for
implementing its own
display() behavior
for how it looks on
the screen.

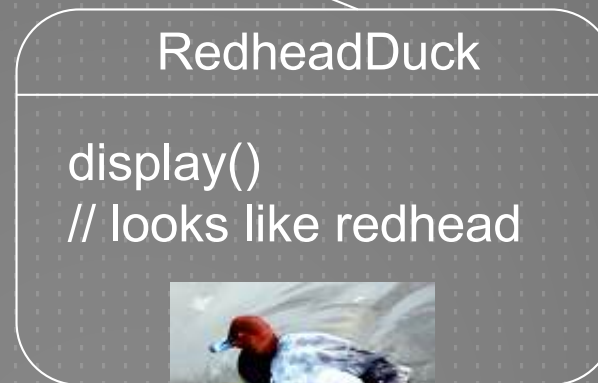


The display() method is
abstract, since all duck
subtypes look different.

Lots of other types of ducks
inherit from the Duck class.

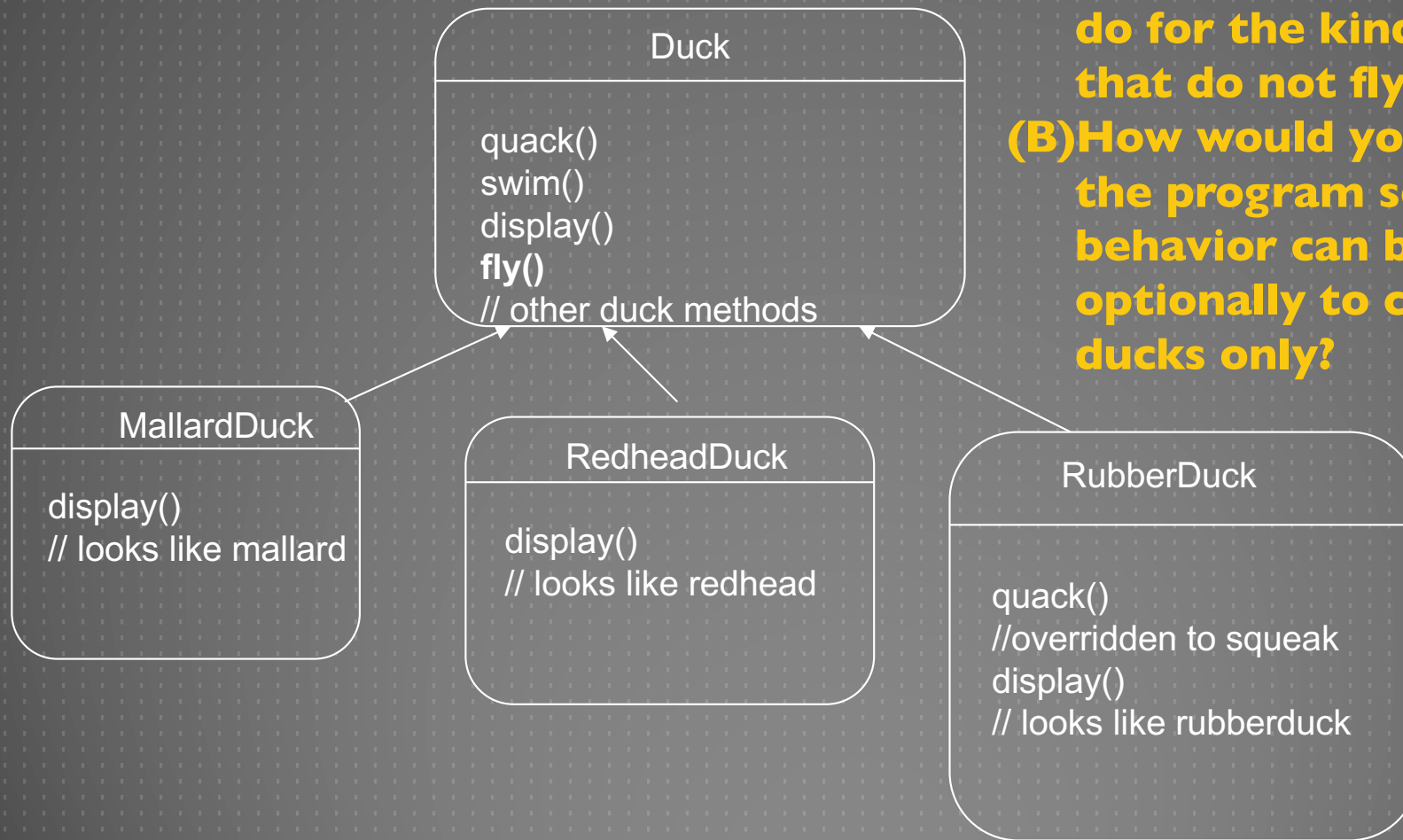


SIMPLE SIMULATION OF DUCK BEHAVIOR



Other duck
types

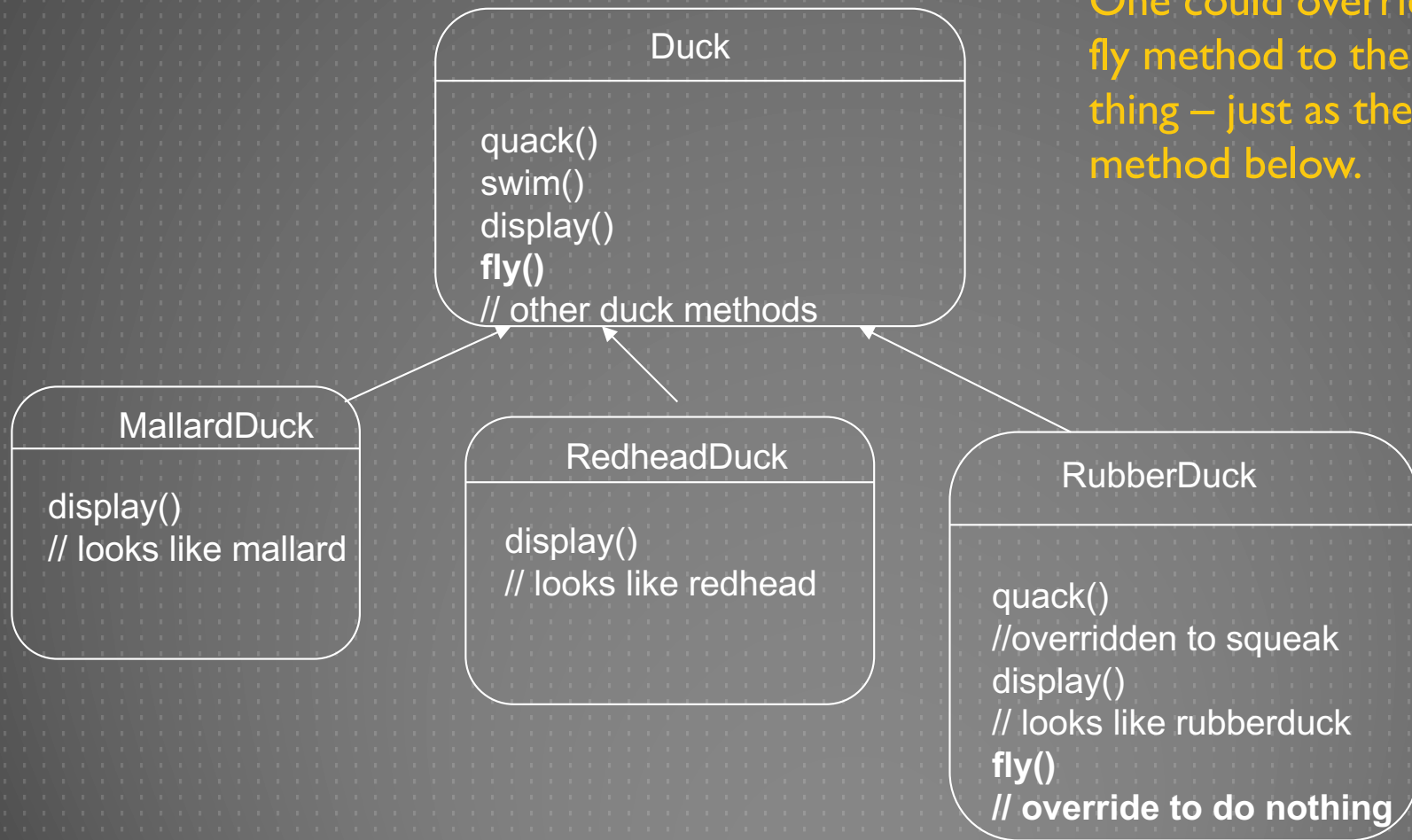
THINK-PAIR-SHARE



(A) If we add a “fly” method to the duck, what will we do for the kinds of ducks that do not fly?

(B) How would you change the program so that “fly” behavior can be added optionally to certain ducks only?

