More Patterns…

Acknowledgement: Head-first design patterns

Chain of Responsibility

Acknowledgement: Head-first design patterns

Problem

Scenario: *Paramount Pictures* has been getting more email than they can handle since the release of the Java-powered Ironman game. From their analysis they get four kinds of email:

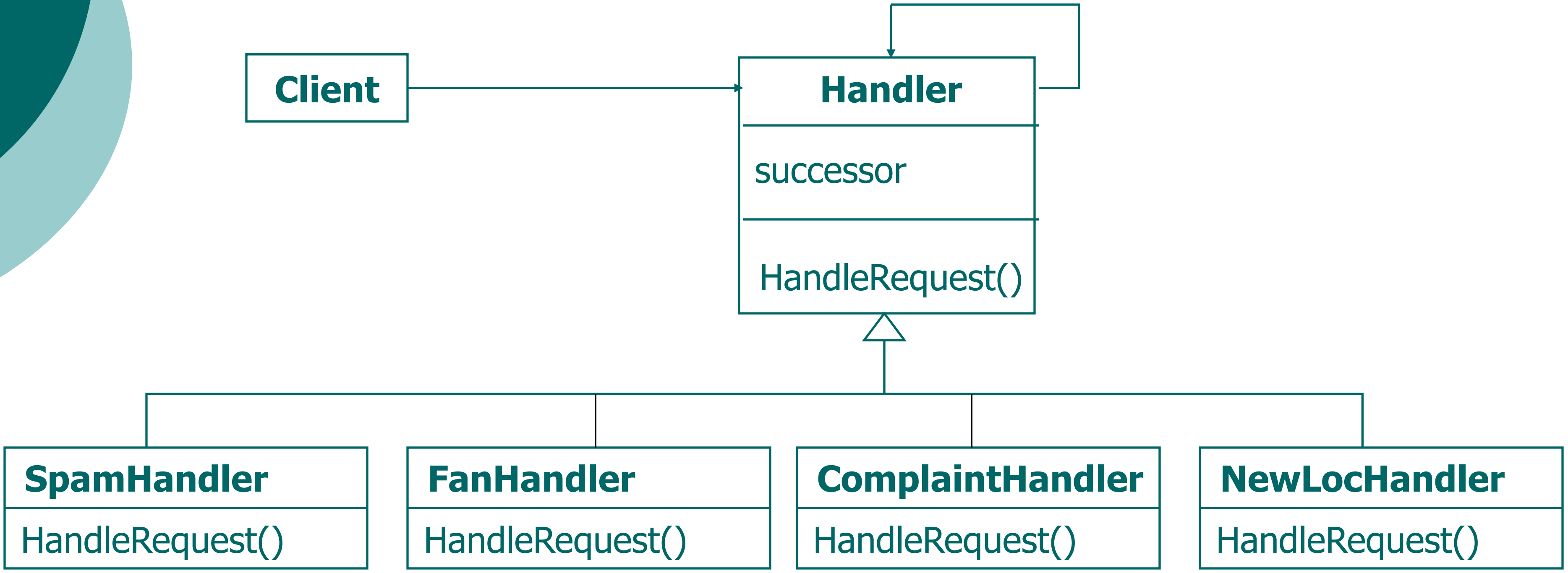
- Fan mail from customers that love the new 1 in 10 game
- Complaints from parents whose kids are addicted to the game
- Requests to put machines in new locations
- Spam

Task: They need you to create a design that can use the detectors to handle incoming email.

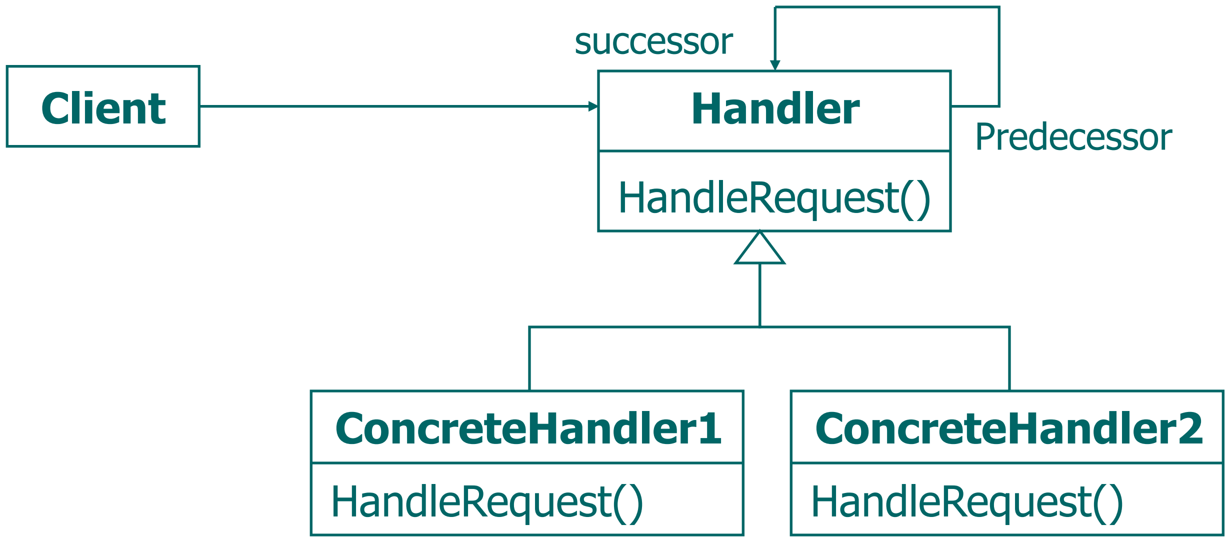
Chain of Responsibility

- Create a chain of objects that examine a request
- Each object examines the request and handles it, or passes it on to the next object in the chain.

