Software Design and Architecture

Foundation of Design Patterns

Design principles

High-level principles

- Single Responsibility
- Open/Closed
- Liskov Substitution Principle
- Interface Segregation
- Dependency Inversion

Low-level principles that can help guide design decisions.

- Encapsulate what varies
- Program to interfaces, not implementations
- Favor composition over inheritance
- Strive for loose coupling

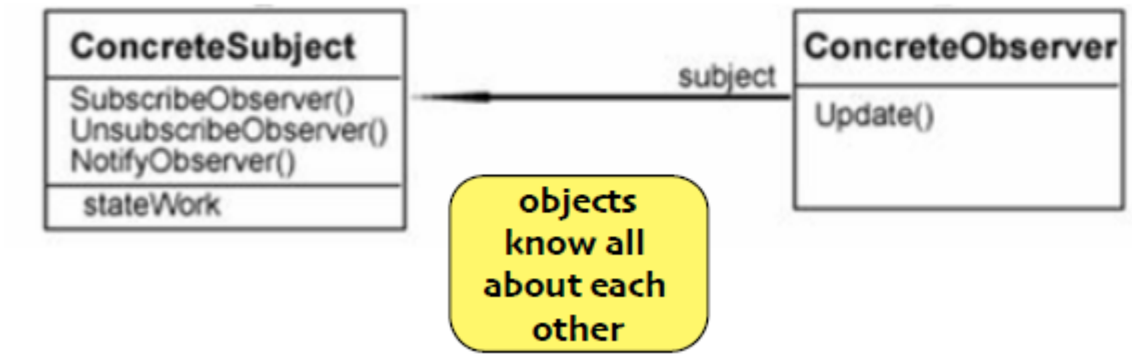
Problem

Related objects need to communicate with each other to maintain consistency.

Solution

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.

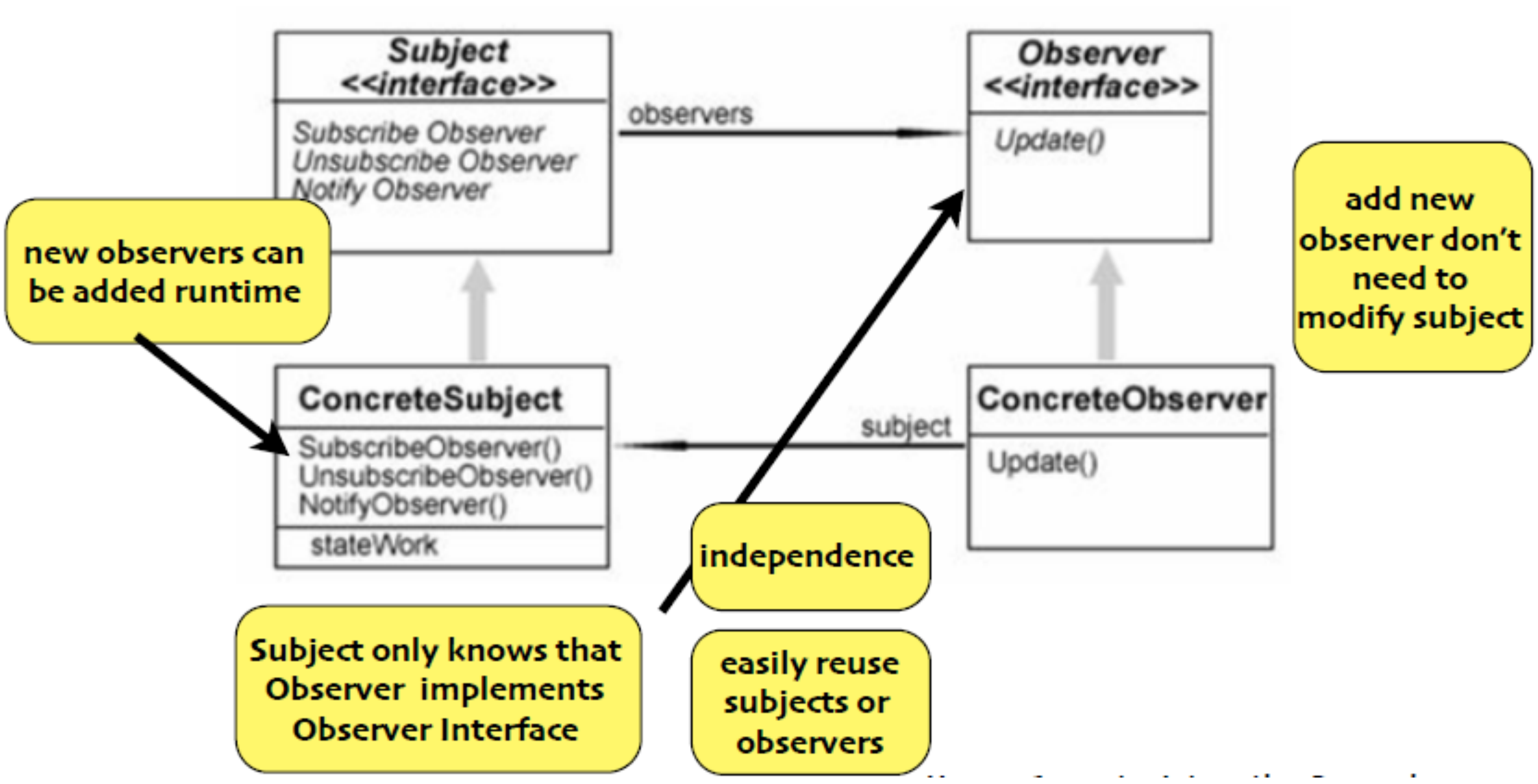
How it was


ConcreteSubject
SubscribeObserver()
UnsubscribeObserver()
NotifyObserver()
stateWork
subject
ConcreteObserver
Update()
objects know all about each other

