

# CS130: Software Engineering

## Lab 3

# Agenda

1. Project Part A - Sample Presentation
2. Design Pattern Discussion
  - a. Strategy
  - b. Mediator
  - c. Observer
  - d. Factory
3. Design Patterns Practice
4. Time for Working on Part A

# **Design Pattern Discussion**

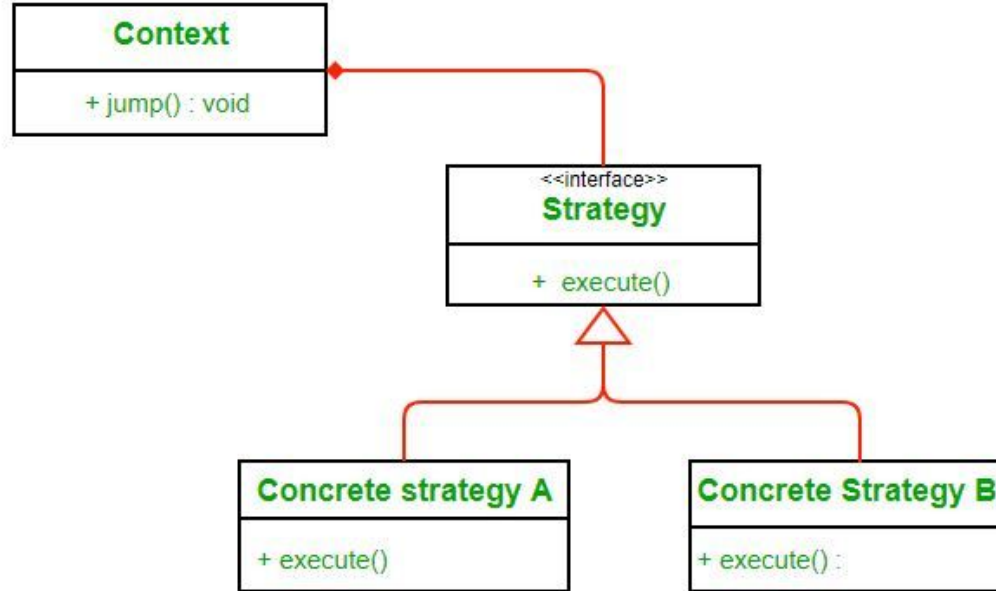
# Strategy Design Pattern

**Definition:** The strategy pattern is a software design pattern that *enables an algorithm's behavior to be selected at runtime*.

The strategy pattern

- defines a family of algorithms,
- encapsulates each algorithm, and
- makes the algorithms interchangeable within that family.

# Strategy Design Pattern



# Strategy Design Pattern - Example

Suppose we are building a game  
“Street Fighter”.

For simplicity assume that a  
character may have four moves  
that is kick, punch, roll and jump.  
Every character has roll and punch  
moves, but kick and jump are  
optional and have different types.

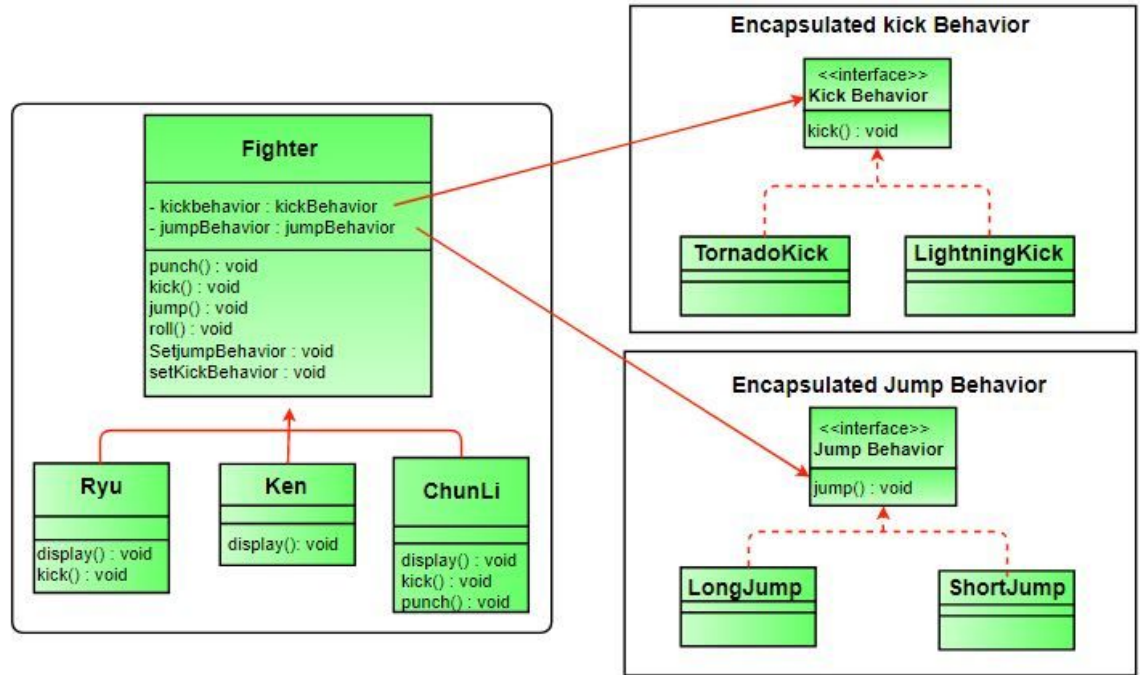
How would you model your  
classes?