TRADEOFFS IN USE OF INHERITANCE AND MAINTENANCE



One could override the fly method to the appropriate thing – just as the quack method below.

EXAMPLE COMPLICATED: ADD A WOODEN DECOY DUCKS TO THE MIX

DecoyDuck

```
quack(){  
  // override to do nothing  
}  
display()  
  // display decoy duck  
fly (){  
  //override to do nothing  
}
```

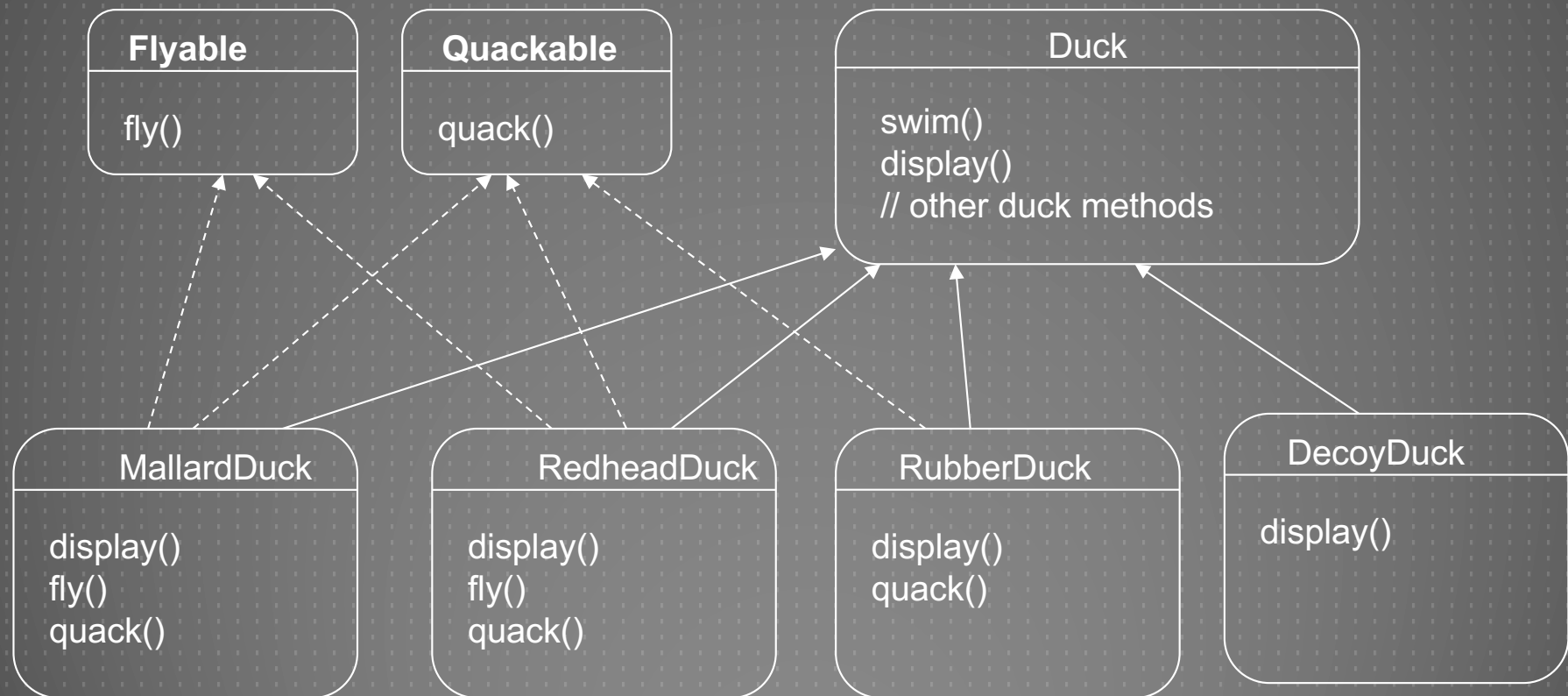


Inheritance is not always the right answer. Every new class that inherits unwanted behavior needs to be overridden.

How about using interfaces instead?

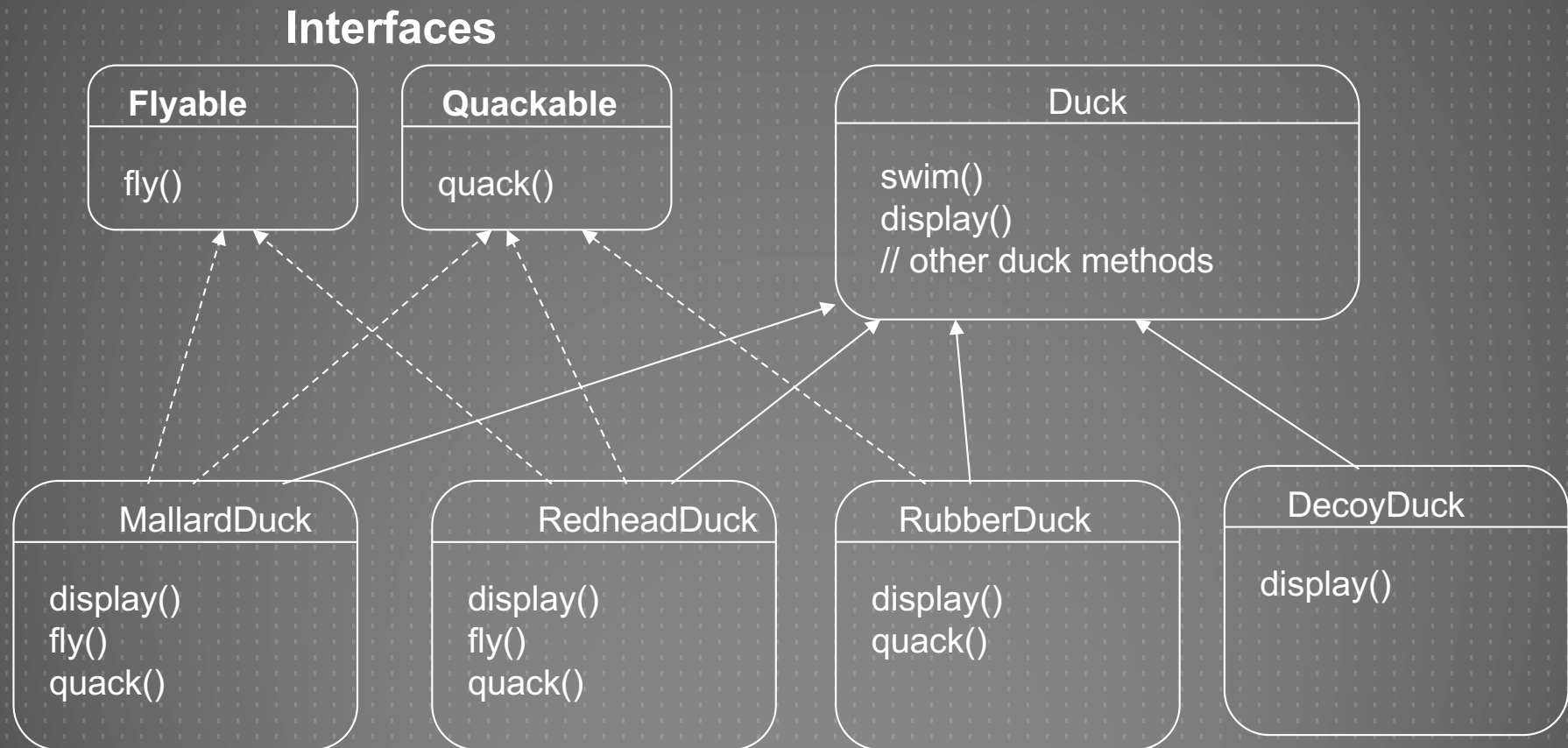
DUCK SIMULATION RECAST USING INTERFACES.

