Factory pattern

Acknowledgement: Head-first design patterns

461 Pizza Store

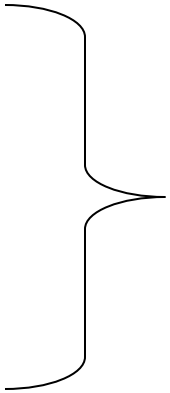
```
public class PizzaStore{
    Pizza orderPizza() {

            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…

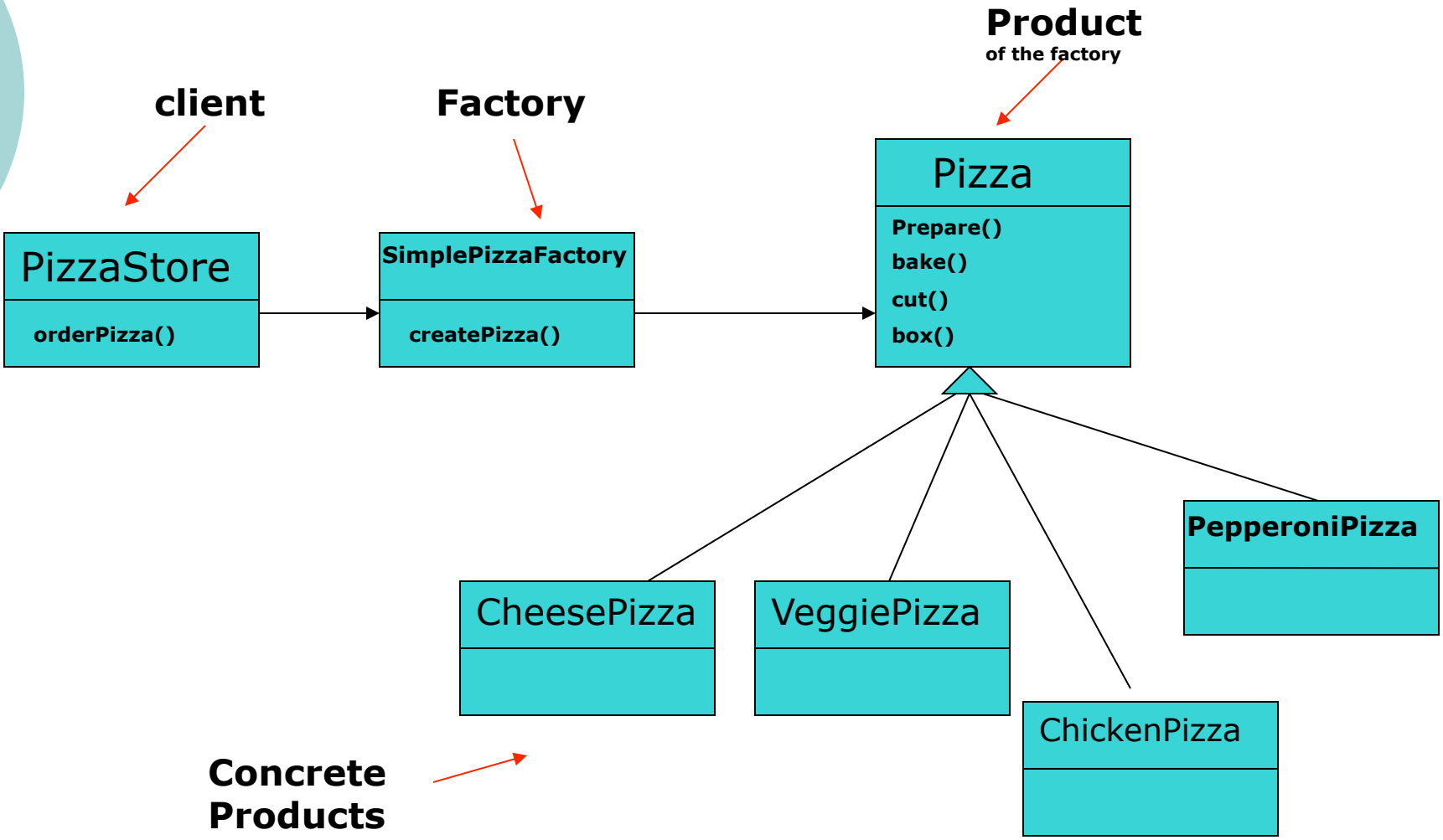
What's the advantage of this?