# Strategy Design Pattern - Example

Suppose we are building a game  
“Street Fighter”.

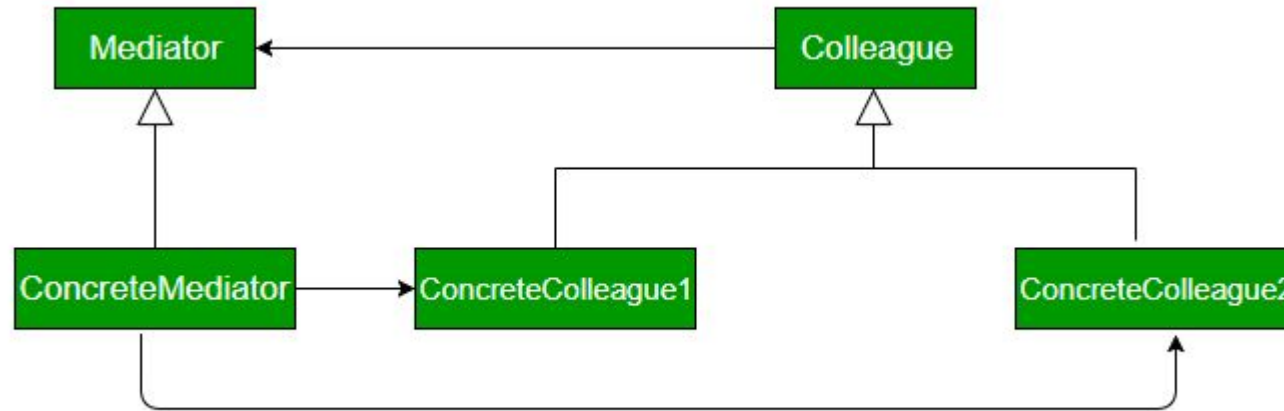
For simplicity assume that a  
character may have four moves  
that is kick, punch, roll and jump.  
Every character has roll and punc  
moves, but kick and jump are  
optional and have different types.

How would you model your  
classes?



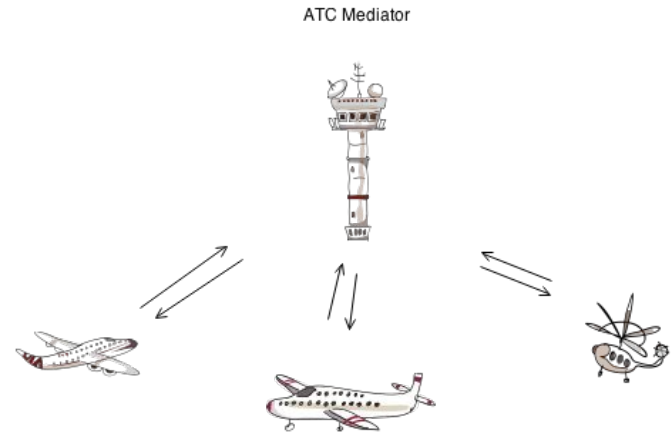
# Mediator Design Pattern

**Definition:** Mediator design pattern helps in *establishing loosely coupled communication between objects by introducing a layer in between* so that the interaction between objects happen via the layer.



# Mediator Design Pattern - Example

```
class MediatorDesignPattern
{
    public static void main(String args[])
    {
        IATCMediator atcMediator = new ATCMediator();
        Flight sparrow101 = new Flight(atcMediator);
        Runway mainRunway = new Runway(atcMediator);
        atcMediator.registerFlight(sparrow101);
        atcMediator.registerRunway(mainRunway);
        sparrow101.getReady();
        mainRunway.land();
        sparrow101.land();
    }
}
```



Example from: <https://www.geeksforgeeks.org/mediator-design-pattern/>

```

class ATCMediator implements IATCMediator {
    private Flight flight;
    private Runway runway;
    public boolean land;
    public void registerRunway(Runway runway) {
        this.runway = runway;
    }

    public void registerFlight(Flight flight) {
        this.flight = flight;
    }

    public boolean isLandingOk() {
        return land;
    }

    @Override
    public void setLandingStatus(boolean status)
    {
        land = status;
    }
}

interface Command {
    void land();
}