Interfaces



THINK-PAIR-SHARE

WHAT ARE PROS AND CONS OF THIS DESIGN?

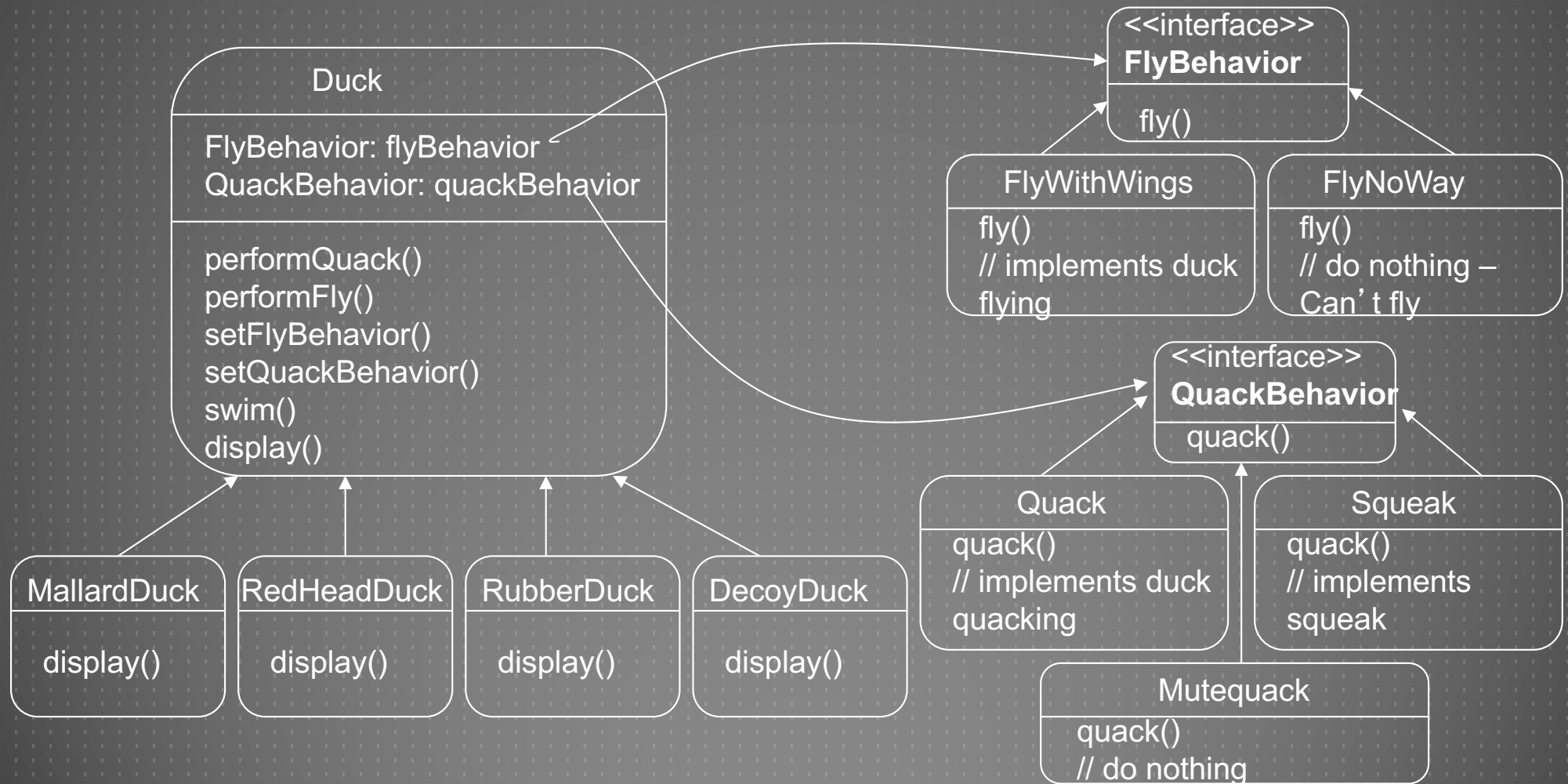


PROS AND CONS

- ▶ Not all inherited methods make sense for all subclasses – hence inheritance is not the right answer
- ▶ But by defining interfaces, every class that needs to support that interface needs to implement that functionality... destroys code reuse!
- ▶ So if you want to change the behavior defined by interfaces, every class that implements that behavior may potentially be impacted

And....