Chain of Responsibility – Email Handler



Chain of Responsibility - Structure



Participants

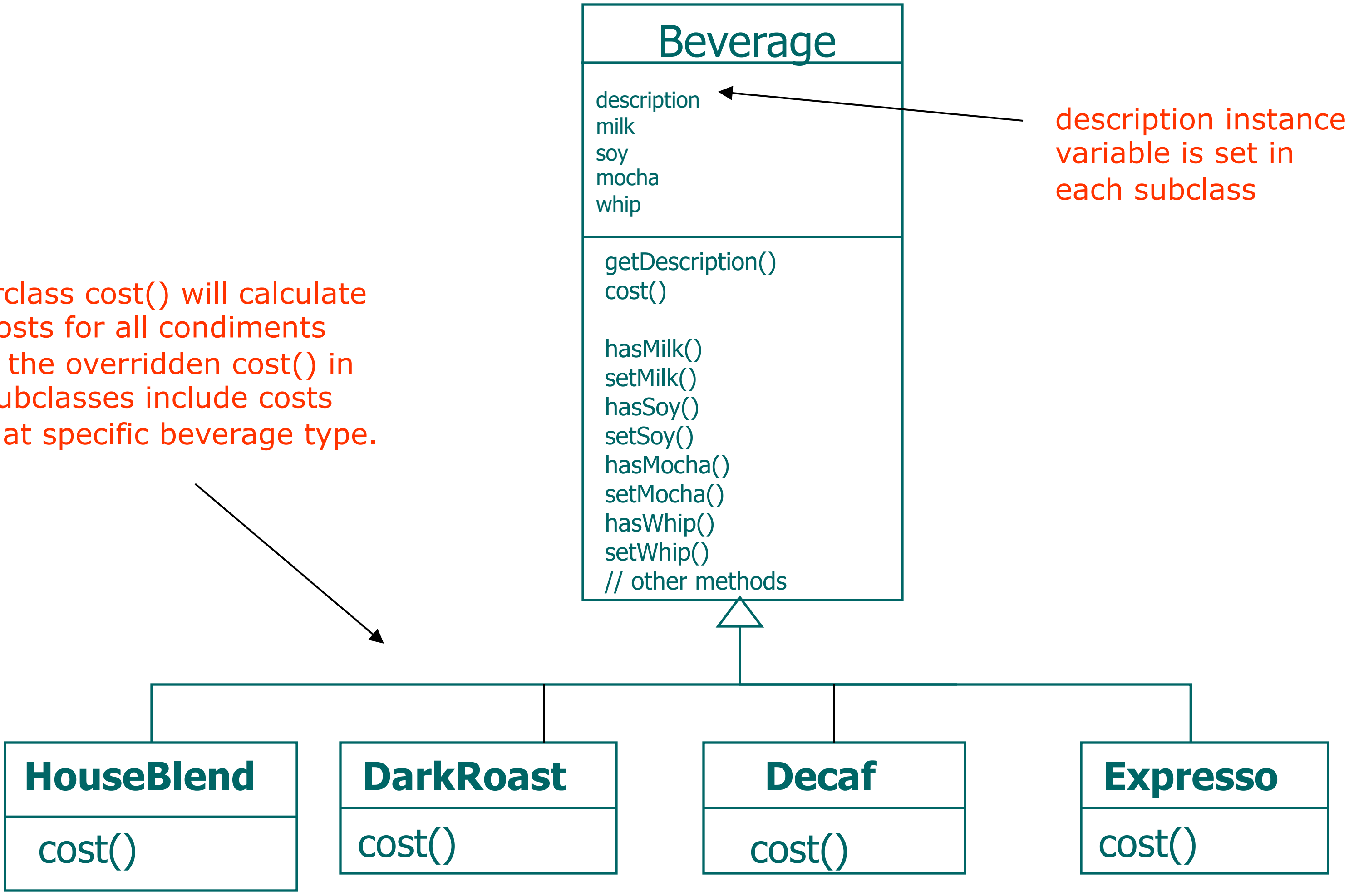
- Handler
  - Defines interface for handling request
  - Implements successor link
- ConcreteHandler
  - Handles all requests for which it is responsible
  - Has access to successor
  - Delegates request to successor when not handled directly
- Client
  - Initiates request to a ConcreteHandler

Benefits and Drawbacks

- Decouples the sender of the request and its receivers
- Simplifies the object because it doesn't have to know the chain' entire structure
- Allows for adding or removing responsibilities dynamically by changing the members or order of the chain
- Commonly used in windows systems to handle events like mouse clicks and keyboard events
- Execution of a request isn't guaranteed
- Can be hard to observe the runtime characteristics and debug

Decorator Pattern

Example - Starbuzz Coffee

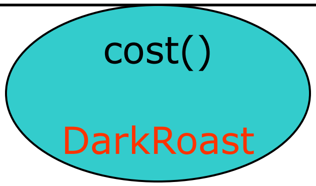


description instance variable is set in each subclass

superclass cost() will calculate the costs for all condiments while the overridden cost() in the subclasses include costs for that specific beverage type.

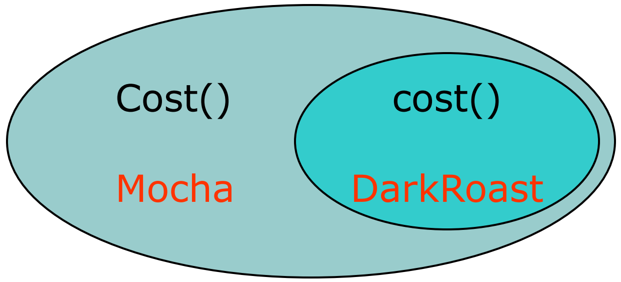
Constructing a drink order with Decorators

- Start with the DarkRoast Object



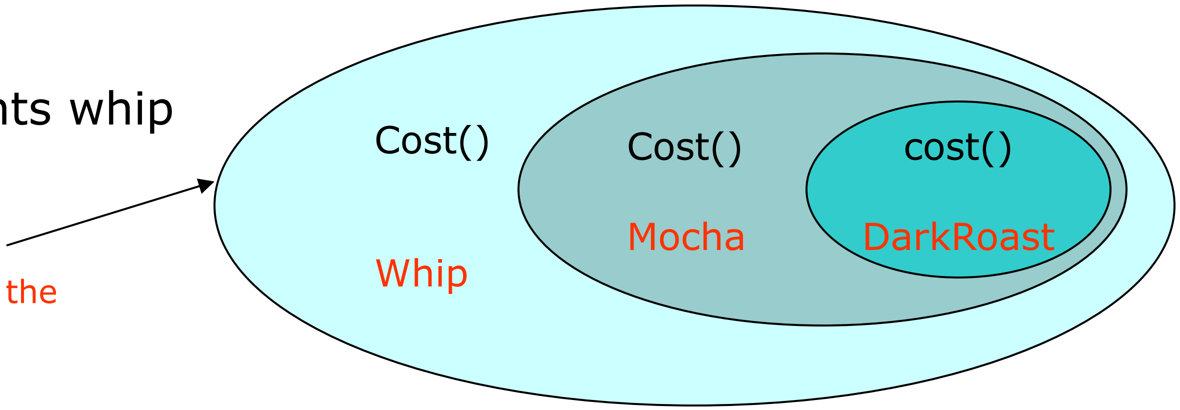
- Customer wants Mocha

Mocha object type mirrors the object it is decorating.