```

```

interface IATCMediator {
    public void registerRunway(Runway runway);
    public void registerFlight(Flight flight);
    public boolean isLandingOk();
    public void setLandingStatus(boolean status);
}

class Flight implements Command {
    private IATCMediator atcMediator;
    public Flight(IATCMediator atcMediator) {
        this.atcMediator = atcMediator;
    }

    public void land() {
        if (atcMediator.isLandingOk()) {
            System.out.println("Successfully Landed.");
            atcMediator.setLandingStatus(true);
        }
        else System.out.println("Waiting for landing.");
    }

    public void getReady() {
        System.out.println("Ready for landing.");
    }
}

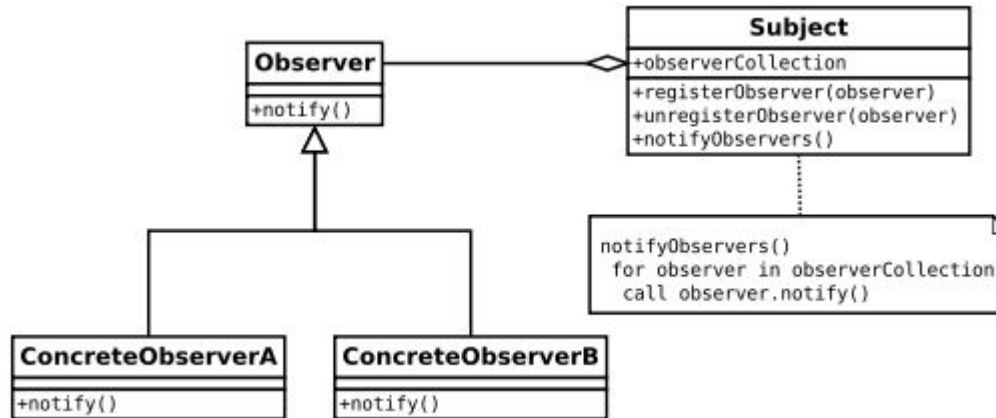
class Runway implements Command {
    private IATCMediator atcMediator;
    public Runway(IATCMediator atcMediator) {
        this.atcMediator = atcMediator;
        atcMediator.setLandingStatus(true);
    }

    @Override
    public void land() {
        System.out.println("Landing permission granted.");
        atcMediator.setLandingStatus(true);
    }
}

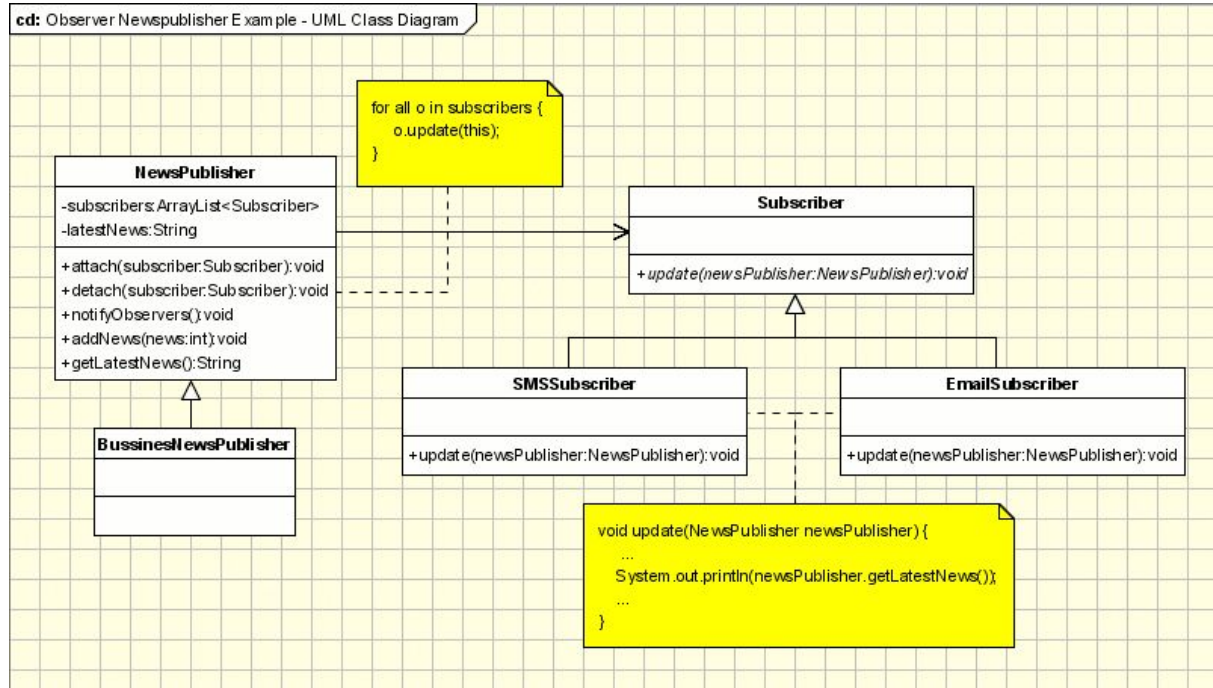
```