ALTERNATIVE DESIGN: STRATEGY



```
public abstract class Duck {  
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
  
    public Duck() {  
    }  
    public void setFlyBehavior (FlyBehavior fb) {  
        flyBehavior = fb;  
    }  
    public void setQuackBehavior(QuackBehavior qb) {  
        quackBehavior = qb;  
    }  
    abstract void display();  
  
    public void performFly() {  
        flyBehavior.fly();  
    }  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
    public void swim() {  
        System.out.println("All ducks float, even decoys!");  
    }  
}
```

```
package headfirst.strategy;

public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new
FlyNoWay();
        quackBehavior = new
Quack();
    }

    public void display() {
        System.out.println("I'm a
model duck");
    }
}
```

```
package headfirst.strategy;
```

```
public class FlyRocketPowered implements  
FlyBehavior {  
    public void fly() {  
        System.out.println("I'm  
flying with a rocket");  
    }  
}
```



```
package headfirst.strategy;

public class MiniDuckSimulator {

    public static void main(String[] args) {

        MallardDuck    mallard = new MallardDuck();
        RubberDuck     rubberDuckie = new RubberDuck();
        DecoyDuck       decoy = new DecoyDuck();

        ModelDuck       model = new ModelDuck();

        mallard.performQuack();
        rubberDuckie.performQuack();
        decoy.performQuack();

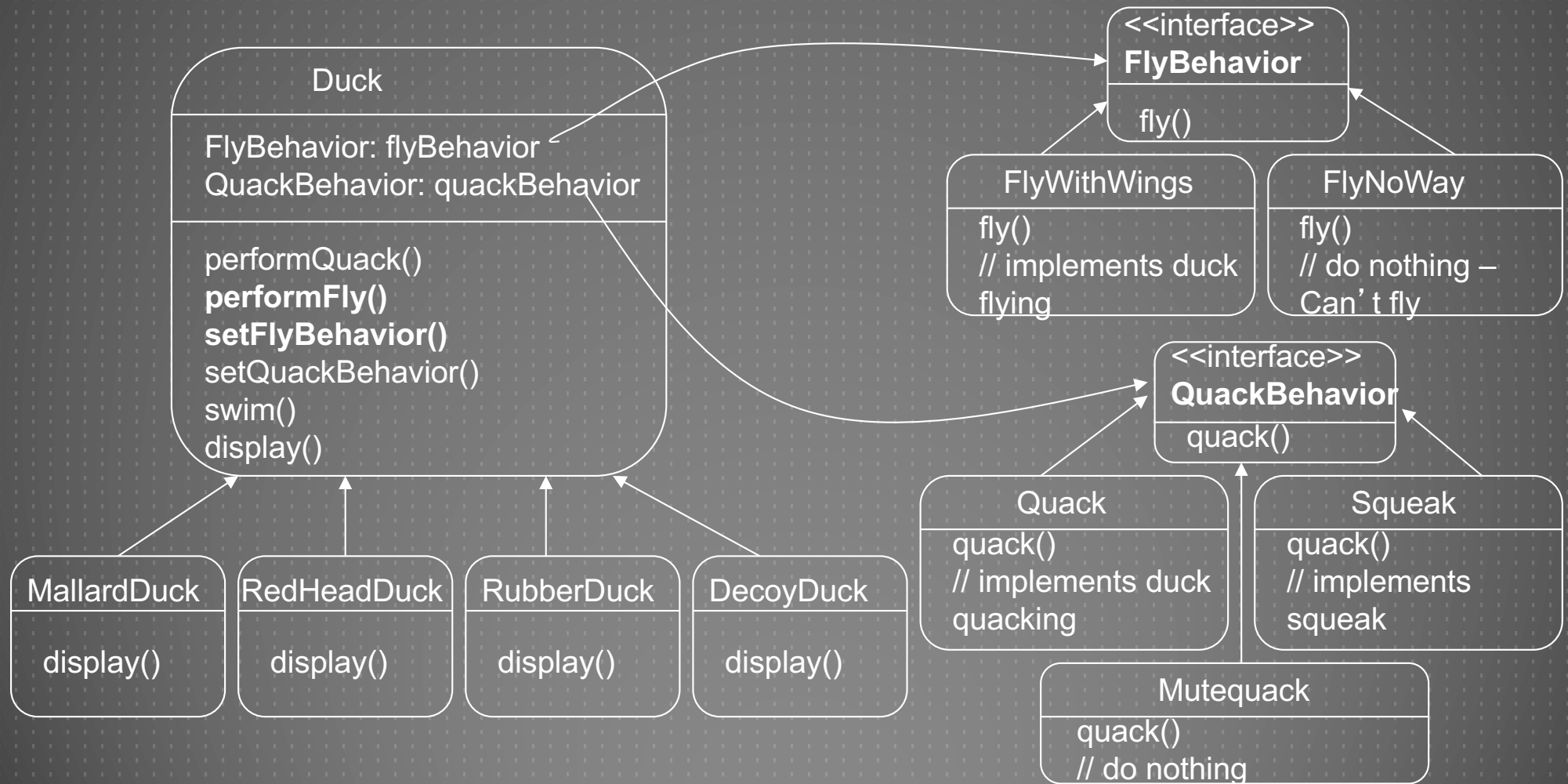
        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered());
        model.performFly();

    }
}
```

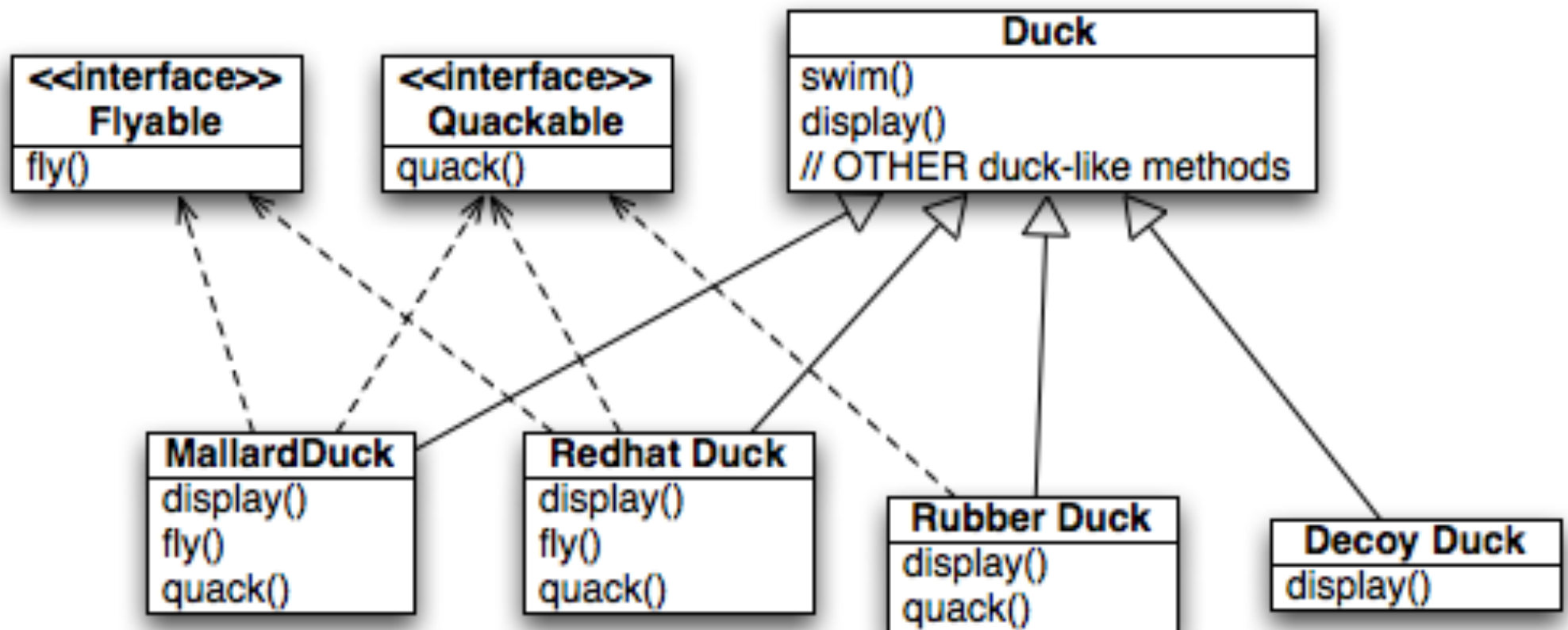
THE STRATEGY PATTERN

The Strategy Pattern defines a family of algorithms, Encapsulates each one, and makes them **interchangeable at runtime**. Strategy lets the algorithm vary independently from clients that use it.

