CSC40232: SOFTWARE ENGINEERING

Professor: Jane Cleland-Huang
Lecture 3: Observer Pattern
Wednesday, January 18th
sarec.nd.edu/courses/SE2017

Department of Computer Science and Engineering

If we aren't supposed to program to an implementation – then how can we actually create new things?

Reptile reptile = new Turtle();

2

```
Pizza orderPizza() {
    Pizza pizza = new Pizza();
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

```
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;
}
```

3

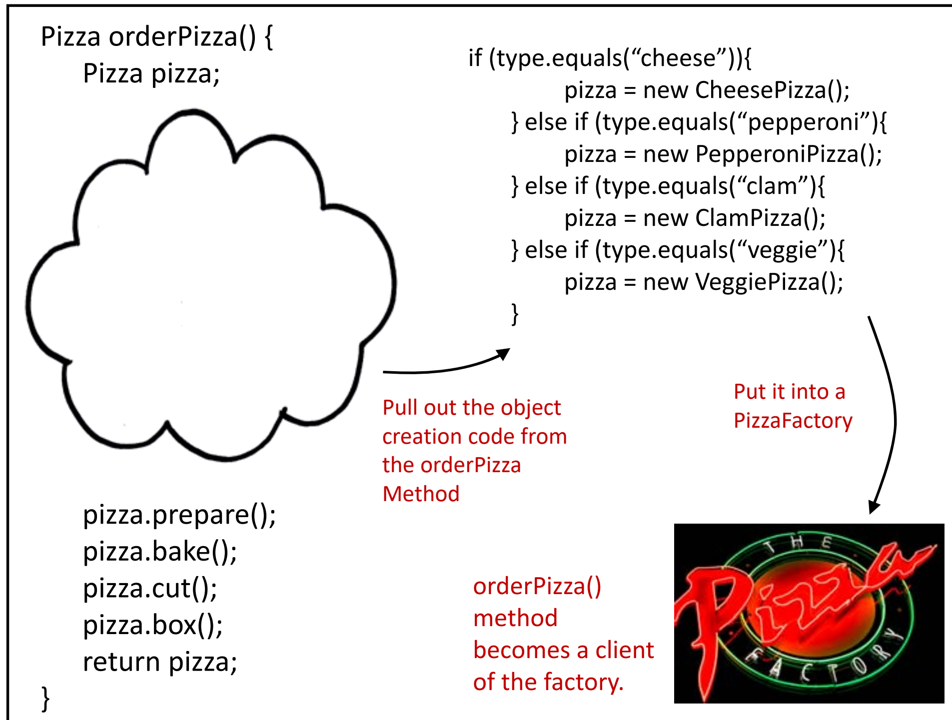
```
Pizza orderPizza() {
    Pizza pizza;
    if (type.equals("cheese")){
        pizza = new CheesePizza();
    } else if (type.equals("greek")){      [struck out]
        pizza = new GreekPizza();          [struck out]
    } else if (type.equals("pepperoni"){
        pizza = new PepperoniPizza();
    } else if (type.equals("clam"){
        pizza = new ClamPizza();
    } else if (type.equals("veggie"){
        pizza = new VeggiePizza();
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

We can expect Pizza types to continually change.