Design Principle

Strive for Loosely coupled designs between objects that interact

Design

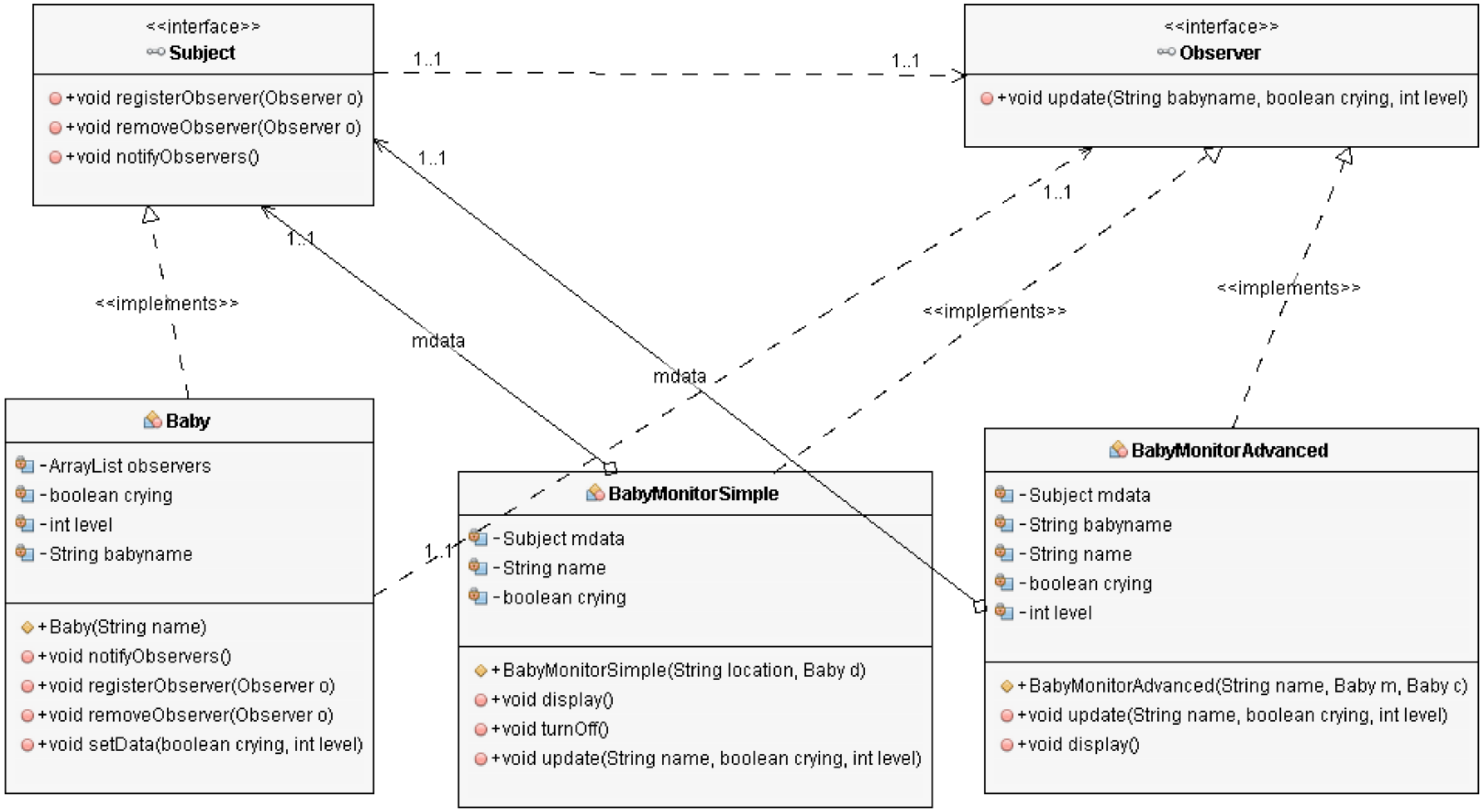

Subject
<<interface>>
Subscribe Observer
Unsubscribe Observer
Notify Observer
observers
Observer
<<interface>>
Update()
ConcreteSubject
SubscribeObserver()
UnsubscribeObserver()
NotifyObserver()
stateWork
subject
ConcreteObserver
Update()
new observers can be added runtime
add new observer don't need to modify subject
independence
Subject only knows that Observer implements Observer Interface
easily reuse subjects or observers

Baby Monitor

Suppose we implement a simple Baby Monitoring System using the Observer pattern with 1 baby and 2 different types of Monitors (simple, advanced).

- Babies can cry at 3 different levels (sobbing, crying, screaming).
- The simple monitor can only notify its user that the baby is crying.
- The advanced monitor also indicates the level of crying.

Baby Monitor



Interface

```
public interface Subject {
 public void registerObserver(Observer o);
 public void removeObserver(Observer o);
 public void notifyObservers();
}


 public interface Observer {
		public void update(String babyname, boolean crying, int level);
 }
```

Subject

```
public class Baby implements Subject {
        private ArrayList observers;
        private boolean crying=false;
        private int level=0;
        private String babyname;
        public Baby(String name){
                this.babyname=name;
                observers=new ArrayList();
        }
        public void notifyObservers() {
                for (int i=0; i< observers.size(); i++) {
                        Observer observer = (Observer) observers.get(i);
                        observer.update(babyname, crying, level);
                }
        }
        public void registerObserver(Observer o) {
                observers.add(o);

        }
        public void removeObserver(Observer o) {
                int i = observers.indexOf(o);
                if (i >=0) {
                        observers.remove(i);
                }
        }
        public void setData(boolean crying, int level) {
                this.crying=crying;
                this.level=level;
                notifyObservers();
        }
}
```

Simple Monitor

```
public class BabyMonitorSimple implements Observer {

        private Subject mdata;
        private String name;
        private boolean crying;

        public BabyMonitorSimple(String location, Baby d) {
                this.mdata=d;
                this.name=location;
                mdata.registerObserver(this);
        }

        public void display() {
                if (crying) {
                        System.out.println("Monitor:" + name + " baby is crying");
                }
        }

        public void turnOff() {
                mdata.removeObserver(this);
        }

        public void update(String name, boolean crying, int level) {
                this.crying=crying;
                display();
        }
}
```

Advanced Monitor

```
public class BabyMonitorAdvanced implements Observer {
            private Subject mdata, cdata;
            private String babyname;
            private String name;
            private boolean crying;
            private int level;

            public BabyMonitorAdvanced(String name, Baby m, Baby c) {
                        this.name=name; this.mdata=m; this.cdata=c;
                        mdata.registerObserver(this);cdata.registerObserver(this);
            }

            public void update(String name, boolean crying, int level) {
                        this.babyname=name;
                        this.crying=crying;
                        this.level=level;
                        display();
            }

            public void display() {
                        if (crying) {
                                    System.out.println("Monitor:"+ name + " baby: " + babyname + " is crying at level: " +
level);
                        }
            }
}
```

Test

```
class TestBabyMonitor {
		public static void main(String[] args) {
				Baby marla = new Baby("marla");
				// one monitor with one baby
				BabyMonitorSimple livingRoom = new BabyMonitorSimple("kitchen ", marla);
				marla.setData(true, 1);
				// one monitor listening to two babies
				Baby charlie = new Baby("Charlie");
				BabyMonitorAdvanced kitchen = new BabyMonitorAdvanced("Living room ", marla, charlie);
				marla.setData(true, 2);
				charlie.setData(true,1);
		}
}
```

Output

```
Monitor:kitchen  baby is crying
Monitor:kitchen  baby is crying
Monitor:Living room  baby: marla is crying at level: 2
Monitor:Living room  baby: Charlie is crying at level: 1
```

Push vs Pull

- All the data gets pushed all the Time

+ every observer has all the latest data

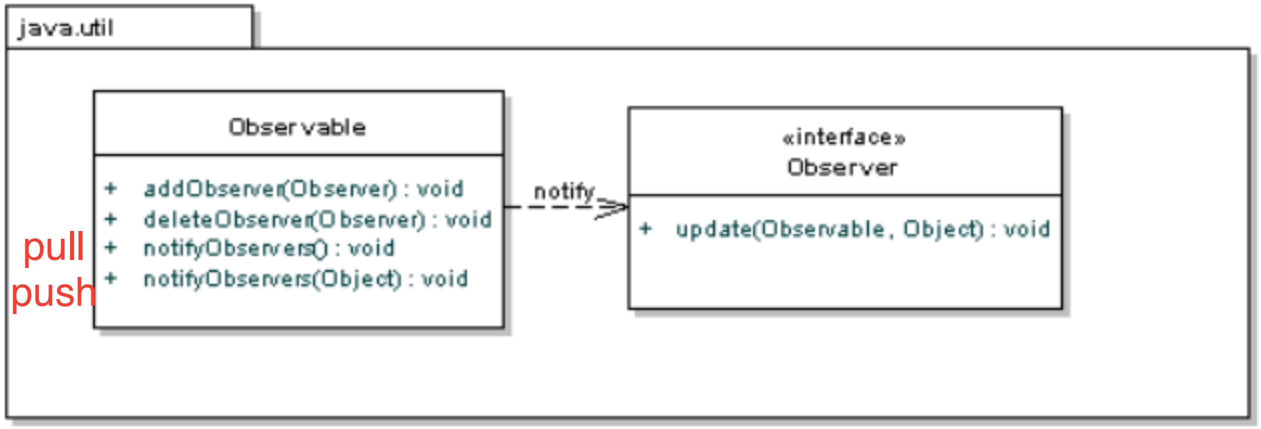
- May need multiple calls to get all the data