- Customer wants whip

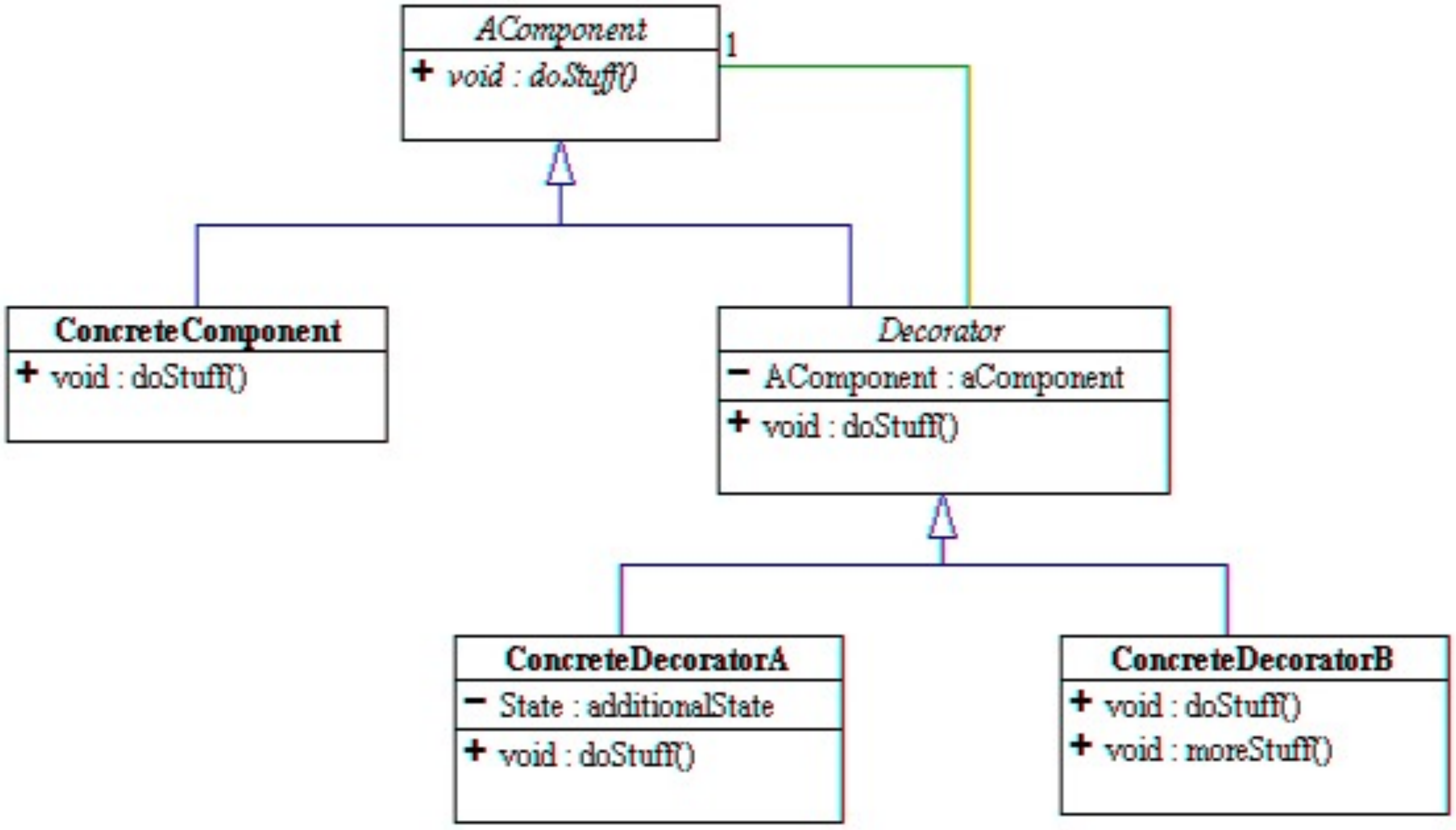
Whip object type mirrors the object it is decorating.



Decorator pattern

- Decorators have the same super type as the objects they decorate
- You can use one or more decorators to wrap an object
- Given that the decorator has the same super type as the object it decorates, we can pass around a decorated object in place of the original object
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like

General Structure



Mediator Pattern

Problem

Scenario: Bob has an amazing auto-house. When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing.

Bob is looking to add additional features:

- No coffee on the weekends…
- Turn of the sprinkler 15 min. before a shower is scheduled…
- Set the alarm early on trash days…

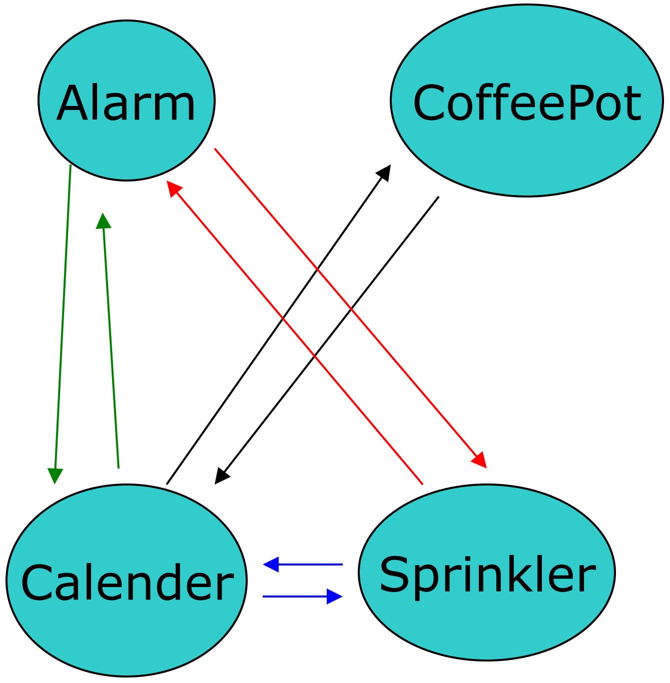
Problem Representation

Alarm

```
onEvent() {
  checkCalender()
  checkSprinkler()
  startCoffee()

 // more stuff
}
```

CoffeePot

```
onEvent() {
  checkCalender()
  checkAlarm()
  startCoffee()

 // more stuff
}
```