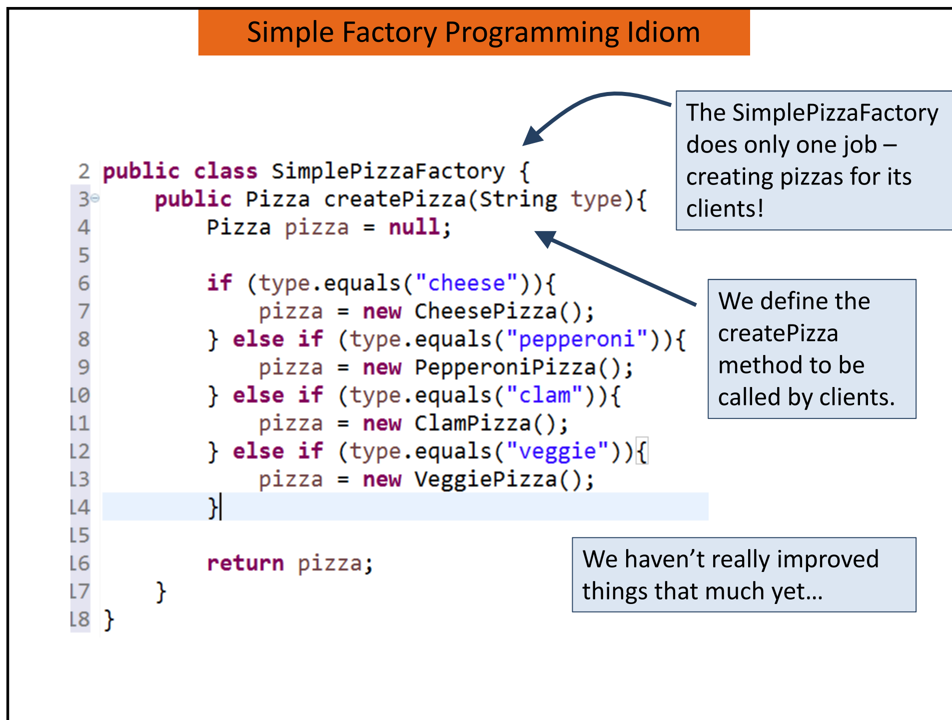
This is probably going to stay pretty much the same.

The problem is dealing with WHICH concrete class needs to be instantiated. This kind of change is breaking OC principle.

Solution: **Separate out and encapsulate the thing that is going to change.**



Pizza orderPizza() {
Pizza pizza;
if (type.equals("cheese")){
pizza = new CheesePizza();
} else if (type.equals("pepperoni"){
pizza = new PepperoniPizza();
} else if (type.equals("clam"){
pizza = new ClamPizza();
} else if (type.equals("veggie"){
pizza = new VeggiePizza();
}
Pull out the object creation code from the orderPizza Method
Put it into a PizzaFactory
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
return pizza;
}
orderPizza() method becomes a client of the factory.
THE Pizza FACTORY



Simple Factory Programming Idiom
2 public class SimplePizzaFactory {
3 public Pizza createPizza(String type){
4 Pizza pizza = null;
6 if (type.equals("cheese")){
7 pizza = new CheesePizza();
8 } else if (type.equals("pepperoni")){
9 pizza = new PepperoniPizza();
10 } else if (type.equals("clam")){
11 pizza = new ClamPizza();
12 } else if (type.equals("veggie")){
13 pizza = new VeggiePizza();
14 }
16 return pizza;
17 }
18 }
The SimplePizzaFactory does only one job – creating pizzas for its clients!
We define the createPizza method to be called by clients.
We haven't really improved things that much yet…

Client Code

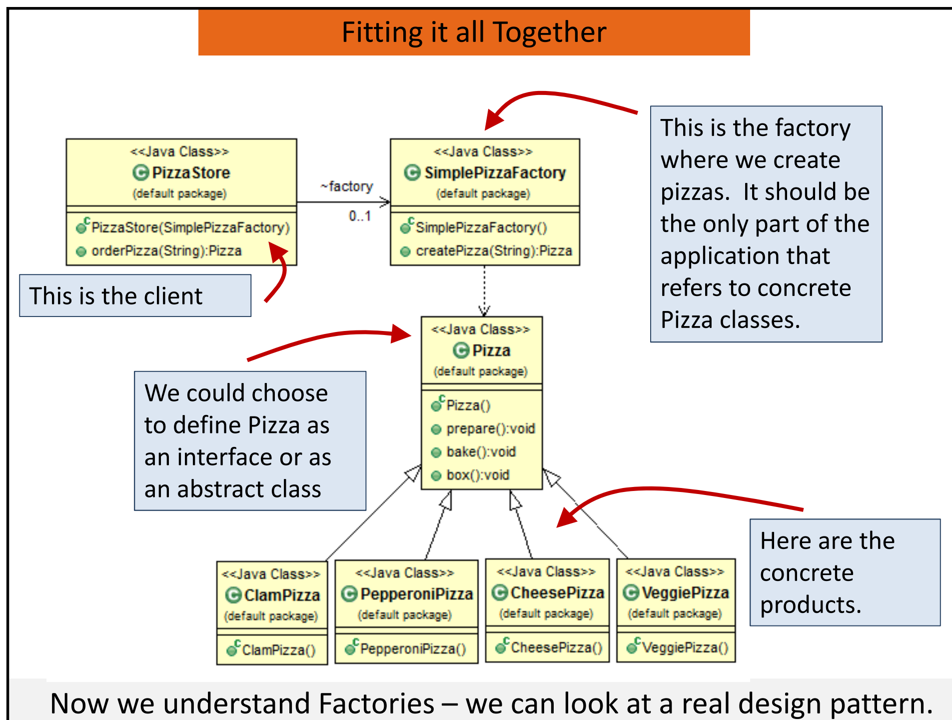
```
 2 public class PizzaStore {
 3     SimplePizzaFactory factory;
 4
 5     public PizzaStore(SimplePizzaFactory factory){
 6         this.factory = factory;
 7     }
 8
 9     public Pizza orderPizza(String type){
10         Pizza pizza;
11         pizza = factory.createPizza(type);
12         pizza.prepare();
13         pizza.bake();
14         pizza.box();
15         return pizza;
16     }
17 }
```