# Observer Design Pattern

**Definition:** The Observer Pattern defines a **one to many dependency** between objects so *that if one object changes state, all of its dependents are notified* and updated automatically.



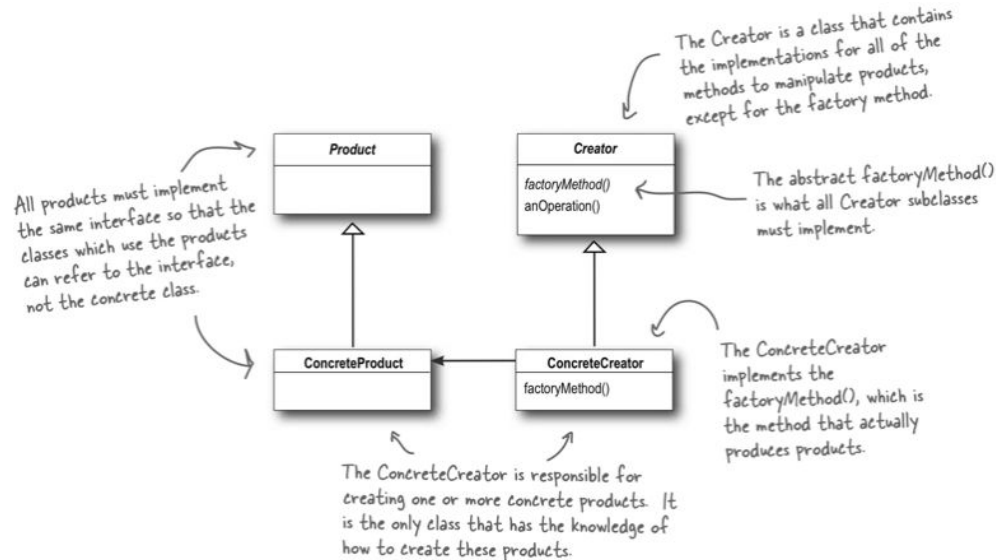
# Observer Design Pattern - Example



Example from <https://www.oodeign.com/observer-pattern.html>

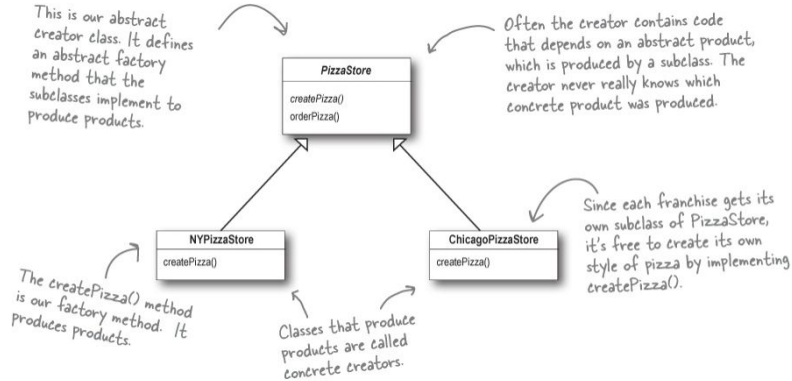
# Factory Design Pattern

**Definition:** Factory design pattern *defines an interface for creating an object but lets subclasses decide which class to instantiate*. Factory Method lets a class defer instantiation to subclasses. Decouples via inheritance, Encapsulates how objects created.