Calender

```
onEvent() {
  checkDayofWeek()
  doSprinkler()
  doCoffee()
  doAlarm()

 // more stuff
}
```

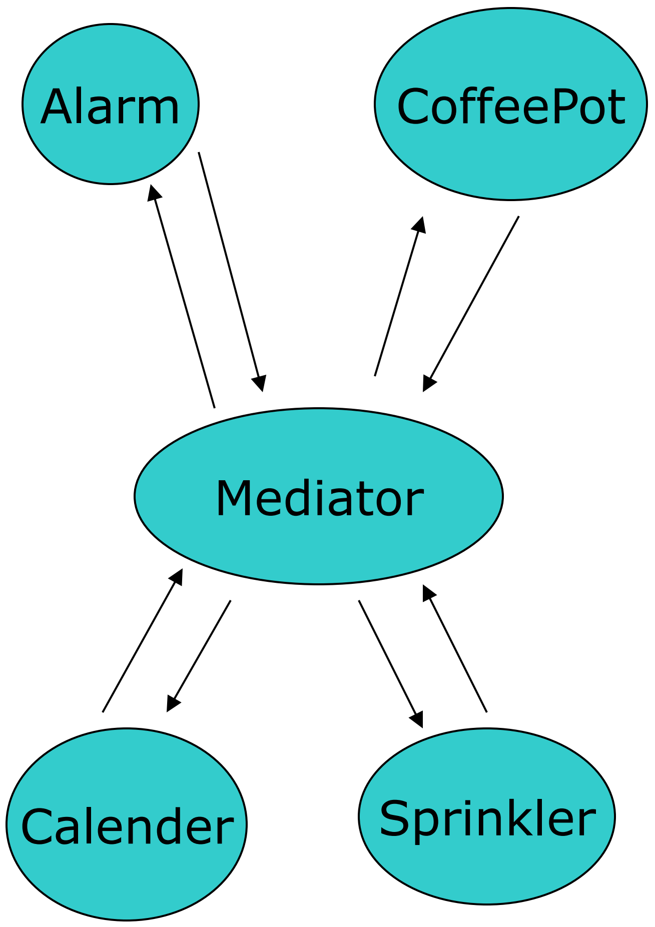
Alarm

```
onEvent() {
  checkCalender()
  checkShowerr()
  checkTemp()
  checkWeather()

 // more stuff
}
```

Mediator in action…

Mediator

```
if (alarmEvent) {
  checkCalender()
  checkShower()
  checkTemp()
}

if (weekend) {
  checkWeather()
  // do more stuff
}

if (trashDay){
  resetAlarm()
  //do more stuff
}
```



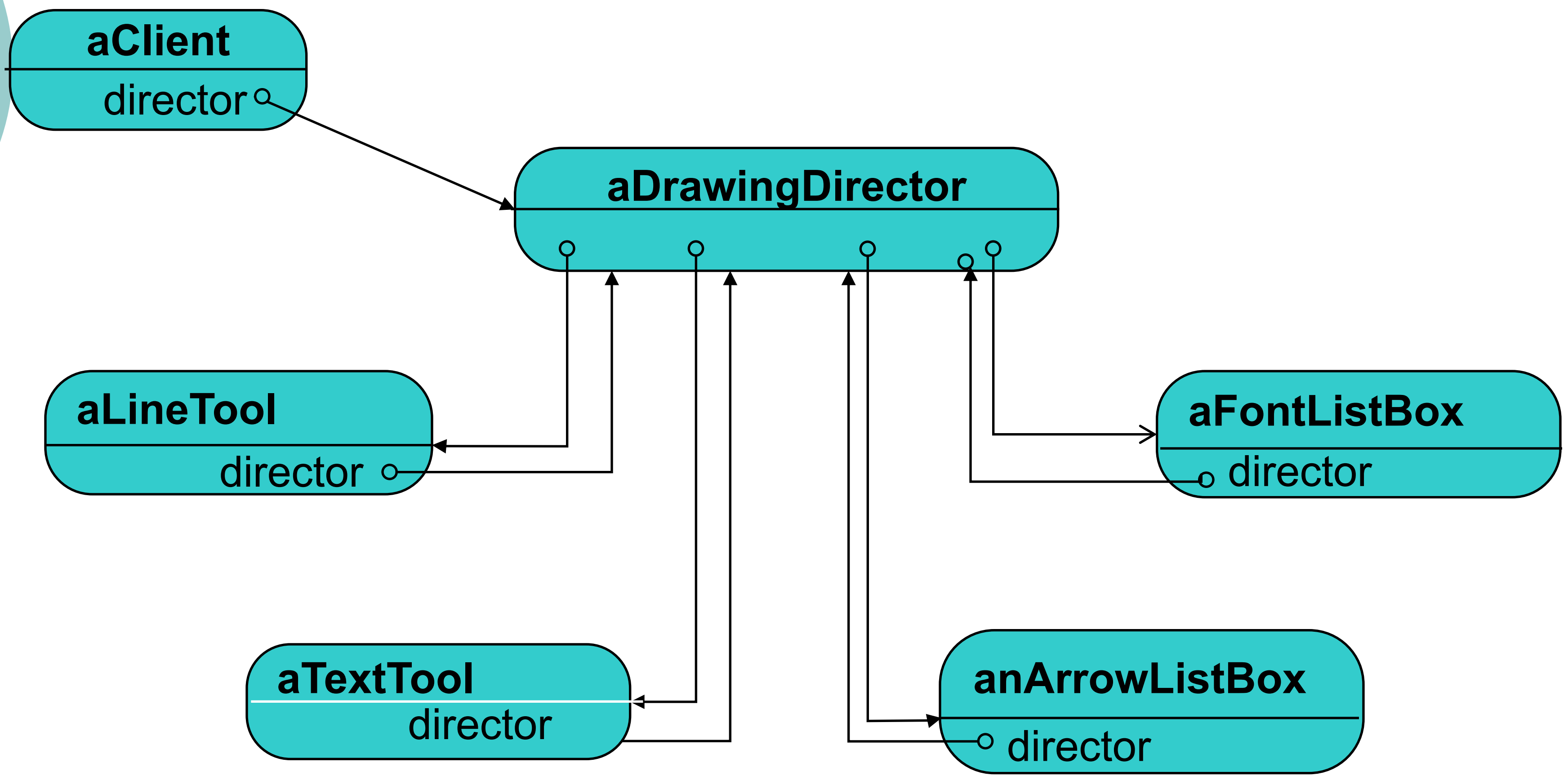
Example: Drawing Editor

- The GUI widgets for a drawing editor are interrelated
- Examples:
  - Insert text enables font widgets and fill widgets
  - Insert 3-D object enables fill-in oriented widgets
  - Insert line or arc enables arrowhead options.
- How to coordinate all these widgets?
- Do not want to have every widget know about every other widget!