Give PizzaStore a reference to SimplePizzaFactory

PizzaStore gets the factory passed to it in the constructor.

…and orderPizza() method delegates the creation of pizzas to the factory.

The “new” operator has been removed from this class.



Return Factory to the Pizza Store

```
 2 public abstract class PizzaStore {
 3
 4     public PizzaStore(){
 5     }
 6
 7     public Pizza orderPizza(String type){
 8         Pizza pizza;
 9         pizza = createPizza(type);
10         pizza.prepare();
11         pizza.bake();
12         pizza.box();
13         return pizza;
14     }
15
16     abstract Pizza createPizza(String ty
17 }
```

We make PizzaStore abstract

createMethod is moved back into a PizzaStore method rather than a factory object.

The factory method is now abstract in Pizza Store.

❶ orderPizza() is defined in the abstract PizzaStore, not the subclasses. The method therefore has no idea which subclass is actually running the code and making the pizzas.

PizzaStore

createPizza()
orderPizza()

NYStylePizzaStore

ChicagoStylePizzaStore

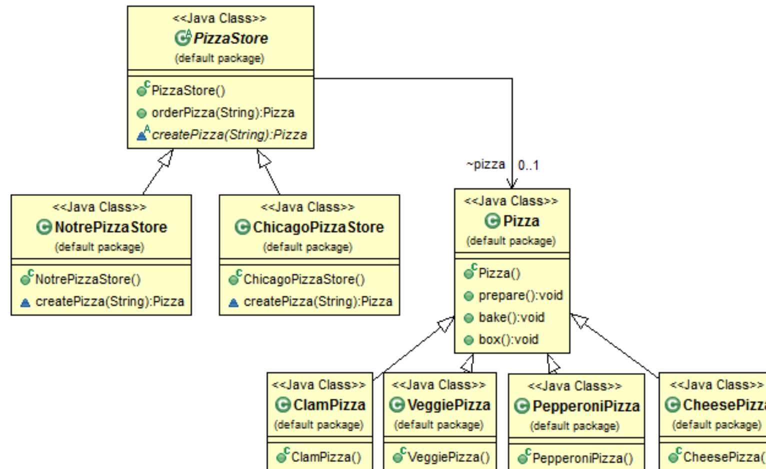
```
pizza= createPizza();
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
```

Which store?

❷ orderPizza() calls createPizza() to actually get a pizza object.

Which kind of pizza does it get? This is NOT up to orderPizza()!!!