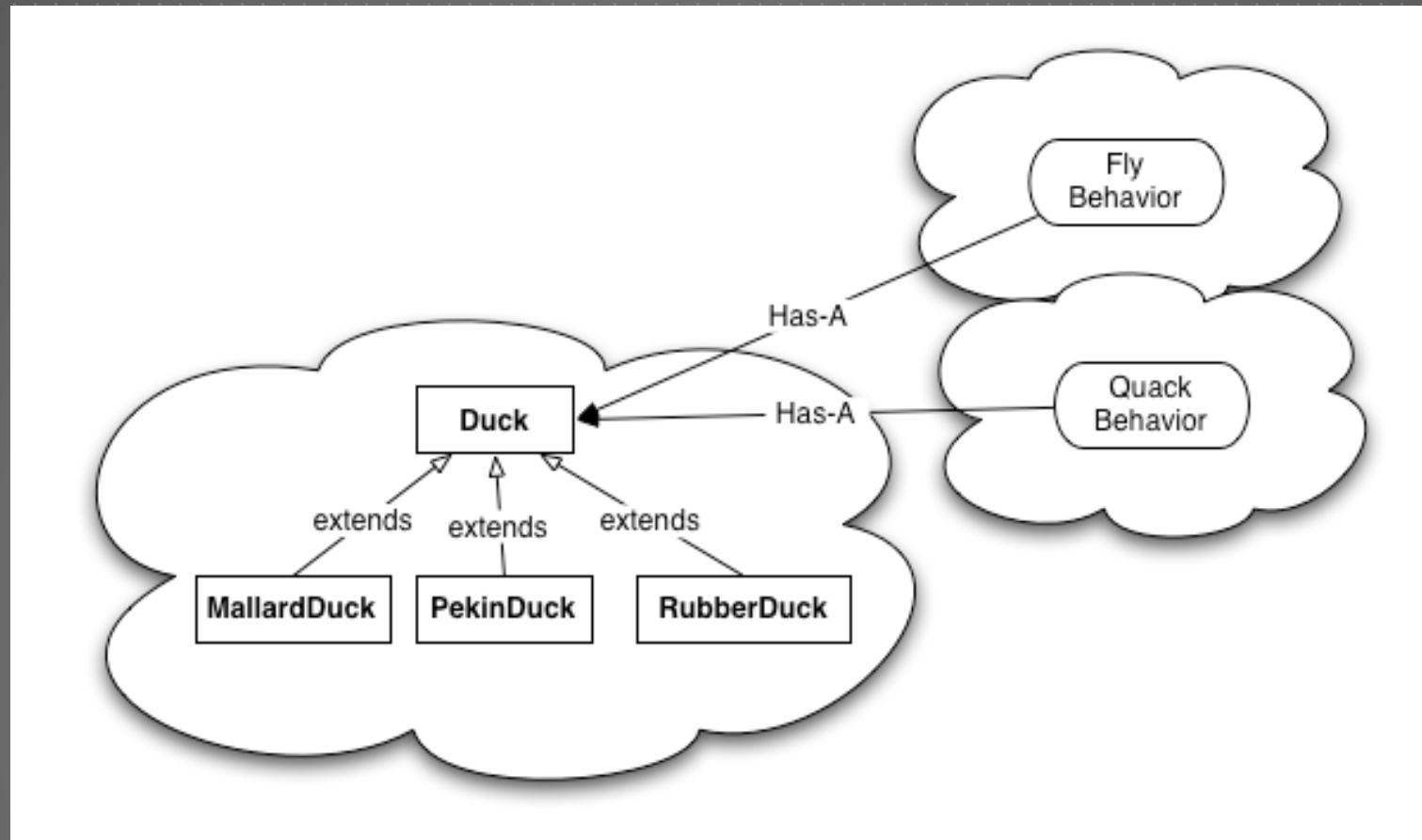
RECAP QUESTION 1: WHAT IS THE STRATEGY AND CONCRETE STRATEGY CLASSES RESPECTIVELY?



RECAP QUESTION 2: WHAT ARE THE DISADVANTAGE OF THIS DESIGN?



RECAP QUESTION 3: WHICH DECISIONS ARE LIKELY TO VARY ACCORDING TO THIS DESIGN?

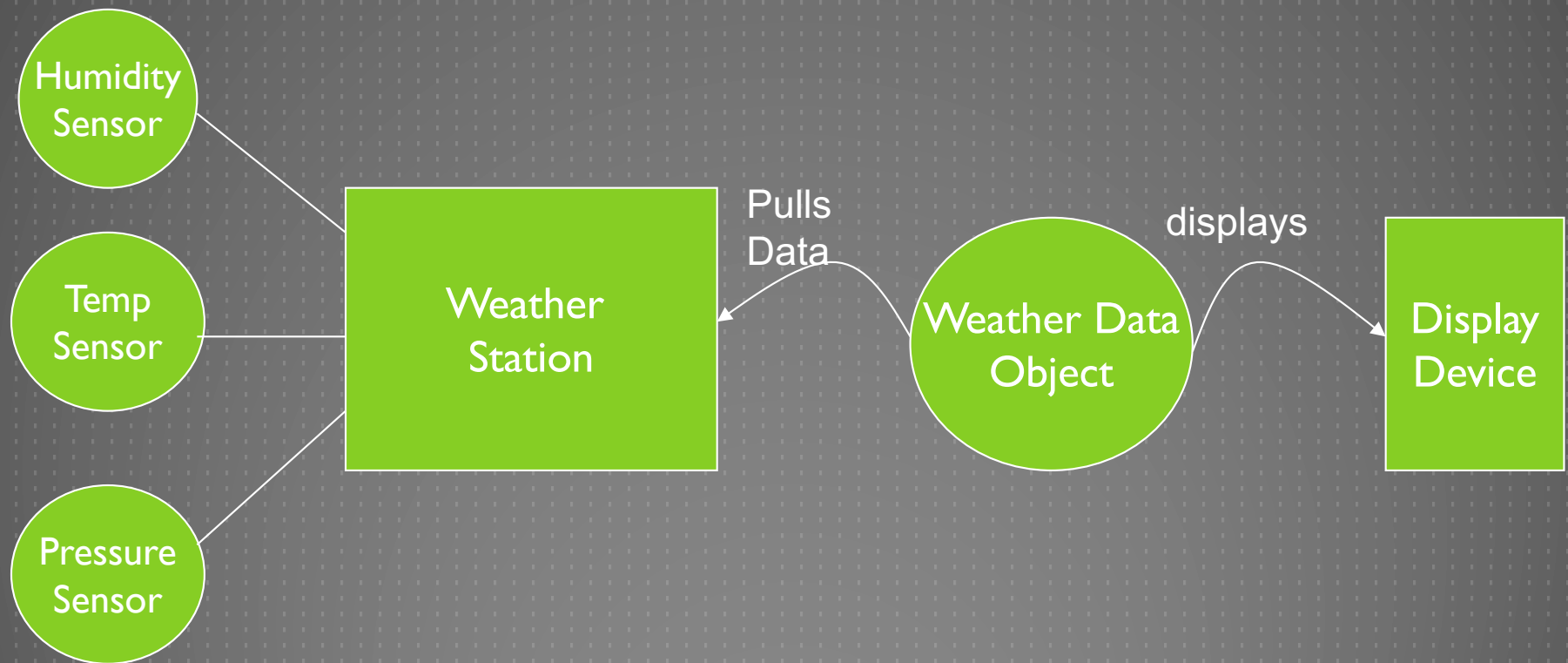


WHAT WAS THE “INTENT” OF STRATEGY

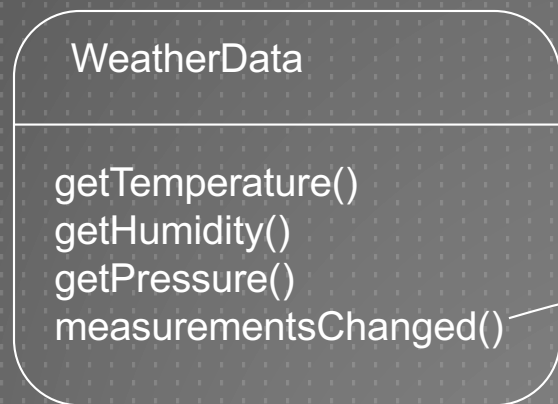
- ▶ I expect to have multiple different behavior implementations of strategy (e.g. flying behavior)
- ▶ I would like to be able to assign different fly behavior to different ducks at both static and runtime easily.