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

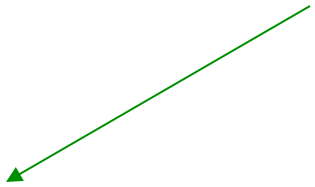


Revisiting - 461 Pizza Store

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

            Pizza pizza = new Pizza();

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

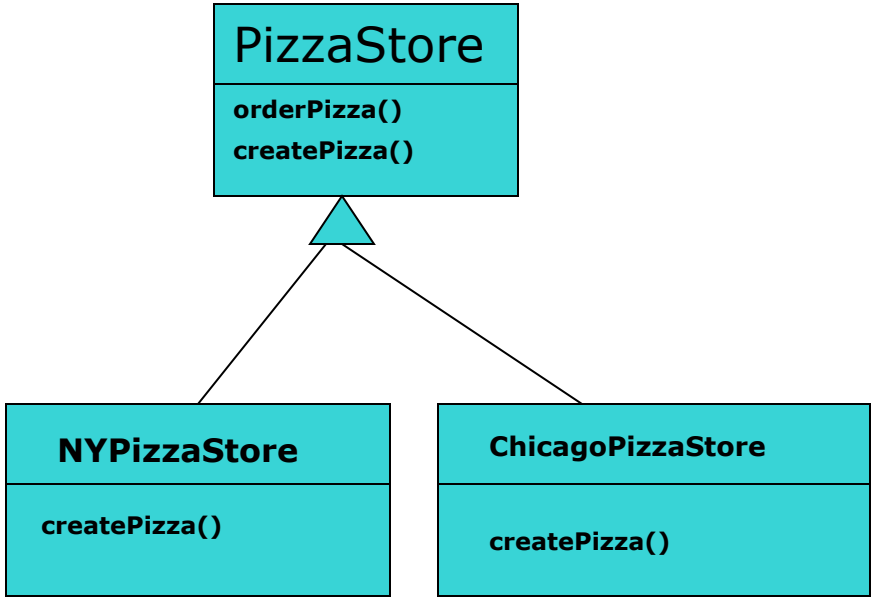
A framework for the pizza store

```