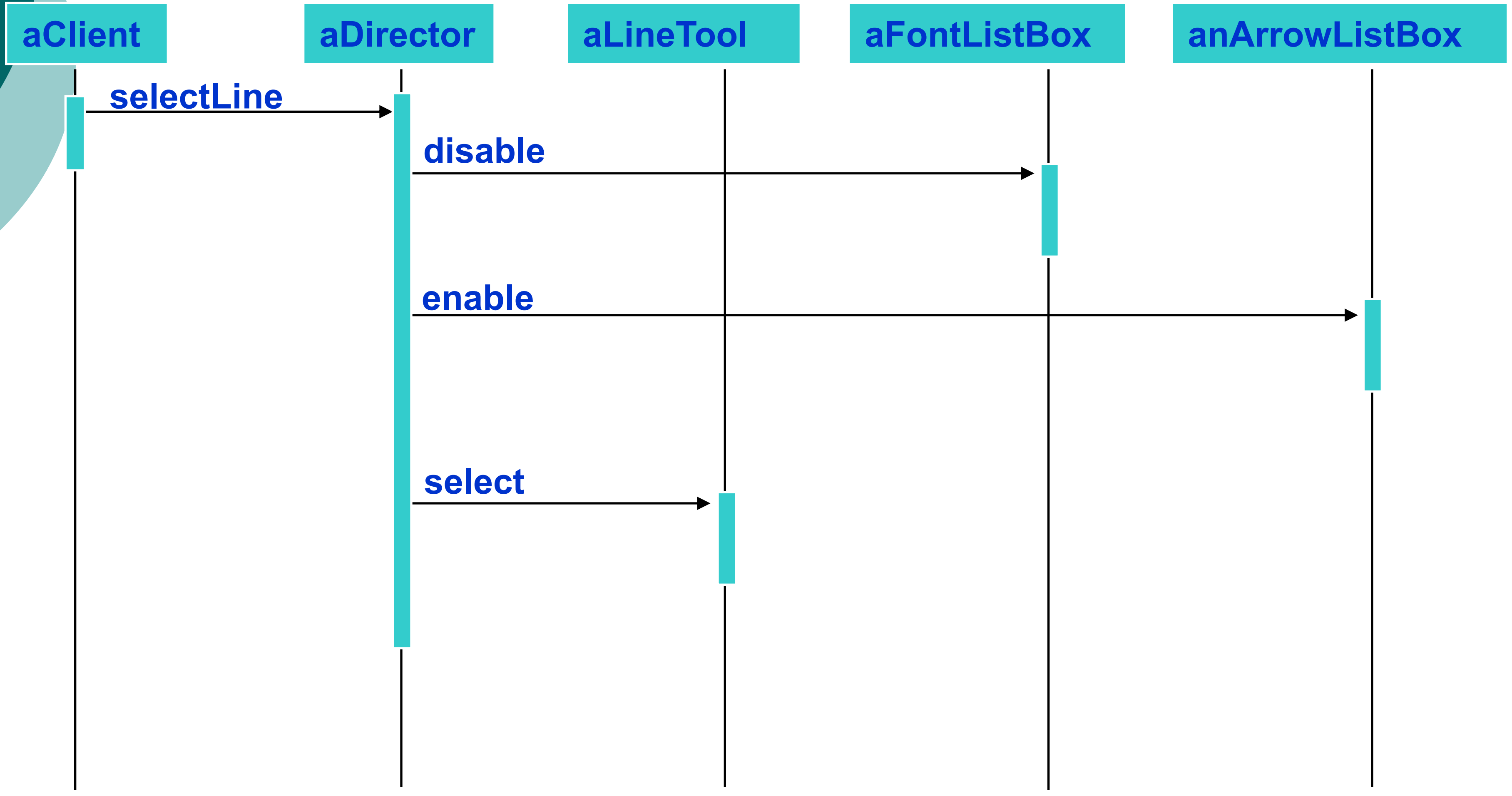
Solution: Mediator Pattern

- Mediator/director objects synchronize object collections.
- Mediator observes all widgets it coordinates
  - Example: for drawing, every change to drawing tool is broadcasted by mediator to other connected widgets
- More efficient and comprehensible than every widget watching every other one.
- Centralized coordination.

Example: Drawing Director



Interaction Chart



Participants

- Mediator
  - Interface for communications among colleague objects
  - Knows all the colleagues
  - Implements cooperative behavior
- Colleagues
  - Know their director / mediator
  - Communicate with mediator to distribute information to other colleagues

Benefits and Drawbacks

- Increases the reusability of the objects supported by the Mediator by decoupling them from the system
- Simplifies maintenance of the system by centralizing logic
- Simplifies and reduces the variety of messages sent between objects in the system
- Without proper design, the Mediator object itself can become overly complex