THE OBSERVER PATTERN

WEATHER MONITORING APPLICATION




WHAT NEEDS TO BE DONE?



```
/*
 * Call this method
 * whenever measurements are
 * Updated
 */
Public void measurementsChanged(){
    // your code goes here
}
```

Update three
different displays



An arrow points from the `measurementsChanged()` method in the `WeatherData` class to the code block. Another arrow points from the code block to the text 'Update three different displays'.

PROBLEM SPECIFICATION

- ▶ weatherData class has three getter methods
- ▶ measurementsChanged() method called whenever there is a change
- ▶ Three display methods needs to be supported: current conditions, weather statistics and simple forecast
- ▶ System should be expandable

FIRST CUT AT IMPLEMENTATION

```
public class WeatherData {  
  
    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 methods  
}
```

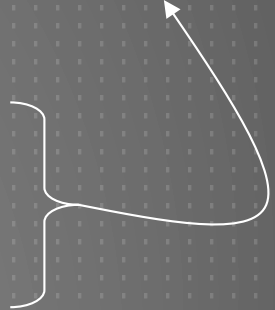
THINK-PAIR-SHARE

- How should the code be updated if we want to support another type of displays beyond the three?

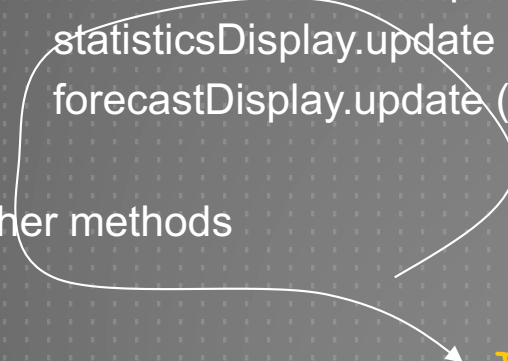
```
public class WeatherData {  
  
    public void measurementsChanged(){  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        currentConditionsDisplay.update (temp, humidity, pressure);  
        statisticsDisplay.update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

FIRST CUT AT IMPLEMENTATION

```
public class WeatherData {  
  
    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 methods  
}
```



Area of change which can be managed better by encapsulation



By coding to concrete implementations there is no way to add additional display elements without making code change

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.

BASIS FOR OBSERVER PATTERN

- ▶ Fashioned after the publish/subscribe model
- ▶ Works off similar to any subscription model
 - ▶ Buying newspaper
 - ▶ Magazines
 - ▶ List servers

WEATHER DATA INTERFACES

```
public interface Subject {  
    public void registerObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
}  
public interface Observer {  
    public void update(float temp, float humidity, float pressure);  
}  
public interface DisplayElement {  
    public void display();  
}
```


IMPLEMENTING SUBJECT INTERFACE

```
public class WeatherData implements Subject {  
    private ArrayList observers;  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherData() {  
        observers = new ArrayList();  
    }  
}
```

REGISTER AND UNREGISTER

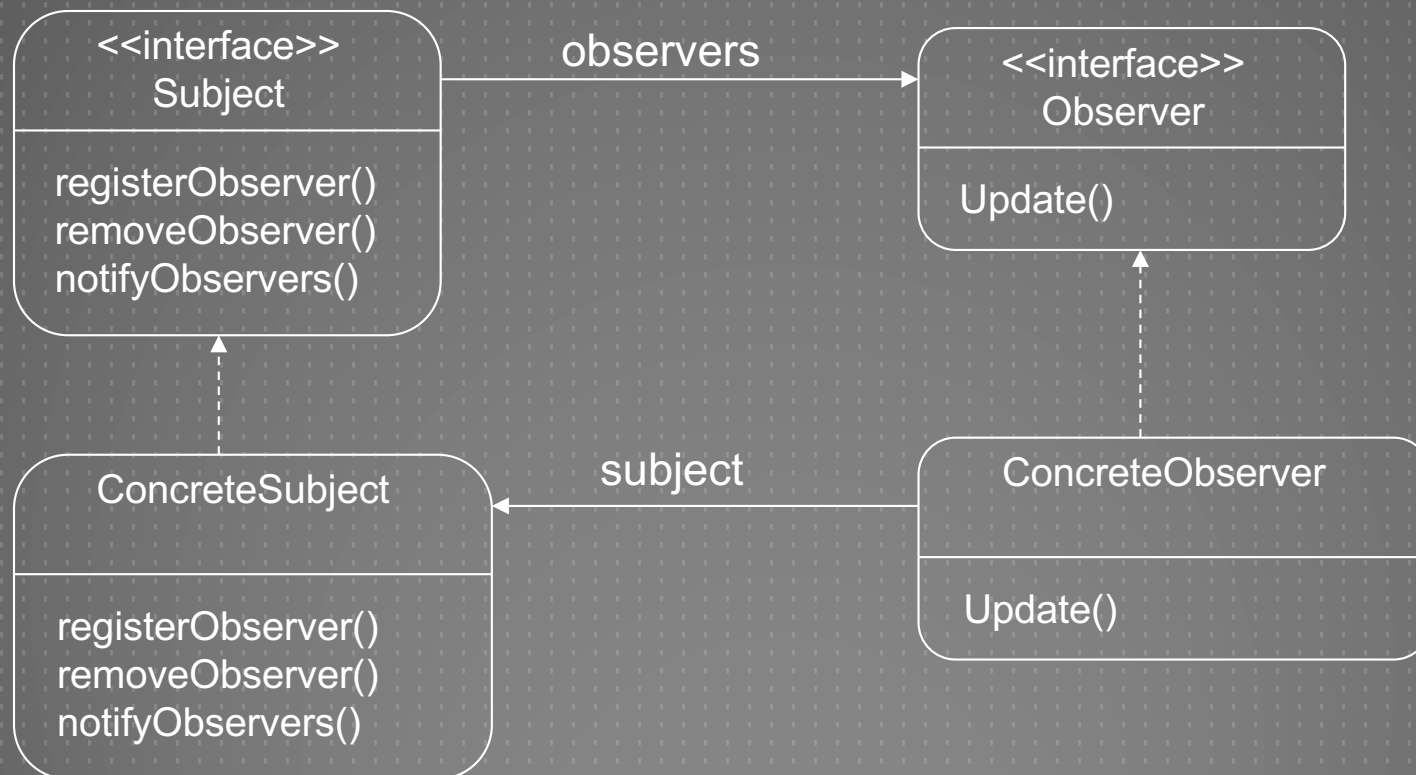
```
public void registerObserver(Observer o) {  
    observers.add(o);  
}
```