+ more flexibility let observers figure it out themselves

Java Observable

this is an abstract class

same


java.util
Observable
+ addObserver(Observer) : void
+ deleteObserver(Observer) : void
+ notifyObservers() : void
+ notifyObservers(Object) : void
pull
push
notify
«interface»
Observer
+ update(Observable, Object) : void

Java does all the work with notifying observers

Observable == Subject

Steps

1. To become observer
    1. Objects needs to implement observer interface
    2. call addObserver()
2. For Observable (subject) to send notifications
    3. Extend java.util.Observable
        1. call setChanged() flag to give you more flexibility
        2. notifyObservers() pull
           or notifyObservers(Object arg) push

Steps

- For observer to receive notifications
  - update method looks different
    - update(Observable o, Object arg)

instead of pushing all data, now just receive a reference to the updated object and use getters to PULL your information

Alternatively use arg to PUSH all your data

```
public void update(Observable o, Object arg)
        if (o instanceof SubjectClass) {
              SubjectClass data = (SubjectClass) o;
                this.property=  data.getProperty()
                         display();
          }
```

```
public class SubjectClass extends Observable {
  private Type property;

  public Type getProperty() {
                return property
          }
```

Limitations

Observable is class not interface

- Can’t use multiple inheritance

Setchanged() is protected

- you must subclass to be able to call it
- violates favor composition over inheritance

Order in which Observers are updated may differ from your own implementation

Delegation

When designing a class, there are four ways to handle an incoming message

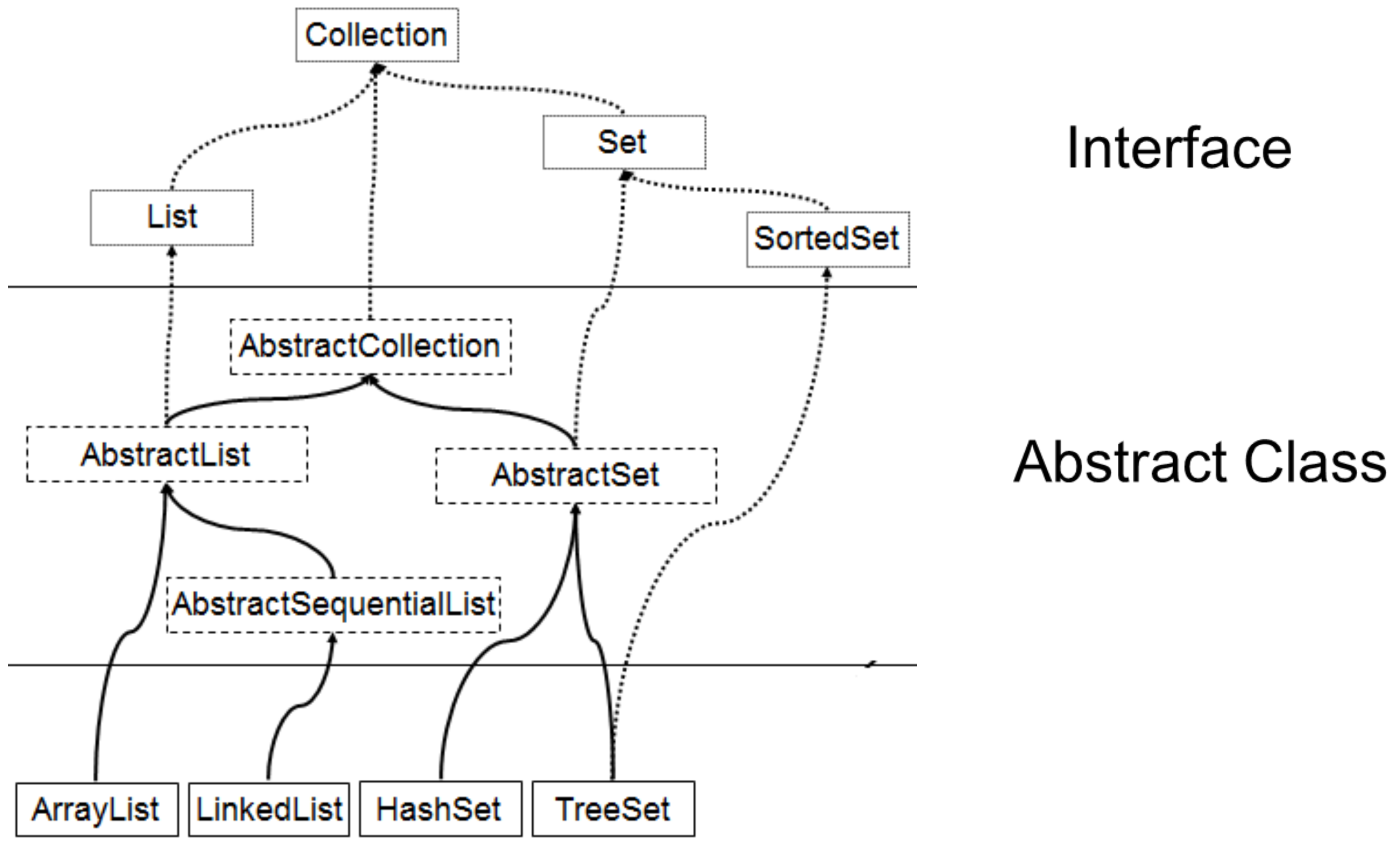
- Handle message by **implementing code in a method**
- Let the **class's superclass** handle the request **via inheritance**
- **Pass the request to another object** (**delegation**)
- some combination of the previous three

Delegation

Delegation is employed when **some other class already exists to handle a request** that might be made on the class being designed

- The host class **simply creates a private instance of the helper class** and **sends messages to it when appropriate**
- As such, delegation is often referred to as a “**HAS-A**” relationship

Java Collection

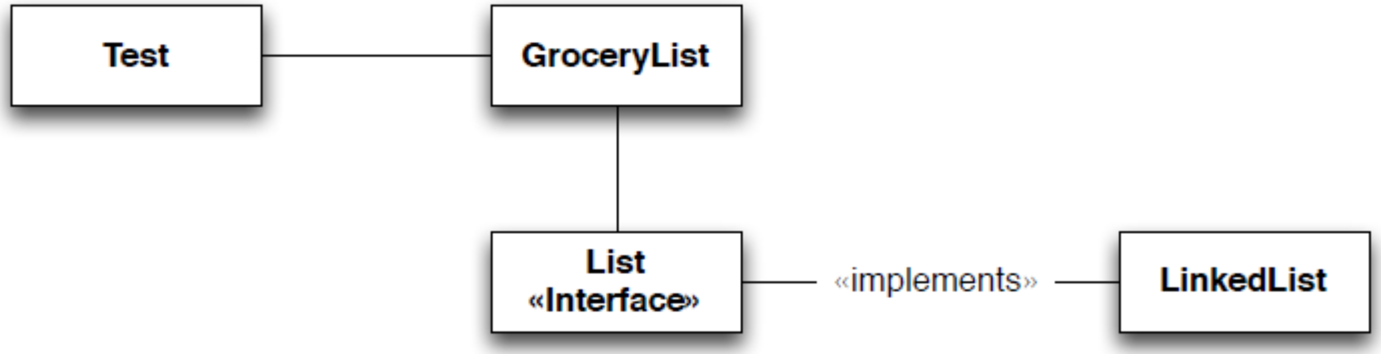


```
import java.util.List;
import java.util.LinkedList;
public class GroceryList {
  private List<String> items;
  public GroceryList() {
    items = new LinkedList<String>();
  }
  public void addItem(String item) {
    items.add(item);
  }
  public void removeItem(String item) {
    items.remove(item);
  }
  public String toString() {
    String result = "Grocery List\n------------\n\n";
    int index = 1;
    for (String item: items) {
       result += String.format("%3d. %s", index++, item) + "\n";
    }
    return result;
  }
}
```

GroceryList delegates all of its work to Java's LinkedList class (which it accesses via the List interface).