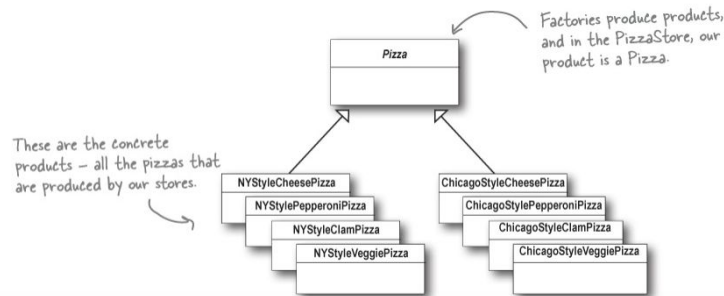


# Factory Design Pattern - lecture example revisited

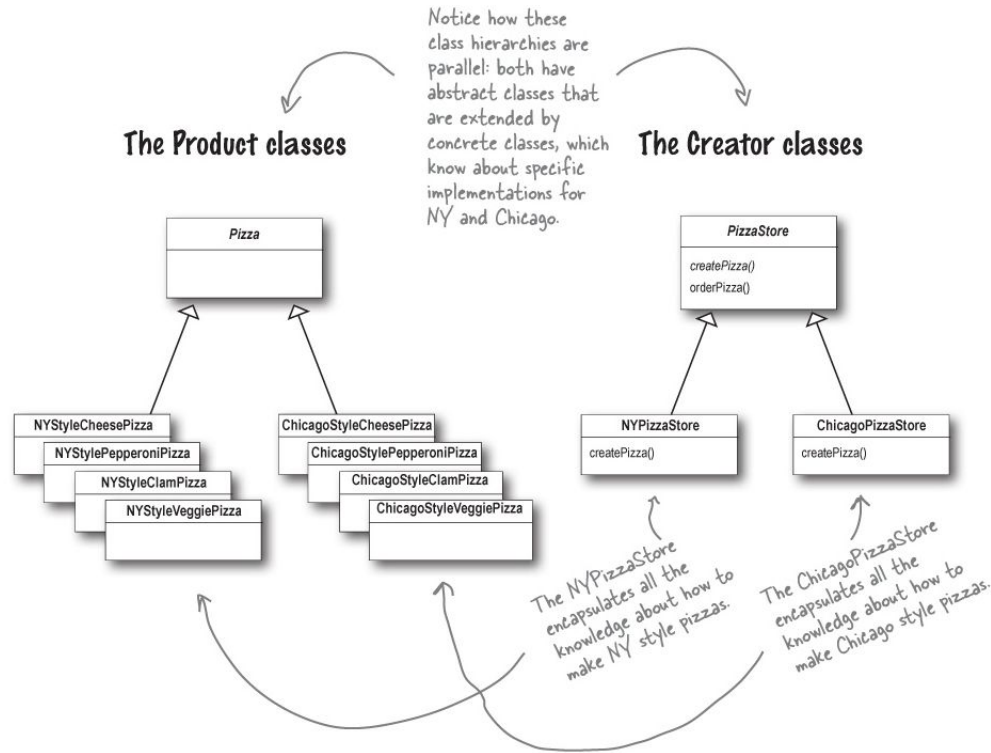
The Creator classes



The Product classes



# Factory Design Pattern - lecture example revisited



# Factory Design Pattern - Benefits

- Decouples via inheritance
- Think types of products factories make
- Open for extension closed for modification
- Encapsulates how objects created
- Client code decoupled from object creation code
- (Dependency Inversion Principle) Depend upon abstractions, not concrete classes