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);
}
```

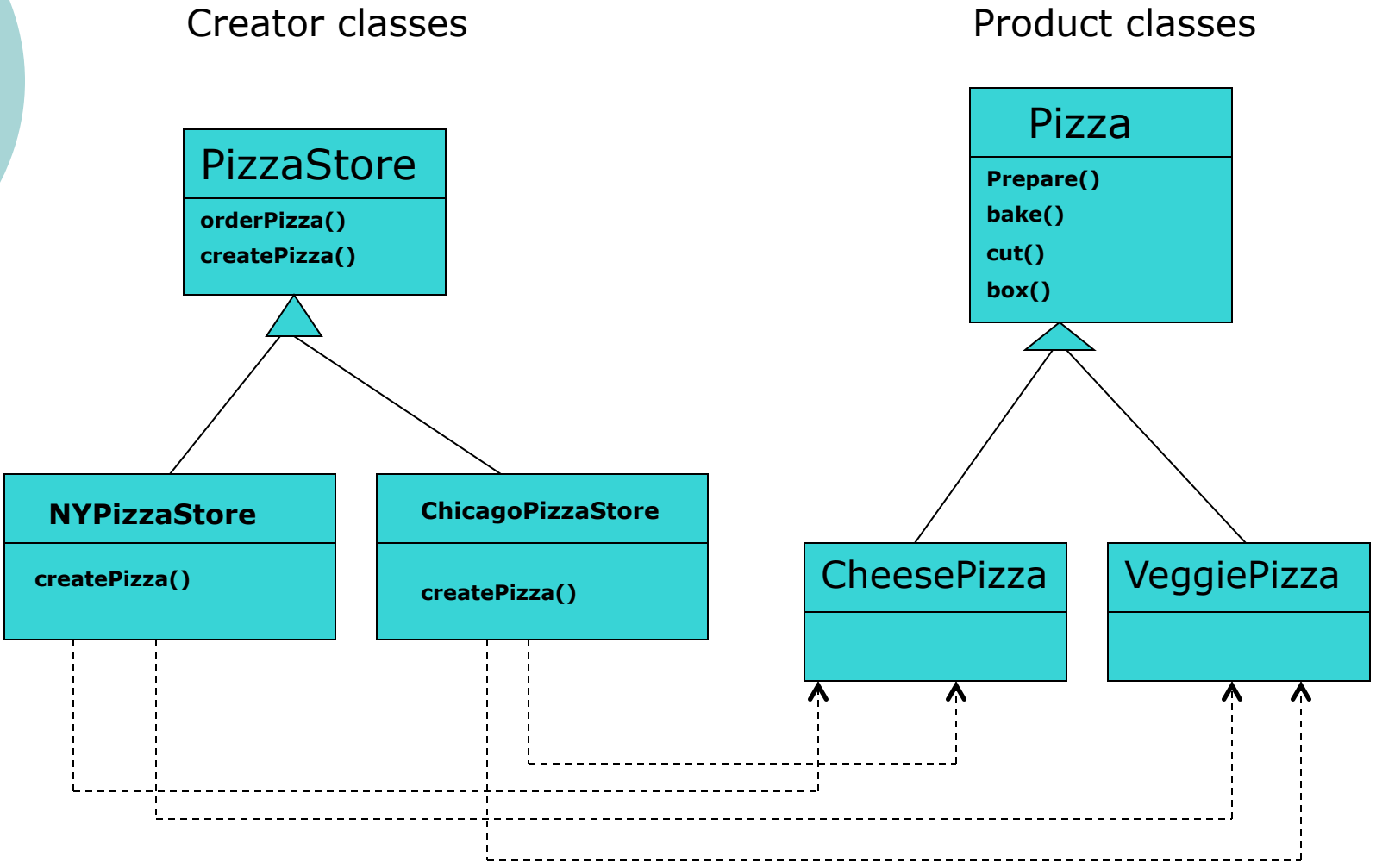


Factory method is now abstract

Allowing the subclasses to decide (creator classes)

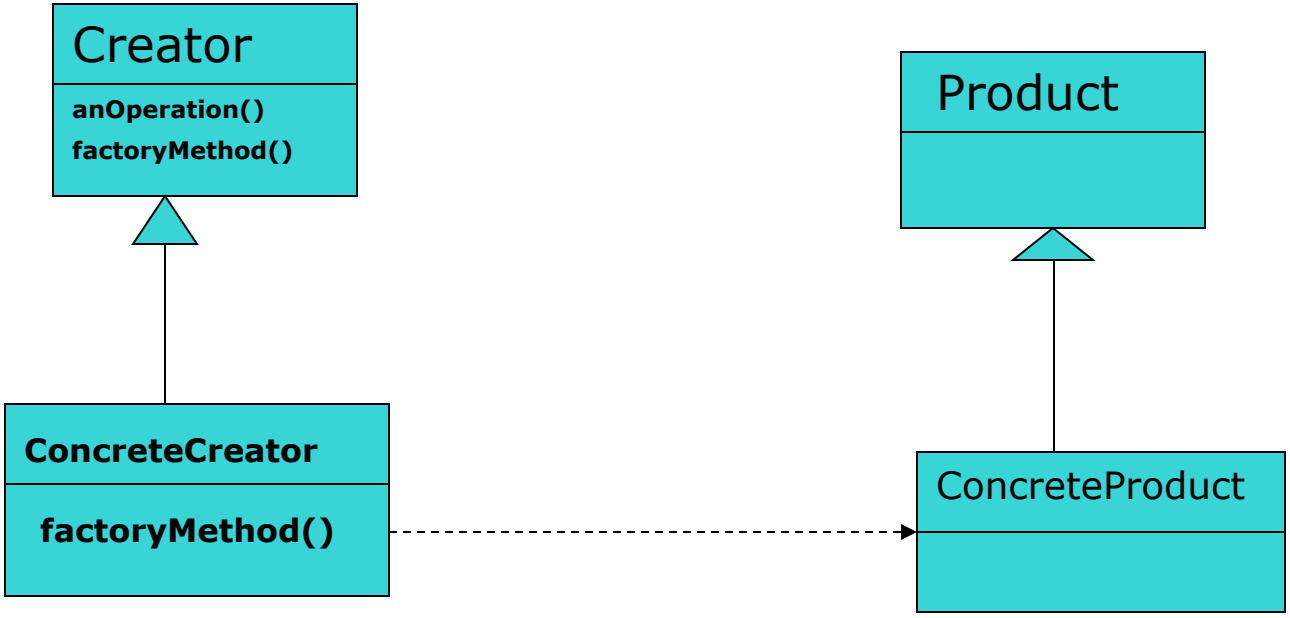


Factory method



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

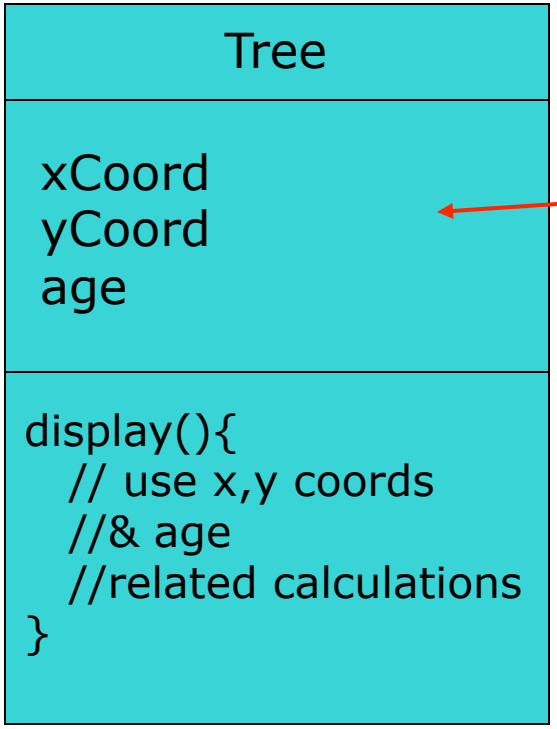
When do we use it?