Factory pattern

Acknowledgement: Head-first design patterns

461 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…

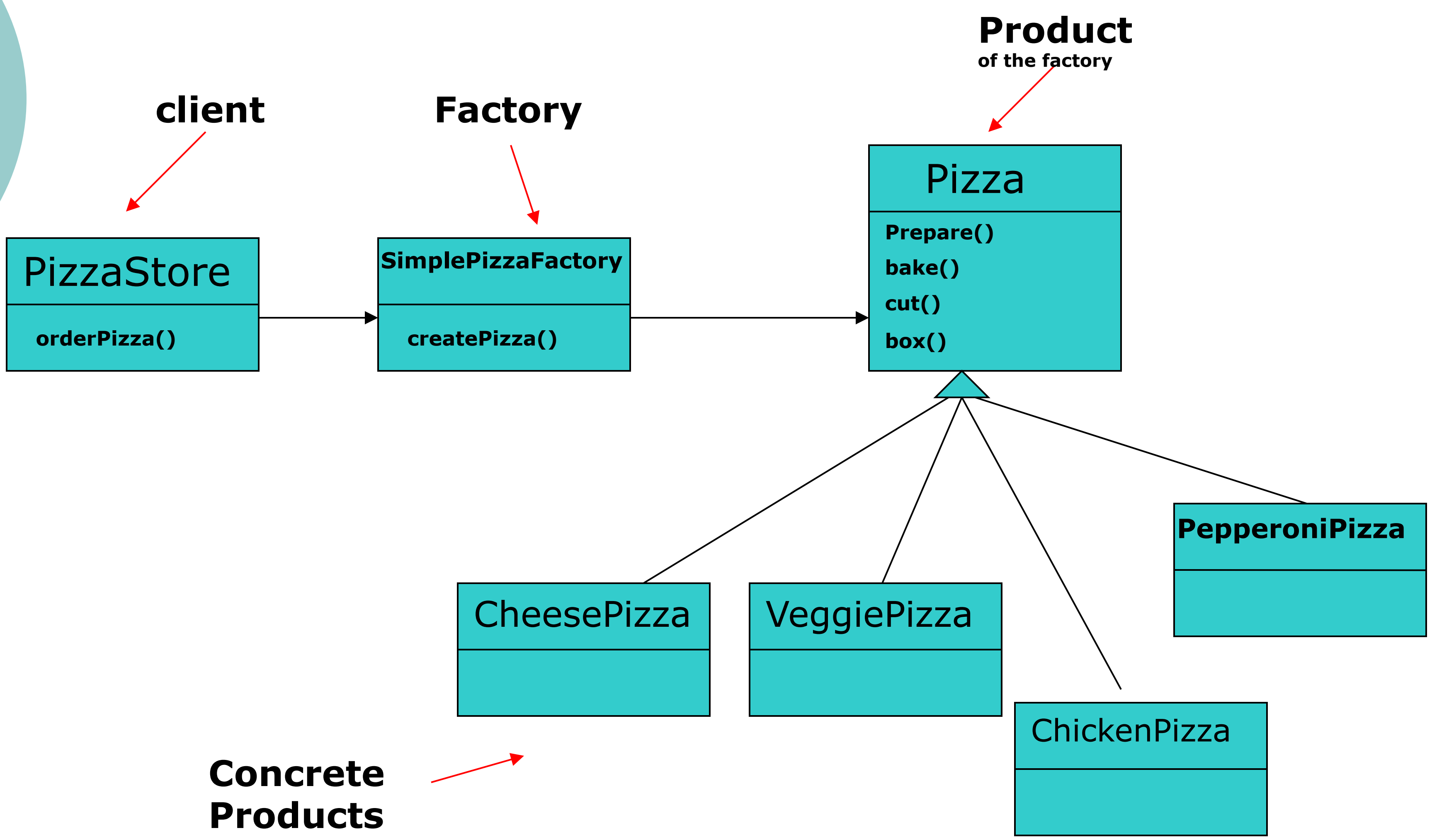
What' s the advantage of this?

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 - 461 Pizza Store

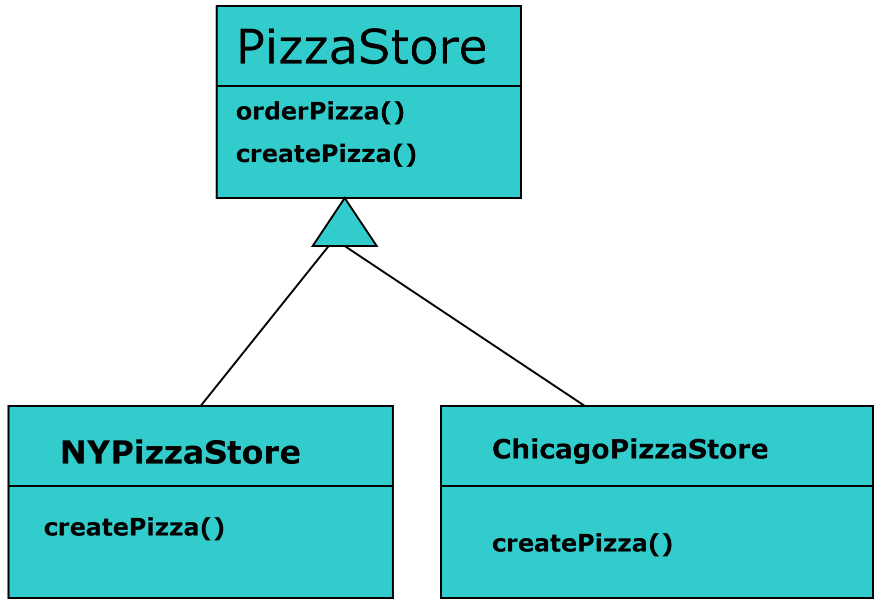
```
public class PizzaStore{
    Pizza orderPizza() {

        Pizza pizza = new Pizza();

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

A framework for the pizza store

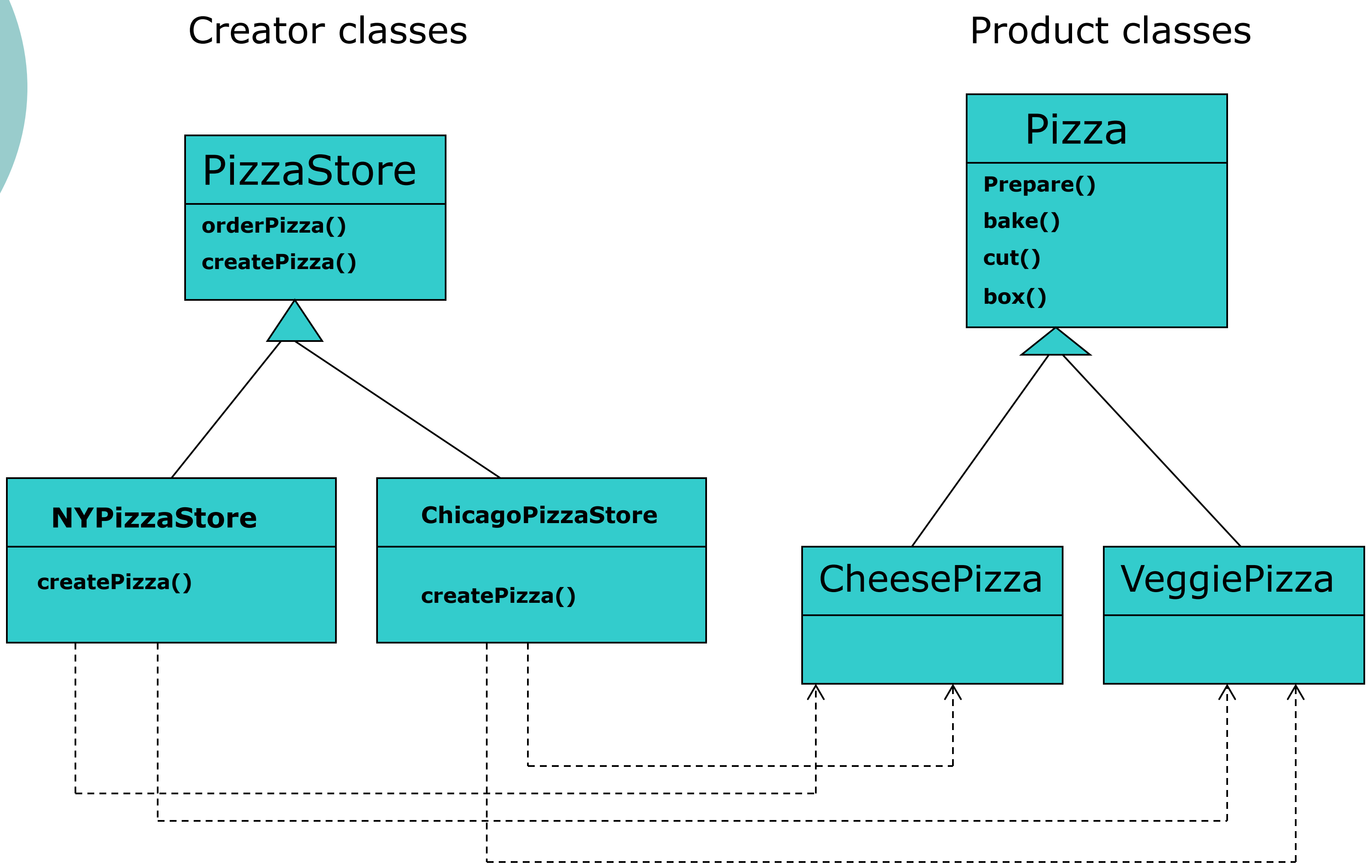
```
public abstract class PizzaStore {