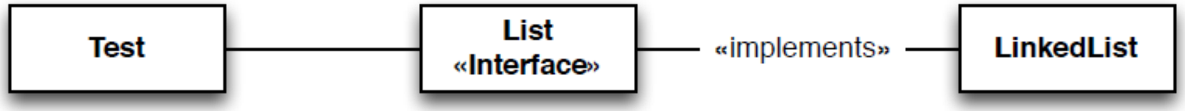
```
public class Test {
   public static void main(String[] args) {
      GroceryList g = new GroceryList();
      g.addItem("Granola");
      g.addItem("Milk");
      g.addItem("Eggs");
      System.out.println("" + g);
      g.removeItem("Milk");
      System.out.println("" + g);
   }
}
```

With the delegation, We get a nice abstraction in our client code. We can create grocery lists, add and remove items and get a printout of the current state of the list.



GroceryList needs "list like" functionality. So, internally, it uses a LinkedList (via a List interface). This is hidden from Test which just sees a "grocery list" with a nice abstraction.

```
import java.util.List;
import java.util.LinkedList;
public class TestWithout {
   public static void printList(List<String> items) {
      System.out.println("Grocery List");
      System.out.println("------------\n");
      int index = 1;
      for (String item : items) {
         System.out.println(String.format("%3d. %s", index++, item));
      }
      System.out.println();
   }
   public static void main(String[] args) {
      List<String> g = new LinkedList<String>();
      g.add("Granola");
      g.add("Milk");
      g.add("Eggs");
      printList(g);
      g.remove("Milk");
      printList(g);
   }
}
```



Without delegation, we get less abstraction. We're using the List interface directly with its method names and We have to create a static method to handle the printing of the list rather than using toString().

Delegation

Now, the two programs (with delegation and without delegation) produce exactly the same output.

Benefits of Delegation

- **Better abstraction**
- **Less code** in classes we write ourselves
- We can **change delegation relationships at runtime**!
    - Unlike inheritance relationships

Delegation

Changing **delegation relationships** at run-time

- A class can use a set at run-time

  ```
  Set<String> items = new HashSet<String>();
  ```

- If the class suddenly needs to be sorted, it can do this

  ```
  items = new TreeSet<String>(items);
  ```

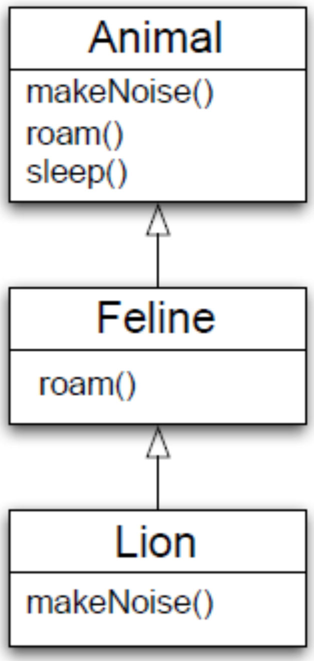
- We have changed the delegation to an entirely new object at run-time and now the items are sorted
  - In both cases, the type of items is Set<String> and we get the correct behavior via polymorphism

Polymorphism

a.roam() invokes Feline.roam()
a.makeNoise() invokes Lion.makeNoise()

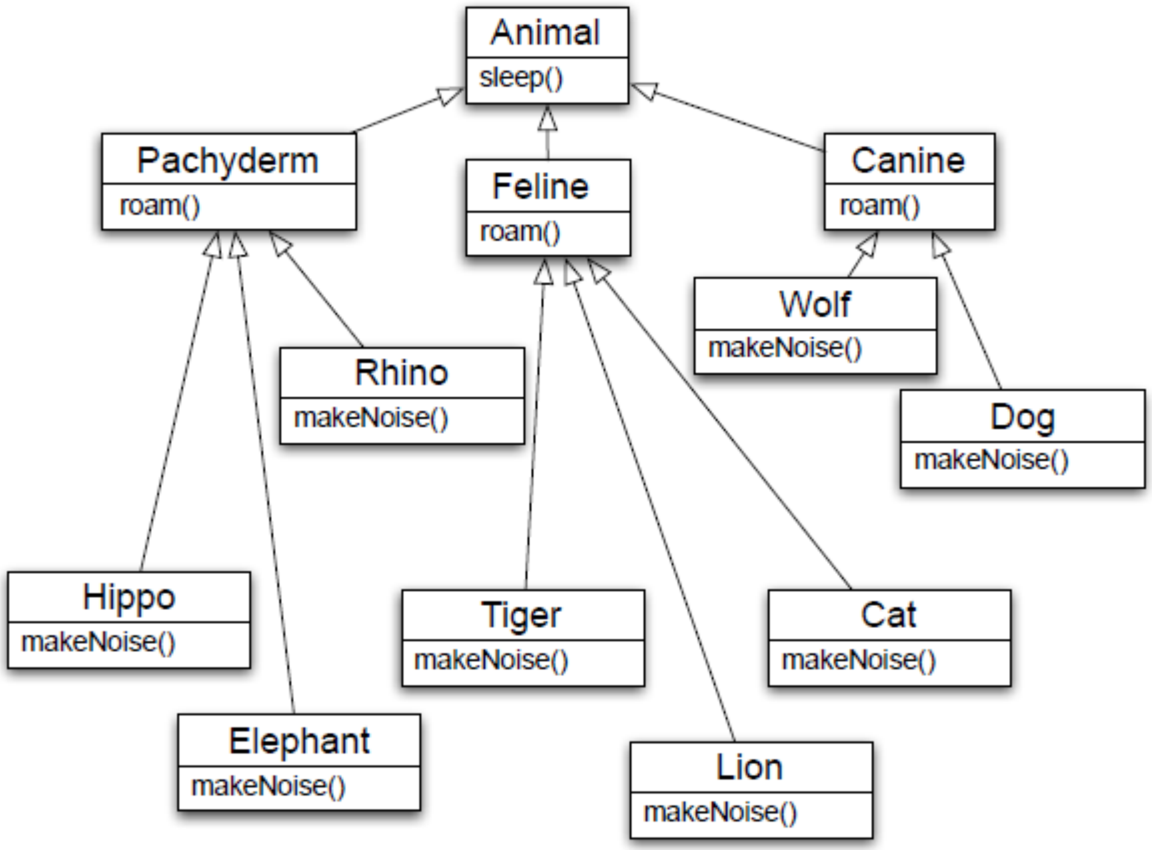
A message sent to Animal travels down the hierarchy looking for the "most specific" method body

- In actuality, method lookup starts with Lion and goes up



```
Animal a = new Lion()
a.makeNoise();
a.roam();
a.sleep();
```

Polymorphism



Polymorphism

Polymorphism allows us to write very abstract code that is robust with respect to the creation of new subclasses

For instance

```
public void goToSleep(Animal[] zoo) {
    for (int i = 0; i < zoo.length; i++) {
        zoo[i].sleep();
}}
```

We don’t have to care what type of animals are contained in the array

We just call sleep() and get the correct behavior for each type of animal

Delegation

Summary

- **Don’t re-invent the wheel… delegate!**
- Delegation is **dynamic** (not static)
- delegation relationships can **change at run-time**
- **Not tied to inheritance**

Delegation

Delegation, **as a design pattern**, is used throughout the iOS and Cocoa frameworks