```
public void removeObserver(Observer o) {  
    int i = observers.indexOf(o);  
    if (i >= 0) {  
        observers.remove(i);  
    }  
}
```

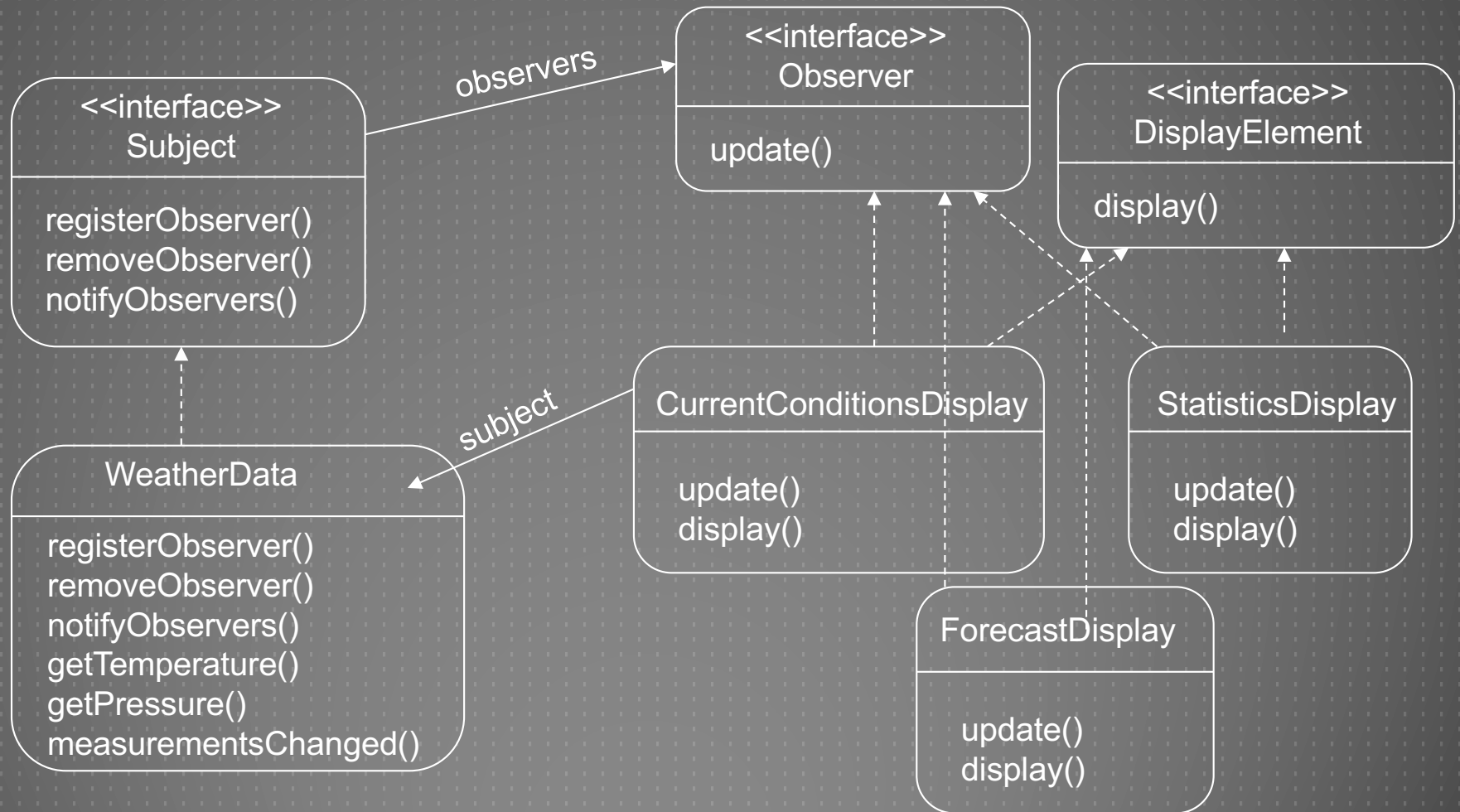
NOTIFY METHODS

```
public void notifyObservers() {  
    for (int i = 0; i < observers.size(); i++) {  
        Observer observer = (Observer)observers.get(i);  
        observer.update(temperature, humidity, pressure);  
    }  
}  
  