# Factory Design Pattern - Example

```
public abstract class CarFactory { // Abstract creator
    // Factory method -- delegated to concrete factories
    // (e.g. Honda, Toyota) to implement to their own choosing
    protected abstract Car createCar(String type);

    public Car orderCar(String type) {
        Car car = createCar(type);
        car.assemble();
        car.paint();
        return car;
    }
}

public class HondaFactory implements CarFactory { // Concrete creator
    Car createCar(String type) {
        if (type.equals("sedan")) {
            return new HondaSedan();
        } else if (type.equals("suv")) {
            return new HondaSUV();
        } else return null;
    }
}

public class ToyotaFactory implements CarFactory { // Concrete creator
    Car createCar(String type) {
        if (type.equals("sedan")) {
            return new ToyotaSedan();
        } else if (type.equals("suv")) {
            return new ToyotaSUV();
        } else return null;
    }
}
```

```
public abstract class Car { // Abstract product
    String name;
    int seats;

    void assemble() {
        System.out.println("Assembling " + name);
        System.out.println("Adding " + seats + " seats");
    }

    void paint() {
        System.out.println("Painting car");
    }
}

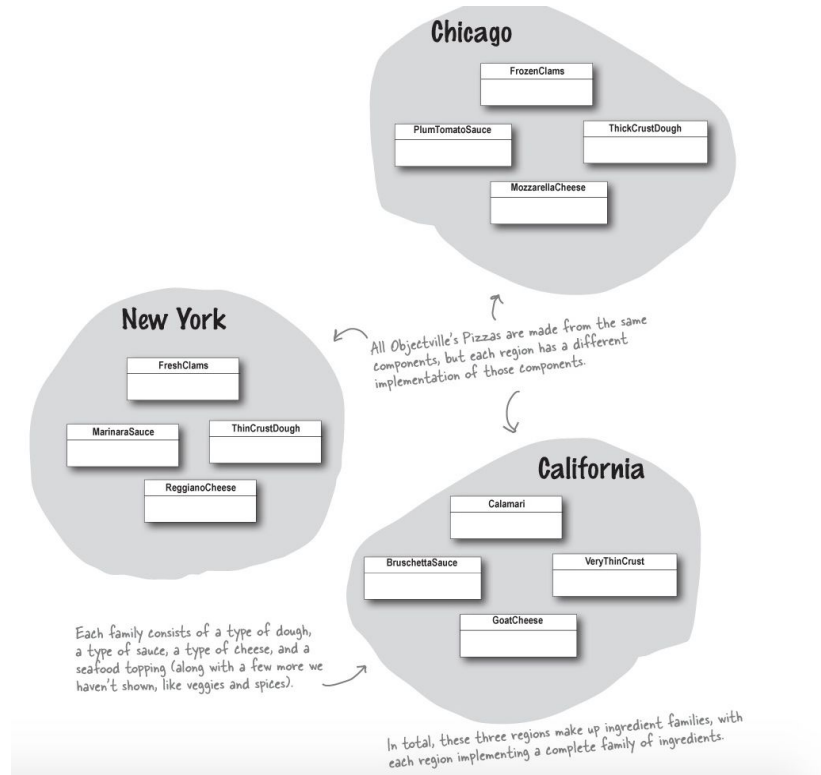
public class HondaSedan extends Car { // Concrete product
    public HondaSedan() {
        name = 'Honda Accord';
        seats = 4;
    }
}

public class ToyotaSedan extends Car { // Concrete product
    public ToyotaSedan() {
        name = 'Toyota Corolla';
        seats = 4;
    }
}

public class HondaSUV extends Car { // Concrete product
    public HondaSUV() {
        name = 'Honda Pilot';
        seats = 7;
    }
}

public class ToyotaSUV extends Car { // Concrete product
    public ToyotaSUV() {
        name = 'Toyota RAV-4';
        seats = 7;
    }
}
```

# Abstract Factory Design Pattern - the intent



# Abstract Factory Design Pattern - revisiting lecture example

```
public interface PizzaIngredientFactory {  
  
    public Dough createDough();  
    public Sauce createSauce();  
    public Cheese createCheese();  
    public Veggies[] createVeggies();  
    public Pepperoni createPepperoni();  
    public Clams createClam();  
  
}
```

**Building the ingredient factories**

For each ingredient we define a  
create method in our interface.

Lots of new classes here,  
one per ingredient.

Code can be found here:

<https://www.oreilly.com/library/view/head-first-design/0596007124/ch04.html>

# Abstract Factory Design Pattern - revisiting lecture example

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {  
  
    public Dough createDough() {  
        return new ThinCrustDough();  
    }  
  
    public Sauce createSauce() {  
        return new MarinaraSauce();  
    }  
  
    public Cheese createCheese() {  
        return new ReggianoCheese();  
    }  
  
    public Veggies[] createVeggies() {  
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };  
        return veggies;  
    }  
  
    public Pepperoni createPepperoni() {  
        return new SlicedPepperoni();  
    }  
  
    public Clams createClam() {  
        return new FreshClams();  
    }  
}
```

For each ingredient in the ingredient family, we create the New York version.

Building the New York ingredient factory

For veggies, we return an array of Veggies. Here we've hardcoded the veggies. We could make this more sophisticated, but that doesn't really add anything to learning the factory pattern, so we'll keep it simple.

The best sliced pepperoni. This is shared between New York and Chicago. Make sure you use it on the next page when you get to implement the Chicago factory yourself

New York is on the coast; it gets fresh clams. Chicago has to settle for frozen.

# Abstract Factory Design Pattern - revisiting lecture example

```
public abstract class Pizza {
    String name;

    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in store box");
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
        // code to print pizza here
    }
}
```

Each pizza holds a set of ingredients that are used in its preparation.

We've now made the prepare method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory.

Reworking the pizzas...

Our other methods remain the same, with the exception of the prepare method.

# Abstract Factory Design Pattern - revisiting lecture example

```
public class ClamPizza extends Pizza {  
    PizzaIngredientFactory ingredientFactory;  
  
    public ClamPizza(PizzaIngredientFactory ingredientFactory) {  
        this.ingredientFactory = ingredientFactory;  
    }  
  
    void prepare() {  
        System.out.println("Preparing " + name);  
        dough = ingredientFactory.createDough();  
        sauce = ingredientFactory.createSauce();  
        cheese = ingredientFactory.createCheese();  
        clam = ingredientFactory.createClam();  
    }  
}
```

← ClamPizza also stashes  
an ingredient factory.

↖ To make a clam pizza, the prepare  
method collects the right  
ingredients from its local factory.

↑  
If it's a New York factory,  
the clams will be fresh; if it's  
Chicago, they'll be frozen.

# Abstract Factory Design Pattern - revisiting lecture example