- Basic pattern involving two objects
- Host and delegate; use delegate to customize host
- Define an interface that a delegate will implement
- some methods are required; the rest are optional
- Host will invoke methods on delegate as needed to influence its behavior

Beautify the following poorly designed Java program.

```
public class Bob {

    public static void main(String[] args) {
        Bob b = new Bob();
        System.out.println(b.mood(1));
        System.out.println(b.mood(2));
        System.out.println(b.mood(3));
    }

    String mood(int mymood) {
        switch (mymood) {
            case 1:
                return "Grumpy";
            case 2:
                return "OK";
            case 3:
                return "Happy";
            default:
                throw new RuntimeException("unknown mood");
        }
    }
}
```

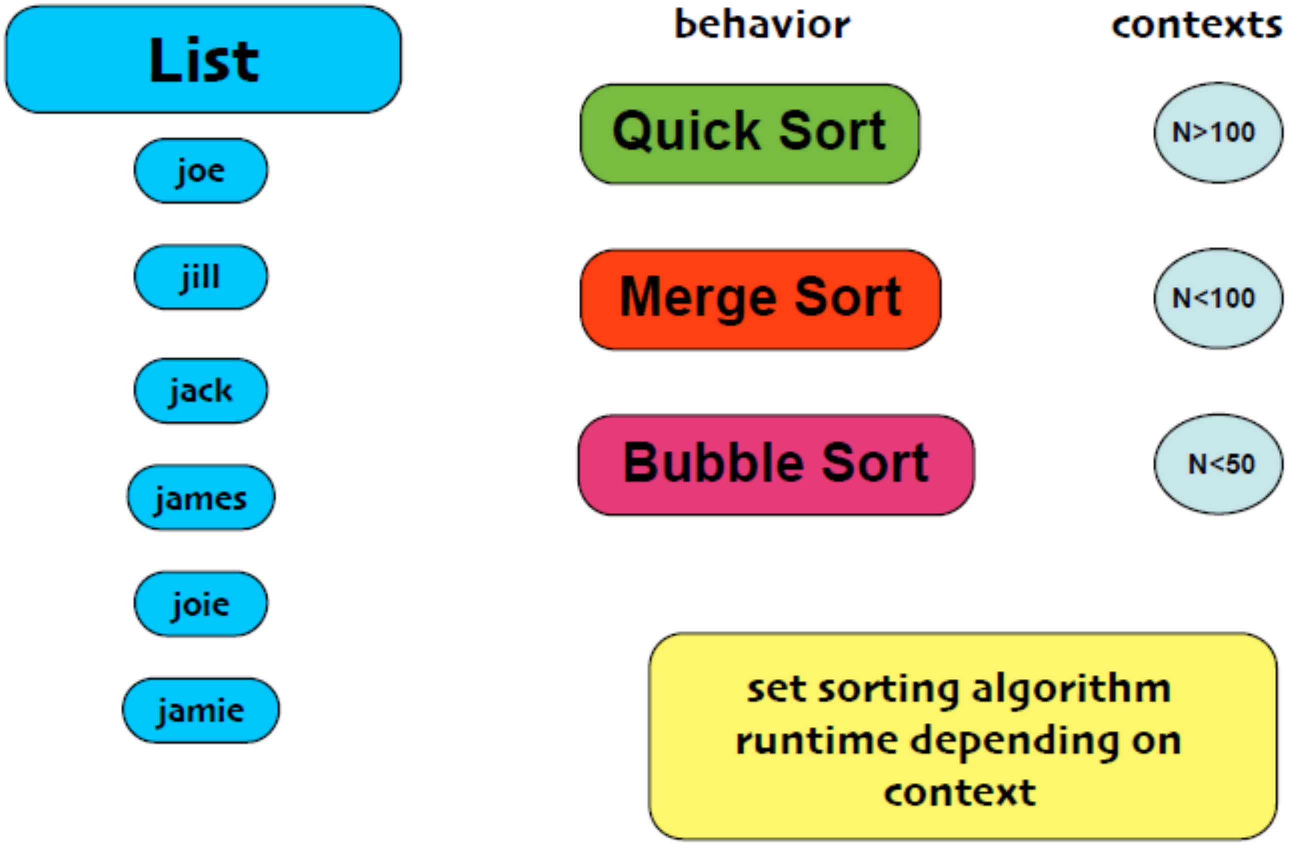
Problem

Your program needs to support different kinds of behavior

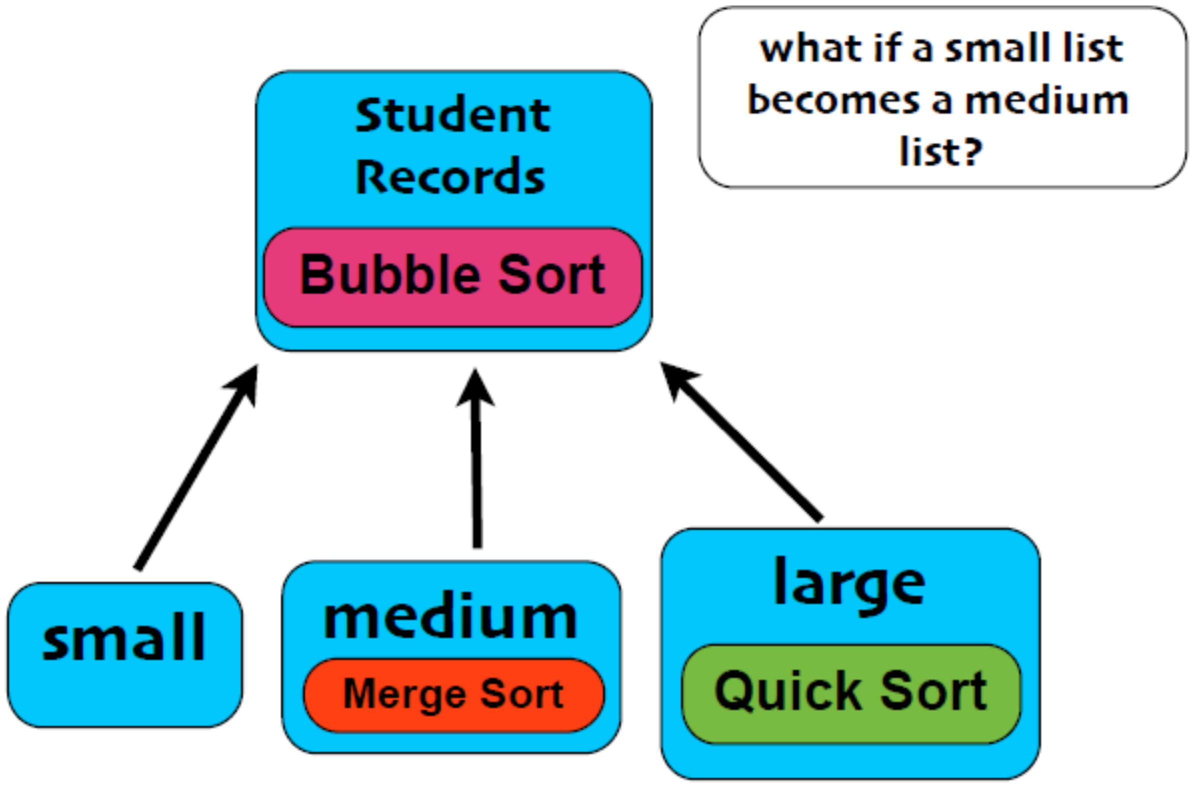
Solution

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

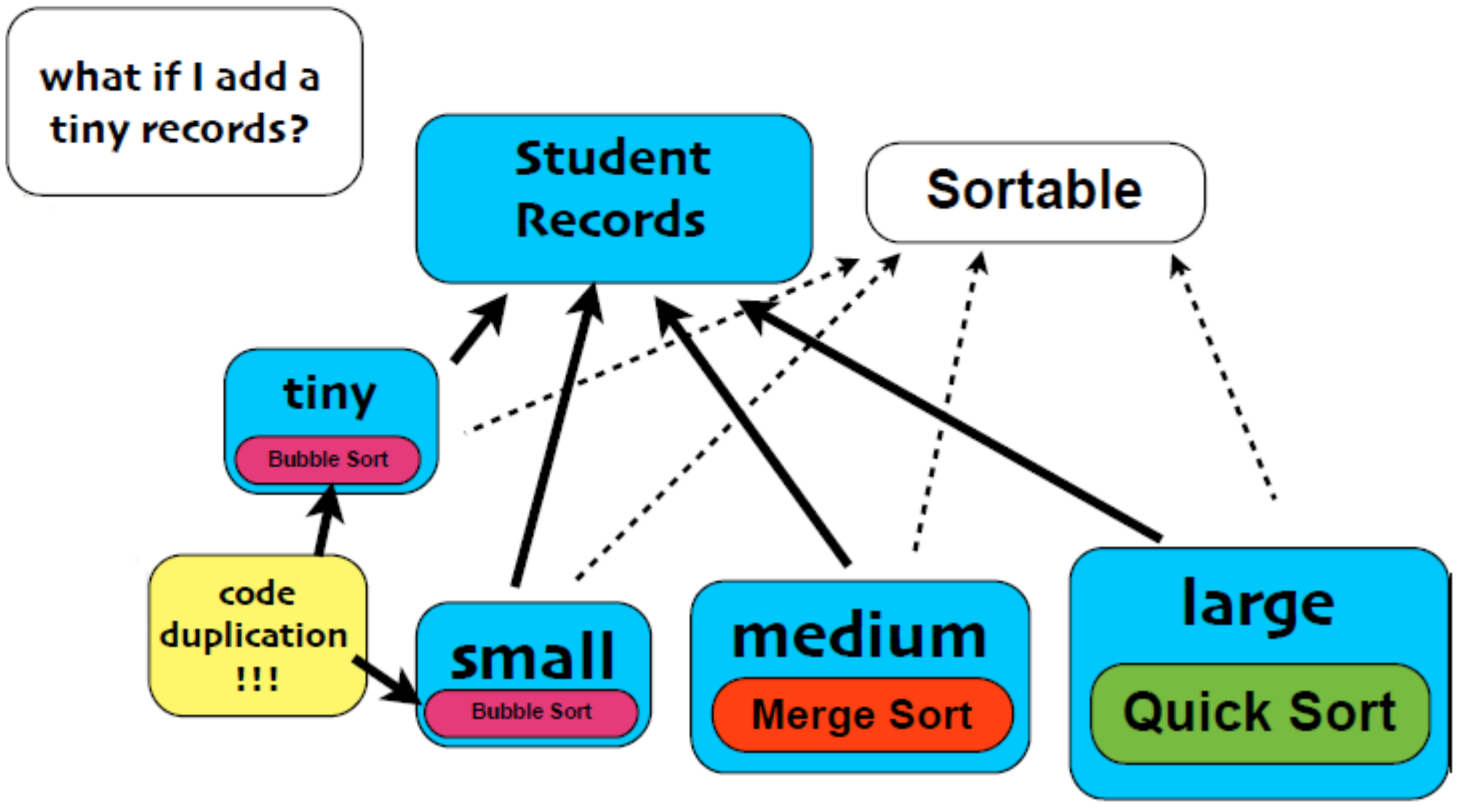
Example


List
joe
jill
jack
james
joie
jamie
behavior
contexts
Quick Sort
N>100
Merge Sort
N<100
Bubble Sort
N<50
set sorting algorithm runtime depending on context

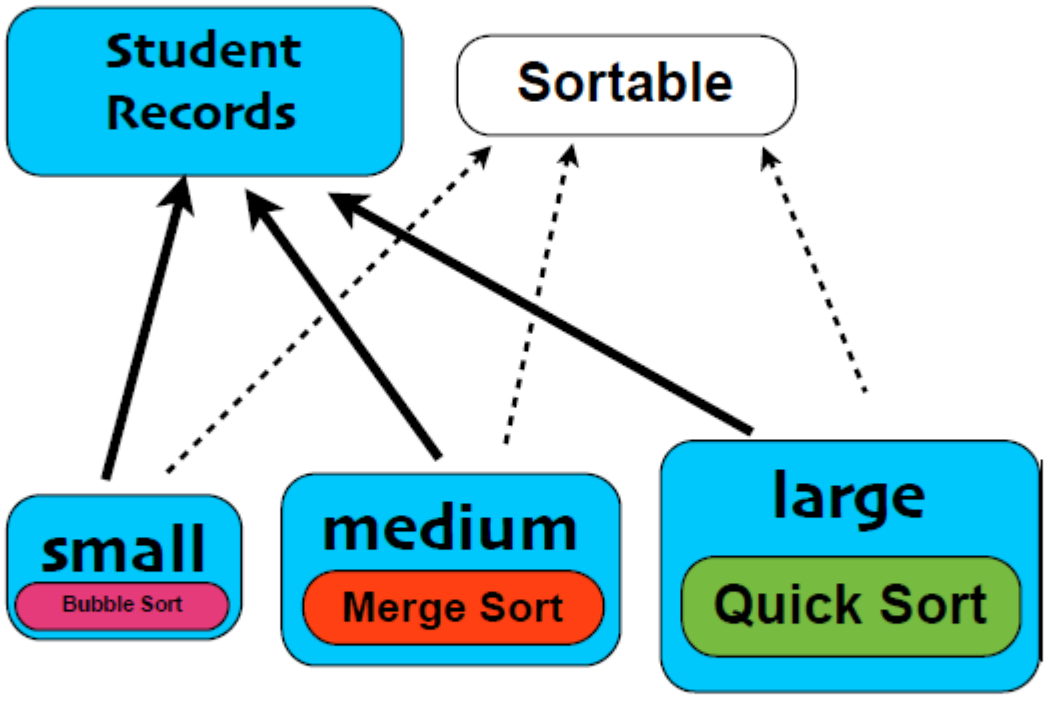
Naive approach



Add Interface?



Add Interface



what if I change the student's records list to a hashtable?