public void measurementsChanged() {  
    notifyObservers();  
}
```

OBSERVER PATTERN – CLASS DIAGRAM



OBSERVER PATTERN – WEATHER DATA



OBSERVER PATTERN – POWER OF LOOSE COUPLING

- ▶ The only thing that the subject knows about an observer is that it implements an interface
- ▶ Observers can be added at any time and subject need not be modified to add observers
- ▶ Subjects and observers can be reused or modified without impacting the other [as long as they honor the interface commitments, i.e. the input arguments (temperature, pressure, humidity) of the update method]

THINK-PAIR-SHARE

- ▶ In the observer pattern, which of the following types of changes are easily accommodated?
 - ▶ (A) adding a new type of observer
 - ▶ (B) adding a new type of subject that requires different subject states (e.g. sensor data kinds)
 - ▶ (C) selectively notifying different observers

RECAP

- ▶ Assignment 1 is due at 11:59 PM.
- ▶ “Command”
- ▶ UML => Information Hiding => Strategy / Observer

► Strategy

- Intent: you have lots of classes / subclasses that you want to implement (e.g., MallardDuck/RedheadedDuck), but some behavior implementations are necessary, and some are optional, and some are not appropriate.
- Alternative 1: Subclassing, and override “fly” to be an empty implementation => confusing to client applications (e.g., RubberDuck d = ... , d.fly();)
- Alternative 2: Use interface Flyable, this leads to MallardDuck and RedheadedDuck duplicate the fly() method implementation. MallardDuck cannot change its flybehavior at runtime. Cannot change to RocketFueledFlyBehavior at runtime.

► Observer

- 1 to n relationships
- Publish and subscribe
- Notification / listener-based implementation (e.g. Java Swing GUI)
- Subject: an abstraction of state that can be updated.
ConcreteSubject: WeatherData is a concrete subject.
- ConcreteObserver: e.g. StatisticsDisplay
- Q: What happens inside “notify” method in Subject?

- ▶ What is easy to change when using Observer pattern?

- ▶ # of observers, adding a new observer to a subject is easy

- ▶ What is difficult to do when using Observer pattern?

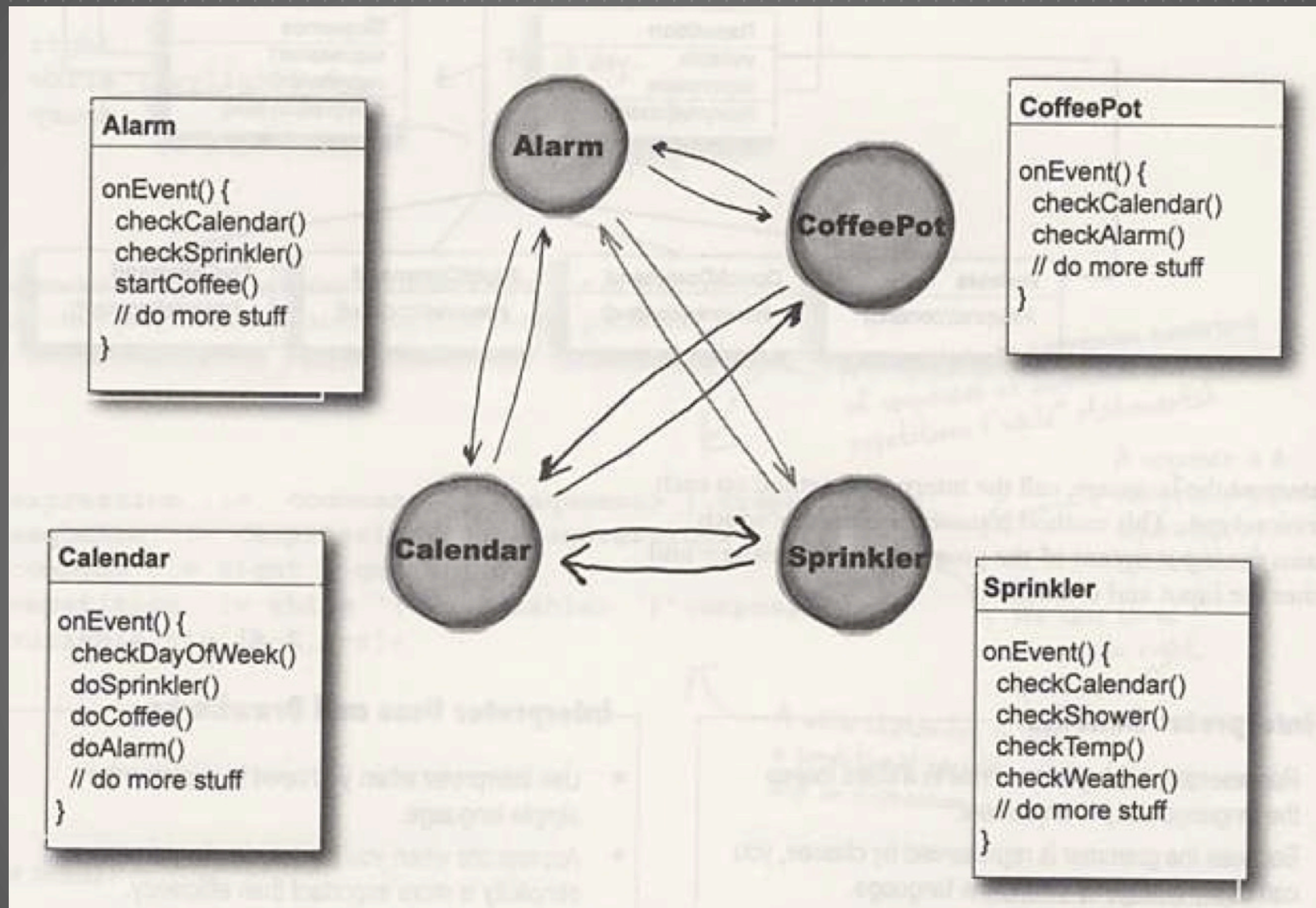
- ▶ Changing the internal states of Subject and modifying the interface of “update” is extremely hard.

MEDIATOR

MEDIATOR

- ▶ To centralize complex communications and control between related objects
- ▶ Scenario:
 - ▶ Bob had a Java enabled auto-house.
 - ▶ All of his appliances are designed to make his life easier
 - ▶ When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing.
 - ▶ No coffee on the weekends
 - ▶ Turn off the sprinkler 15 minutes before a shower is scheduled.

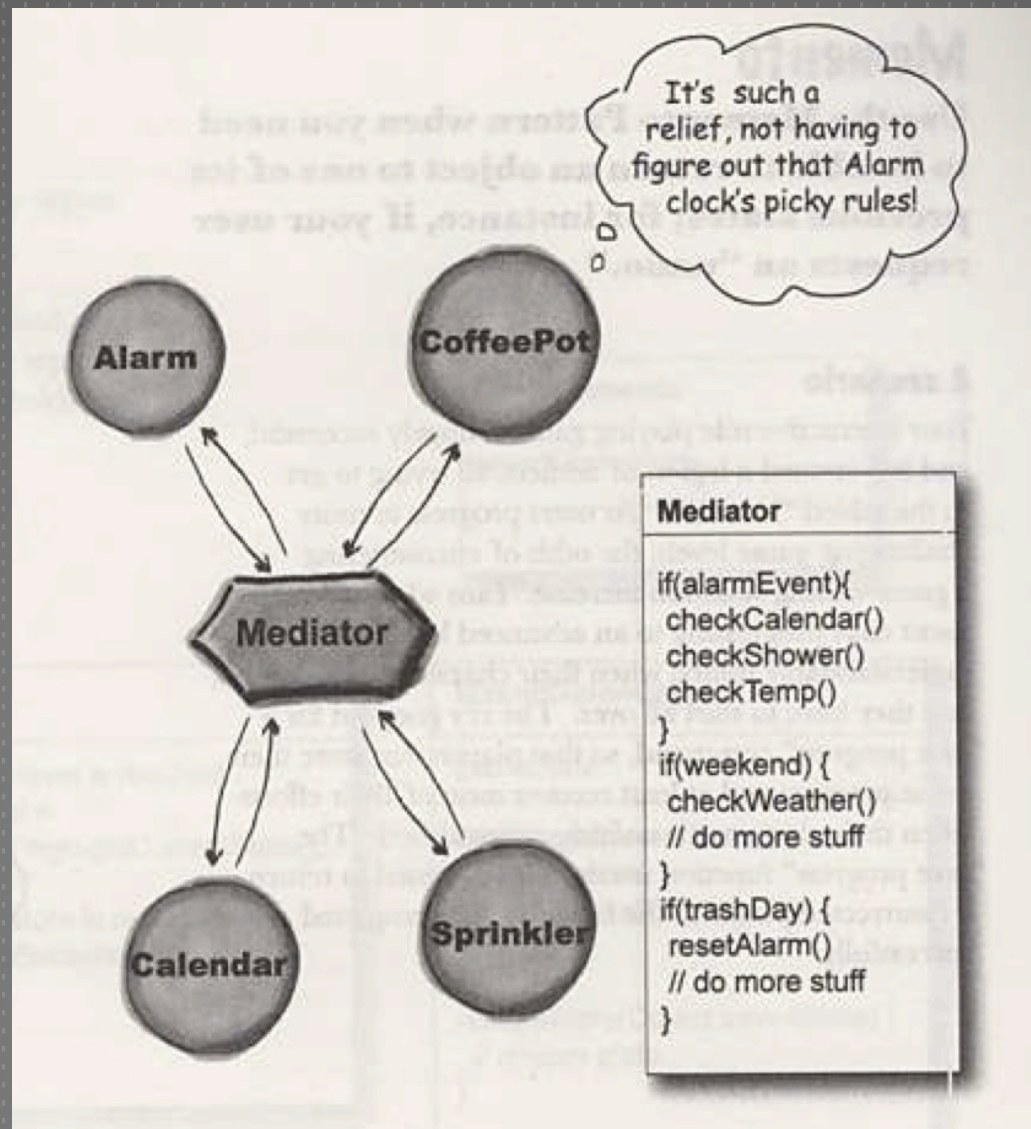
COMPLEX INTER-COMMUNICATION



MEDIATOR IN ACTION

- ▶ With a mediator added to the system all appliance objects can be greatly simplified
 - ▶ They tell the Mediator when their state changes
 - ▶ They respond to requests from the Mediator
- ▶ Before adding the mediators all objects need to know about each other

MEDIATOR IN ACTION



MEDIATOR IN ACTION

- ▶ With the mediator in place, all objects are all completely decoupled from each other.
- ▶ Mediator contains all control logic for the entire system.
- ▶ When an existing appliance needs a new rule or a new appliance is added to the system, you will know that all logic will be added to the mediator

MEDIATOR BENEFITS

- ▶ Increase the reusability of the objects supported by the Mediator by decoupling them from the system
- ▶ Simplifies maintenance of the system by centralizing control logic
- ▶ Simplified and reduces the variety of messages sent between objects in the system

MEDIATOR DRAWBACKS

- ▶ Without proper design, the mediator object itself can become overly complex.
- ▶ Mediator also exposes a single point of vulnerability and complexity.

RECAP

- ▶ The Strategy Pattern defines a family of algorithms, and it lets the algorithm vary independently from clients that use it. The choice of the algorithm can be replaced at runtime.
- ▶ 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.
- ▶ The Mediator Pattern encapsulates many to many dependencies between objects.

QUESTIONS?