```
public class NYPizzaStore extends PizzaStore {  
  
    protected Pizza createPizza(String item) {  
        Pizza pizza = null;  
        PizzaIngredientFactory ingredientFactory =  
            new NYPizzaIngredientFactory();  
  
        if (item.equals("cheese")) {  
  
            pizza = new CheesePizza(ingredientFactory);  
            pizza.setName("New York Style Cheese Pizza");  
  
        } else if (item.equals("veggie")) {  
  
            pizza = new VeggiePizza(ingredientFactory);  
            pizza.setName("New York Style Veggie Pizza");  
  
        } else if (item.equals("clam")) {  
  
            pizza = new ClamPizza(ingredientFactory);  
            pizza.setName("New York Style Clam Pizza");  
  
        } else if (item.equals("pepperoni")) {  
  
            pizza = new PepperoniPizza(ingredientFactory);  
            pizza.setName("New York Style Pepperoni Pizza");  
  
        }  
        return pizza;  
    }  
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas.

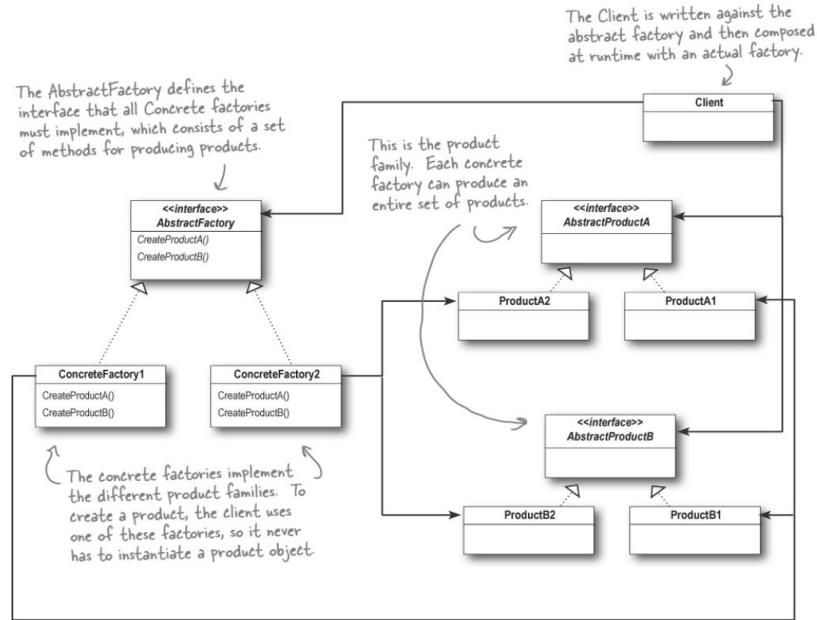
We now pass each pizza the factory that should be used to produce its ingredients.

Look back one page and make sure you understand how the pizza and the factory work together!

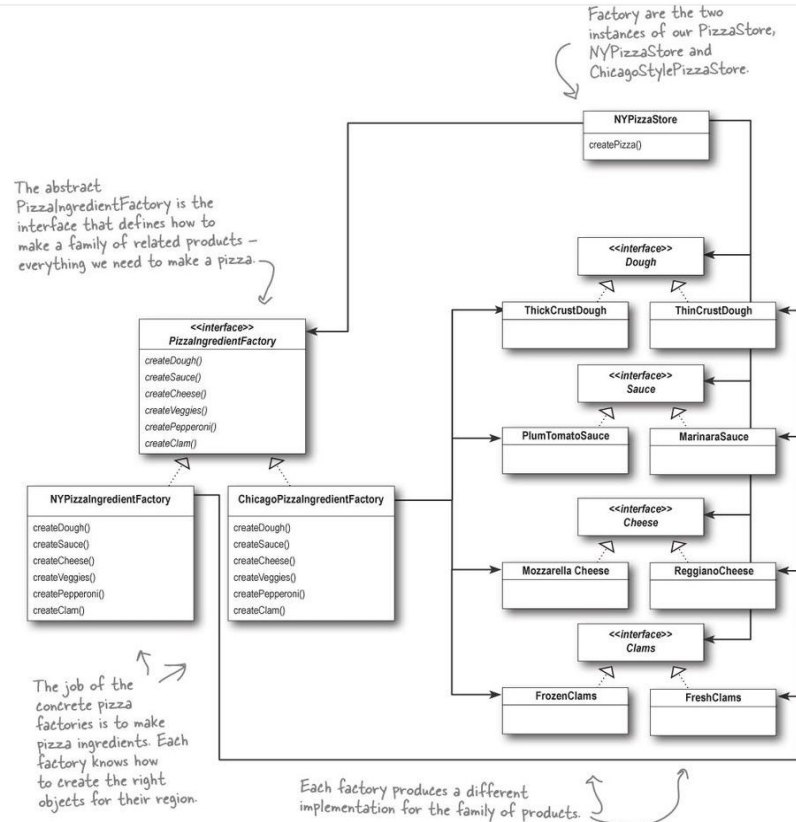
For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.