    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)

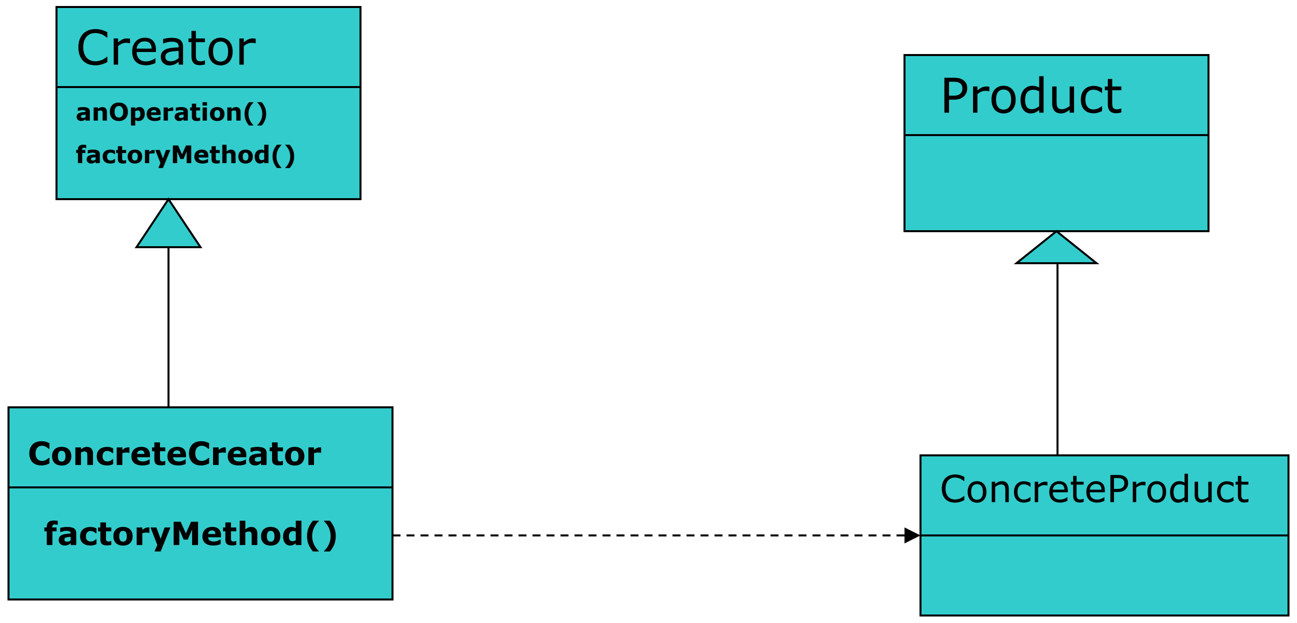


Factory method



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



Flyweight pattern

Acknowledgement: Head-first design patterns

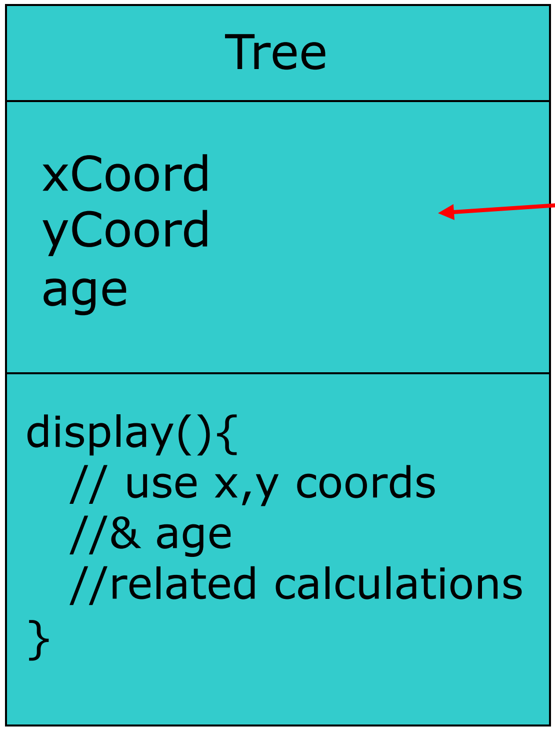
When do we use it?