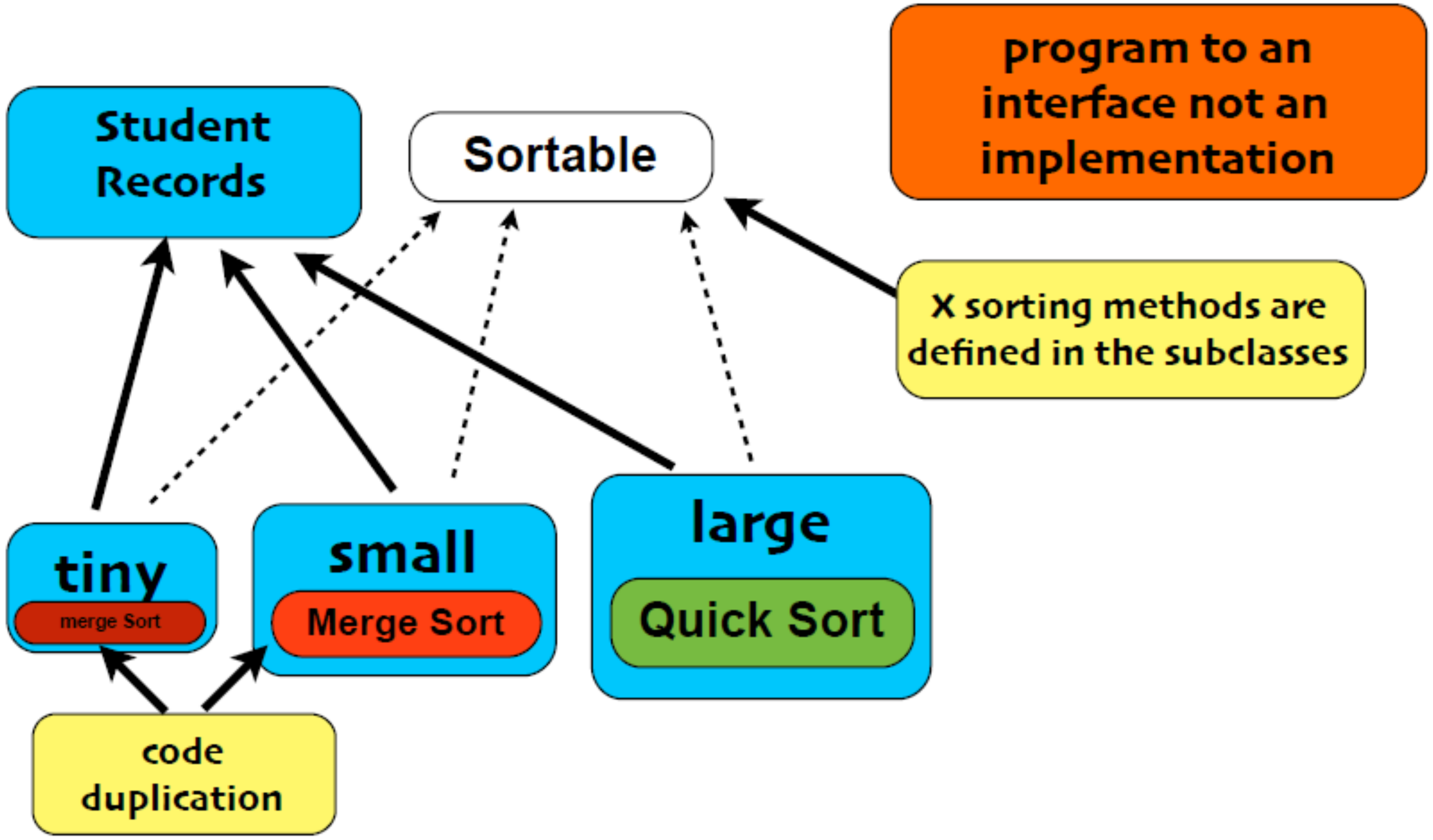
maintenance nightmare!!