# Abstract Factory Design Pattern

**Definition:** Abstract Factory Design Pattern Provides an interface *for creating families of related or dependent objects* without specifying their concrete classes



# Abstract Factory Design Pattern - revisiting lecture example



## Difference between Strategy Pattern and Factory Method Pattern

### Strategy

#### *Behavioral Pattern*

Used to perform operations in a particular manner - such as instantiating a certain behaviour at runtime

Encapsulates algorithms

### Factory Method

#### *Creational Pattern*

Used to create objects of a specific type

Encapsulates object creation

# Difference between Factory Method and Abstract Factory

## Factory Method Pattern

It's a method!

The Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

## Abstract Factory Pattern

Abstract factory is an object. Many methods where instantiation happens.

A class delegates the responsibility of object instantiation to another object via composition

# **Design Pattern Practice**

# Practice Question 1

This program comprehension question intends to provide an experience for you to examine the use of design patterns in a real world application. Please download the JHotDraw from this link: <https://sourceforge.net/projects/jhotdraw/files/JHotDraw/JHotDraw%207.6/jhotdraw-7.6.nested.zip/download>

In the package `org.jhotdraw.draw` the class `Figure` uses the Observer design pattern. Answer the following questions:

- a) Mention which method class corresponds to `registerObserver()` method that we have discussed?
- b) Which method class corresponds to `update()`
- c) What is the benefit of this pattern from the perspective of the information hiding principle?

## Answer 1 - part a & b

B. `org.jhotdraw.draw.Figure` also uses the Observer pattern. Look at the code for `org.jhotdraw.draw.Figure`. Please indicate which method class corresponds to the “`registerObserver()`” method in the lecture slides. Please also indicate which method corresponds to the “`update()`” method in lecture slides. (5pts)

**`registerObserver()` : `addFigureListener` – 2 pts**

**`update()` : `changed()`**

or

**from `AbstractFigureListener`**

**`public void figureAreaInvalidated(FigureEvent e)`**

**`public void figureAttributeChanged(FigureEvent e)`**

**`public void figureAdded(FigureEvent e)`**

**`public void figureChanged(FigureEvent e)`**

**`public void figureRemoved(FigureEvent e)`**

**`public void figureRequestRemove(FigureEvent e)`**

## Answer 1 - part c

In the package `org.jhotdraw.draw` investigate the class `Figure` and answer the following questions:

The observer pattern allows for the decoupling of observer objects from the subject object, allowing for the subject object to have zero knowledge of how the observers are implemented. Since the interface of the observer object is fixed, the subject code does not need to be modified in order to extend the type of observers it allows.

## Practice Question 2

This program comprehension question intends to provide an experience for you to examine the use of design patterns in a real world application. Please download the JHotDraw from this link: <https://sourceforge.net/projects/jhotdraw/files/JHotDraw/JHotDraw%207.6/jhotdraw-7.6.nested.zip/download>

In package `org.jhotdraw.draw.connector`, please investigate the classes `ChopRectangleConnector`, `ChopBezierConnector` & `ChopRoundRectangleConnector` and answer the following questions:

- a) What design pattern is being implemented and why?
- b) What are the name of concrete classes that override `ChopRectangleConnector`?
- c) What components would need to be modified if we add a new class `ChopPolygonConnector`
- d) What components would need to be modified if a new method `KarateChop()` was introduced in `ChopRectangleConnector`

## Answer 2

In package `org.jhotdraw.draw.connector`, please investigate the class `ChopRectangleConnector` and answer the following questions:

- a) Strategy design pattern. Because all the classes (`ChopBezierConnector`, etc.) extend properties of `ChopRectangleConnector` to form new algorithms
- b) `ChopBezierConnector`, `ChopDiamondConnector`, `ChopEllipseConnector`, `ChopRoundRectangleConnector`, `ChopTriangleConnector`, `StickyRectangleConnector`
- c) A new class `ChopPolygonConnector` would be created that would simply extend properties of `ChopRectangleConnector` [`ChopDiamond` extends `ChopRectangleConnector`]
- d) All of the classes in b) would need to be changed along with the base class `ChopRectangleConnector`

Team Time