- 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 have lots of trees in a particular landscape design

Tree class



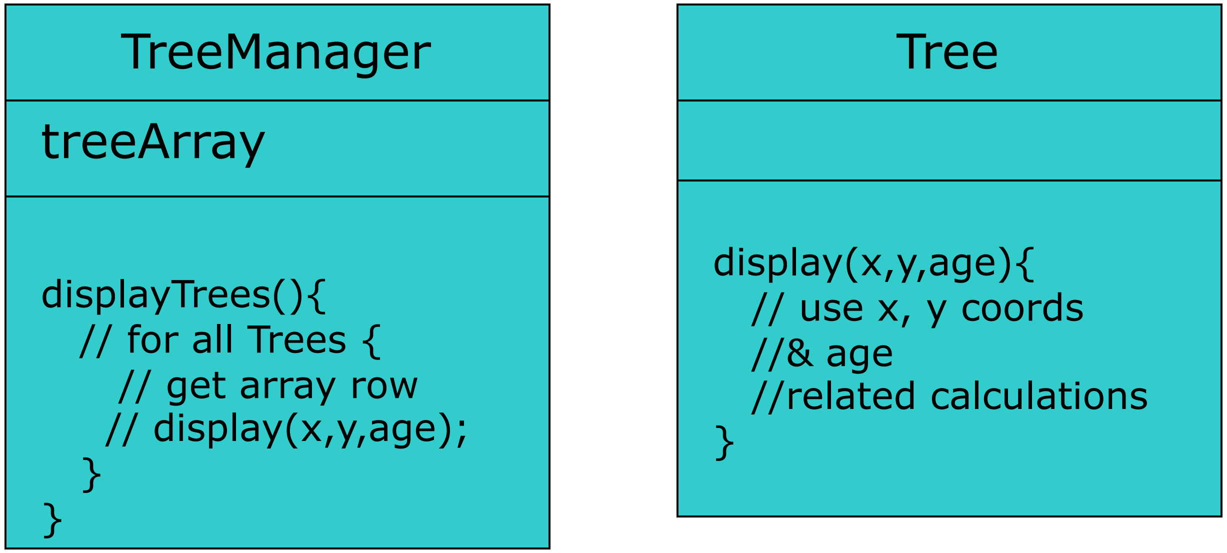
```
Tree

xCoord
yCoord
age

display(){
   // use x,y coords
   //& age
   //related calculations
}
```

Each Tree instance maintains its own state

What happens if a 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



```
displayTrees(){
   // for all Trees {
      // get array row
     // display(x,y,age);
   }
}
```

```
display(x,y,age){
   // use x, y coords
   //& age
   //related calculations
}
```

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.

Prototype pattern

Acknowledgement: Head-first design patterns

Prototype

- Allows for creation of new instances by copying existing instances

  (In java, it's done by clone() method – shallow copy. If deep copies are needed, they should be handled as serializable objects)

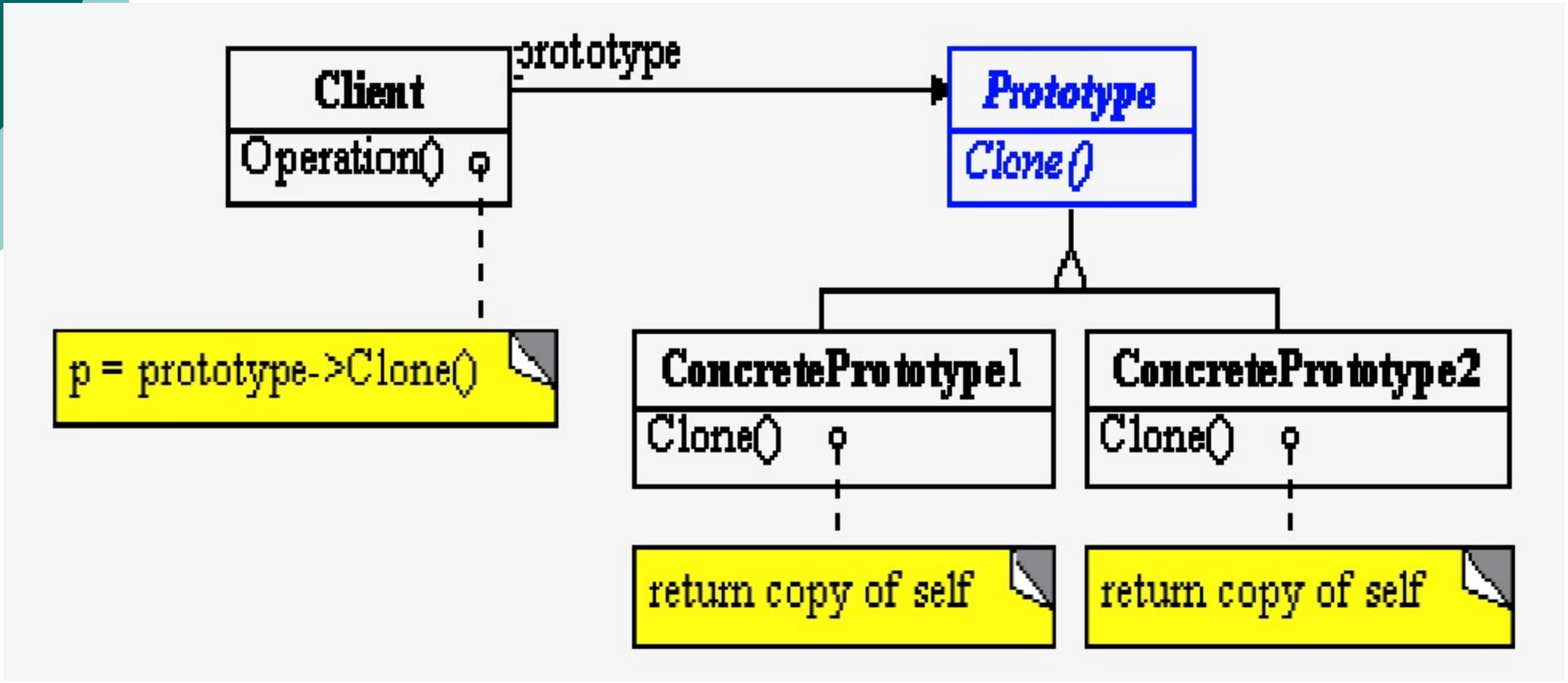
- Client can make new instances without knowing which specific class is being instantiated

- Provide an object like the one it should create

- This template object is a *Prototype* of the ones we want to create

- When we need new object, ask prototype to copy or clone itself

Abstract Structure



Participants

Client

- *Creates a new object by asking a prototype to clone itself*

Prototype

- *Declares an interface for cloning itself*

Concrete Prototype

- *Implements an operation for cloning itself*