Design Principles

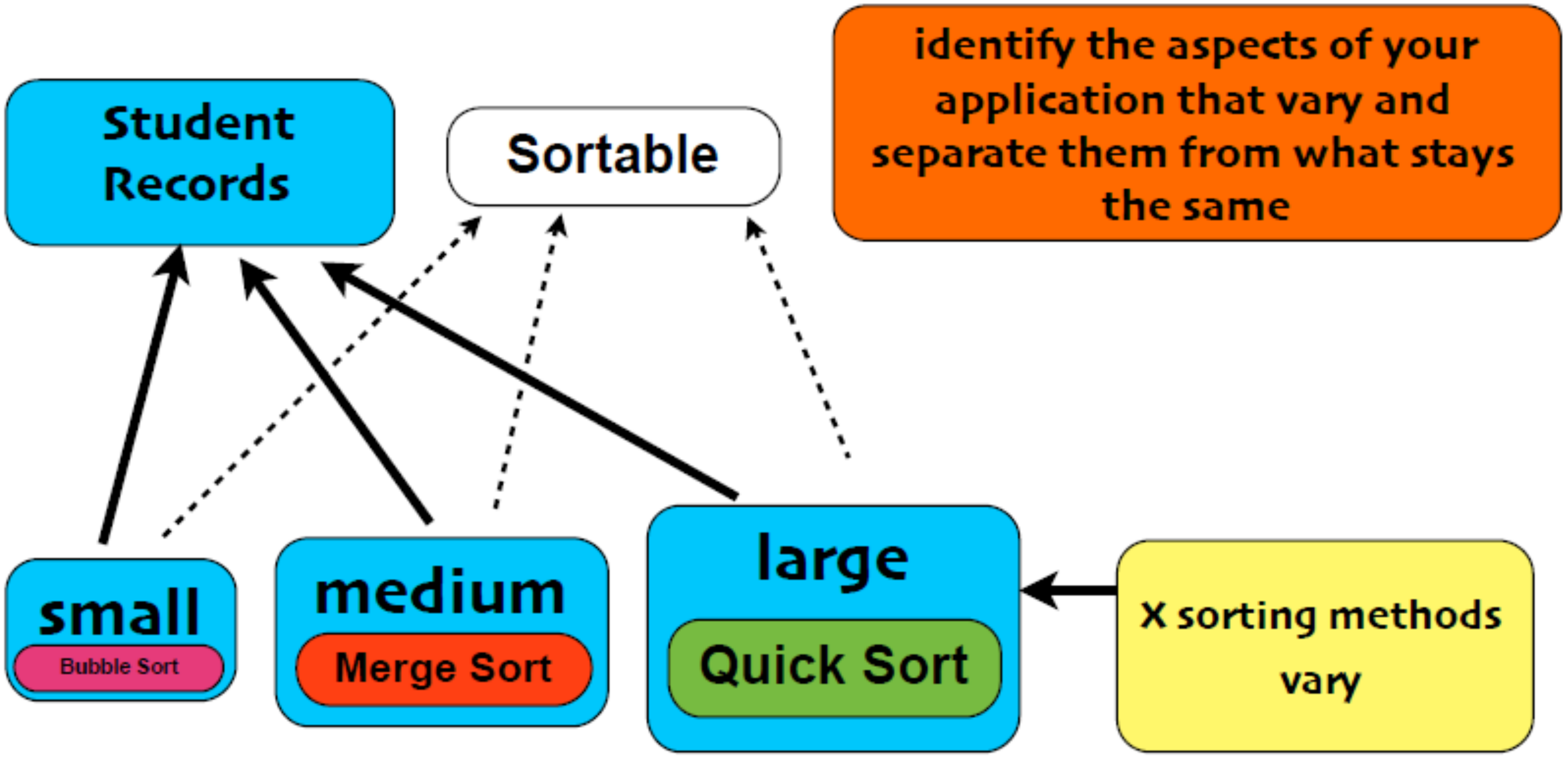
Encapsulate what varies

Program to interfaces, not implementations

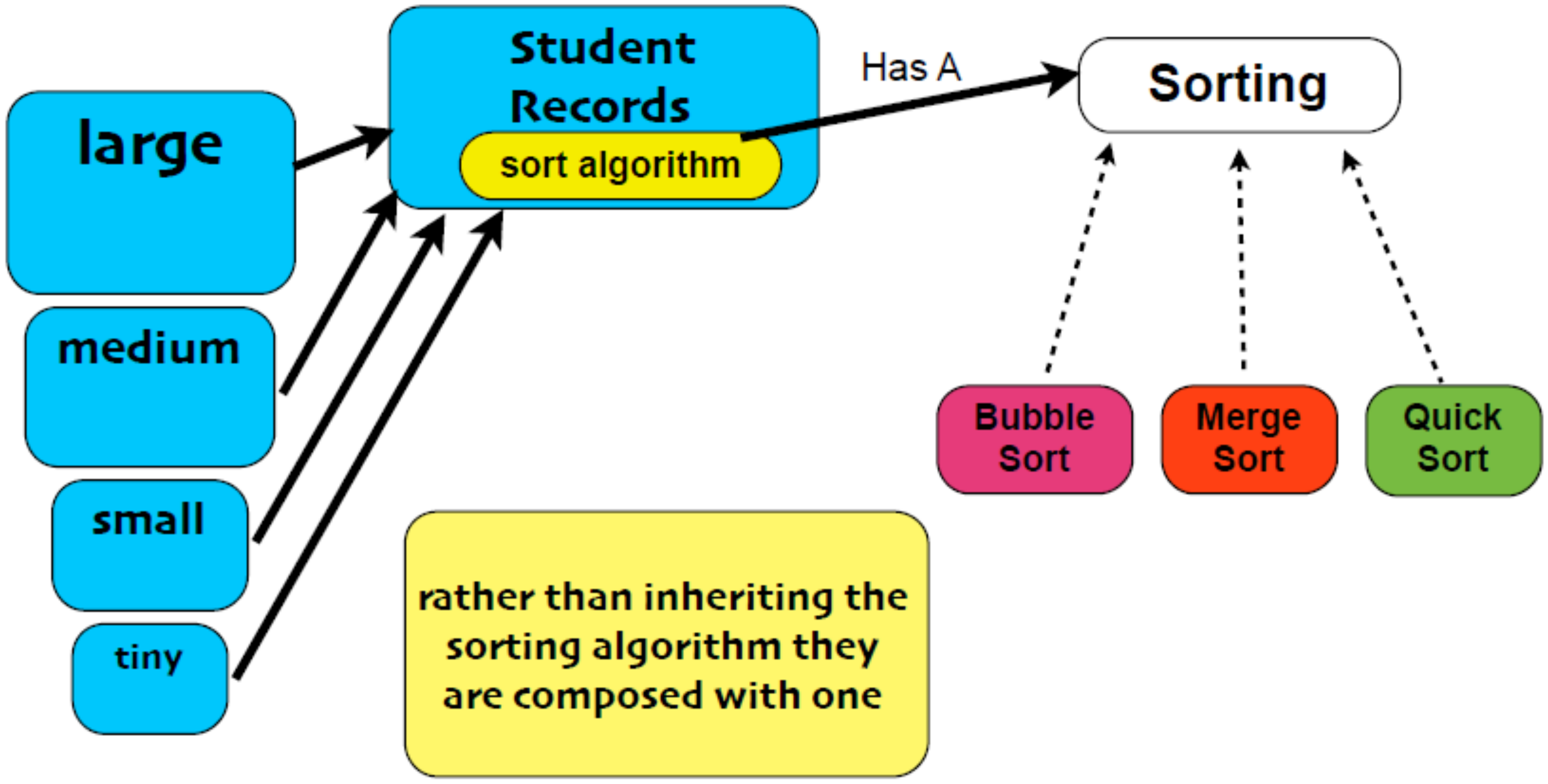
Program to Interface


program to an interface not an implementation
Student Records
Sortable
X sorting methods are defined in the subclasses
tiny
merge Sort
small
Merge Sort
large
Quick Sort
code duplication

Program to Interface


Student Records
Sortable
identify the aspects of your application that vary and separate them from what stays the same
small
Bubble Sort
medium
Merge Sort
large
Quick Sort
X sorting methods vary

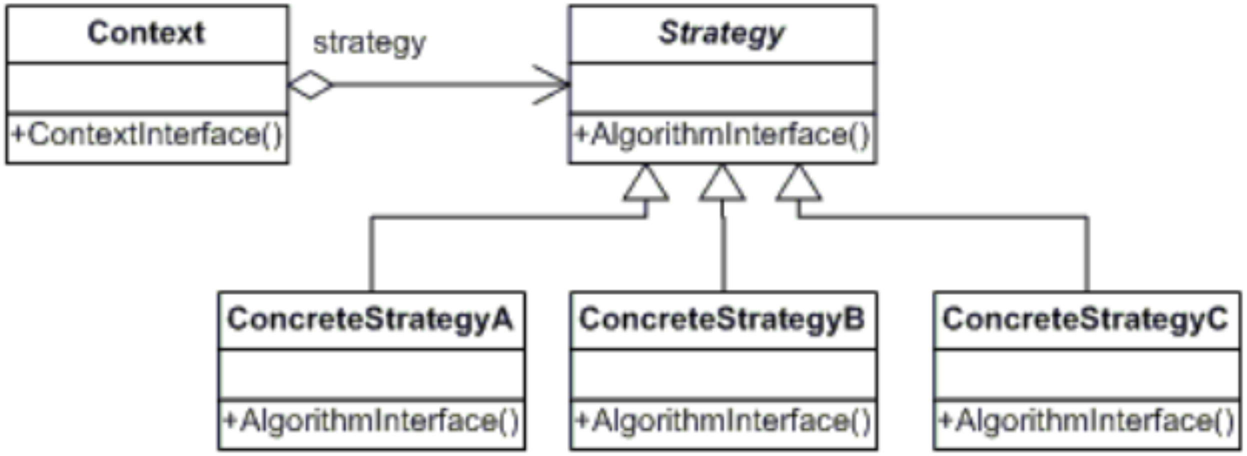
Program to Interface



Design Principles

Favor composition over inheritance

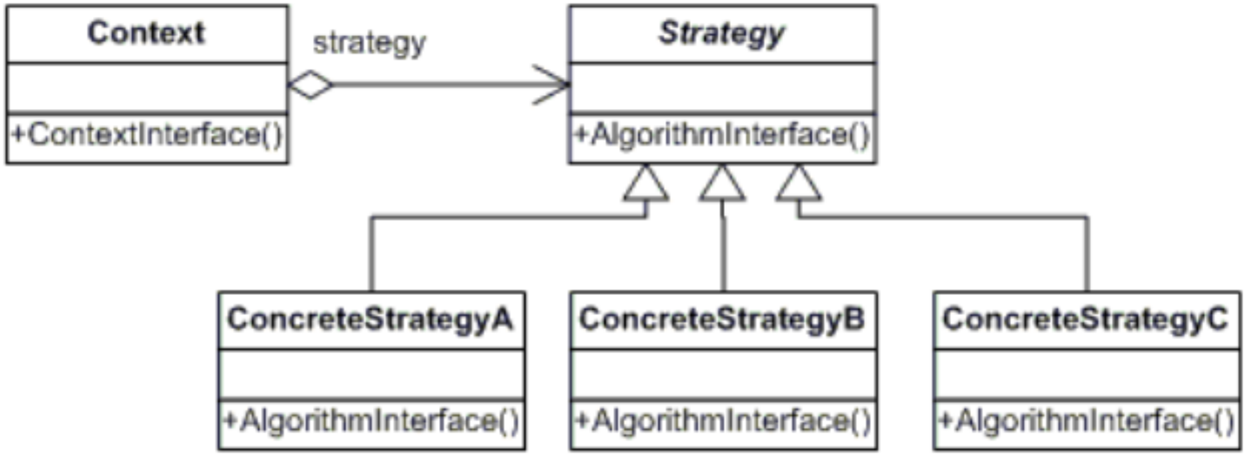
Solution



The Strategy Pattern first identifies the behaviors or algorithms that vary and separate them from the system that stays the same.

These behaviors or algorithms are encapsulated in classes that implement a common interface. This enables the developer to program to an interface and not an implementation.

Participants



- Strategy declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- ConcreteStrategy implements the algorithm using the Strategy interface
- Context is configured with a ConcreteStrategy object maintains a reference to a Strategy object may define an interface that lets Strategy access its

Inclass Exercise

Microservice Architecture:
ACID properties and CAP Theorem