- 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

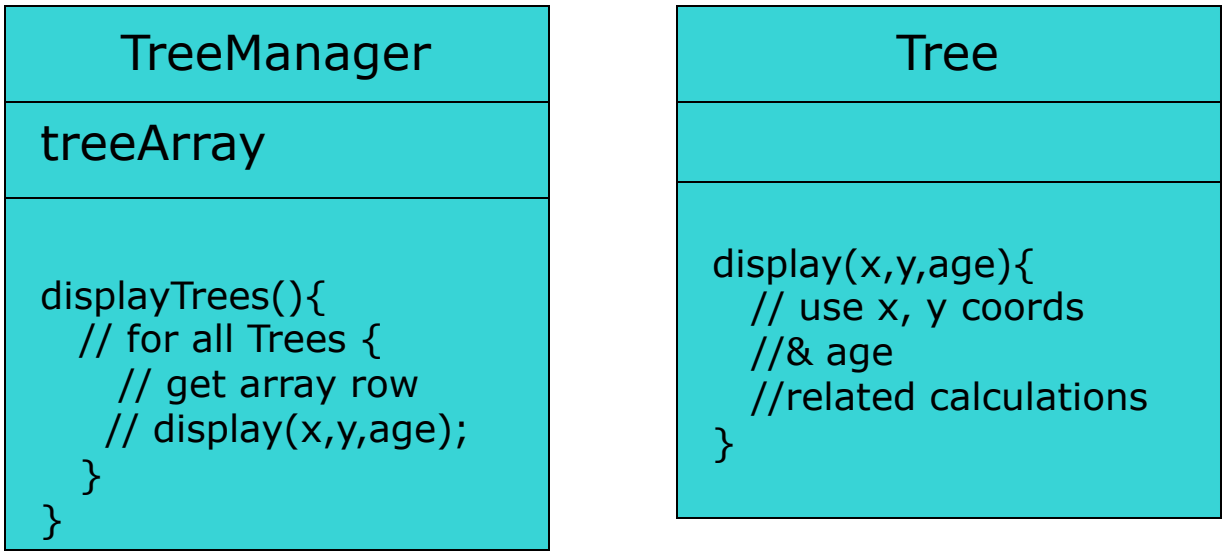


Each Tree instance maintains its own state

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



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)

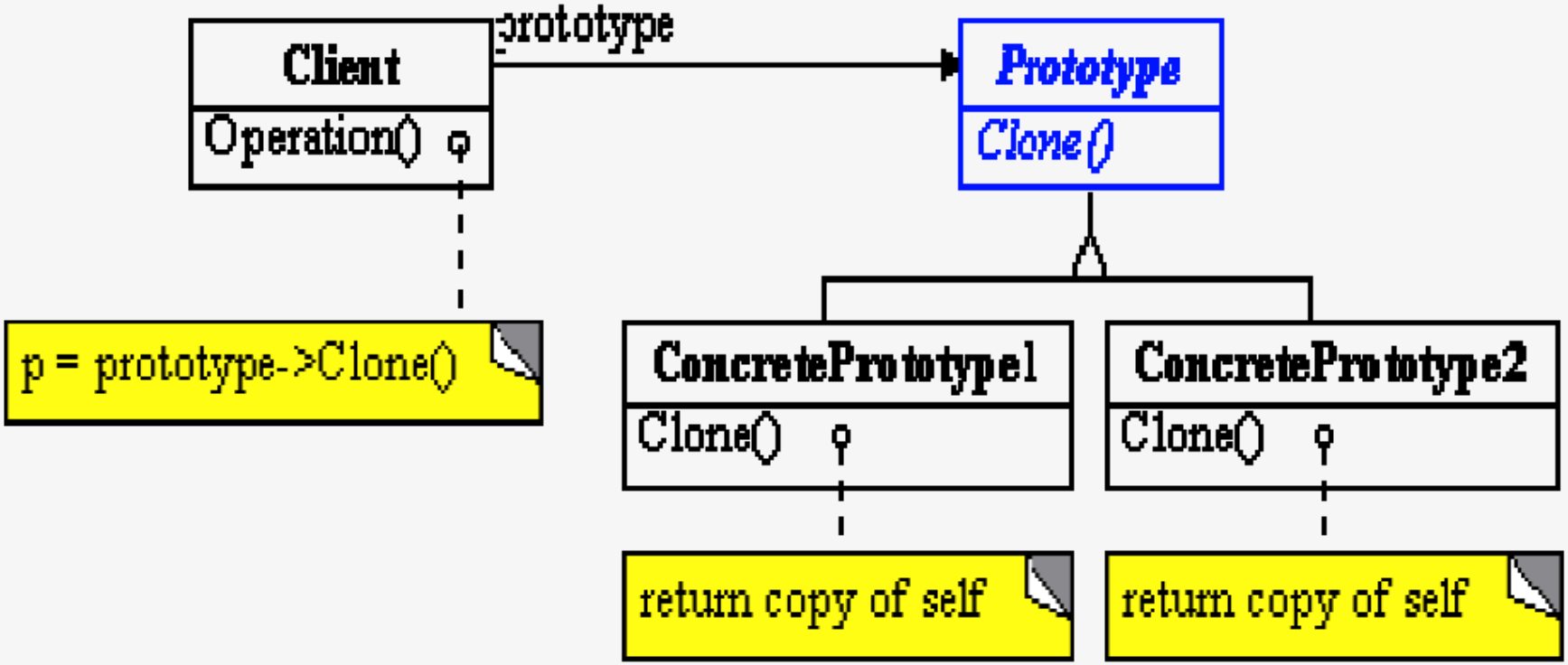
- Client can make new instances without knowing which specific class is being instantiated

- 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*

Memento Pattern

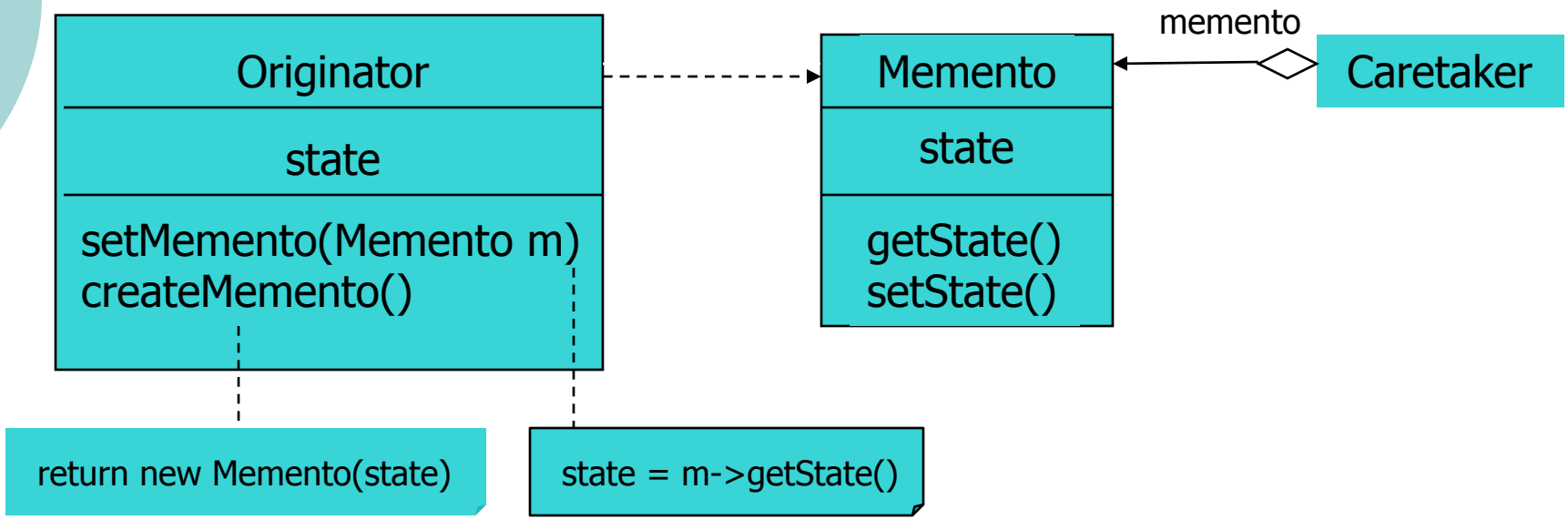
The Problem

- 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

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?
- May also be add overhead to storage in the Caretaker