All subclasses override createPizza() – but use orderPizza();



abstract Product factoryMethod(String type)

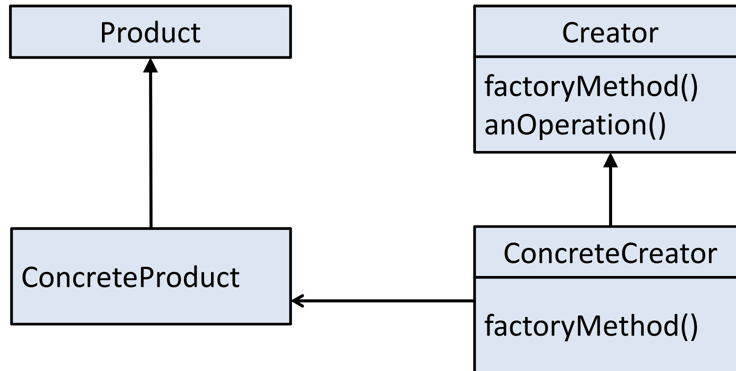
A factory method is abstract so that the subclasses handle object creation.

A factory method may be parameterized (or not) to select among several variations of a product.

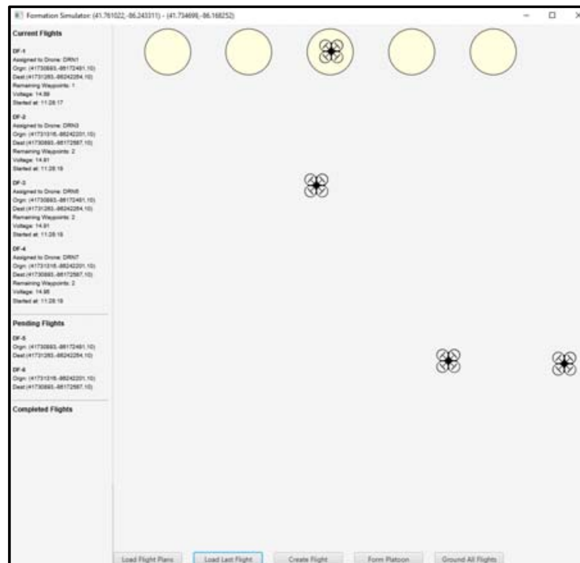
A factory method returns a product that is typically used within methods defined in the superclass.

A factory method isolates the client (i.e. the code in the superclass such as orderPizza()) from knowing what kind of conduct Product is actually created.

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to iinstantiate. Factory Method lets a class defer instantiation to subclasses.



Design Thinking



Dronology is designed to manage the flight of virtual and physical drones.

Drones are assigned unique launch locations (bases). Flight plans are loaded from XML and the drones fly their routes without crashing into each other.

Drones may fly solo, in a platoon, or in more complex formations.

Drones must not crash into each other.

14

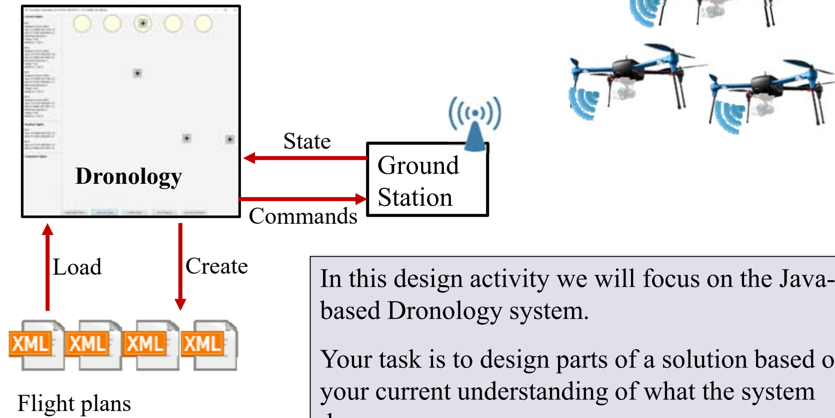
Requirements

ID	Requirement	
R1	A fleet of drones will be generated at simulation startup.	Fleet Management
R2	The system will either be in physical mode or virtual mode but never simultaneously in both.	Runtime Mode
R3	When the system is in physical mode only physical drones will be used.	Runtime Mode
R4	When the system is in virtual mode only virtual drones will be used.	Runtime Mode
R5	The system will not switch modes at runtime.	Runtime Mode
R6	Flight plans will be specified in XML	Flight Plans
R34	Operator will select a flight plan to import	Flight Plans
R7	Each flight plan will include a set of waypoints specified in terms of coordinates.	Flight Plans
R8	Each flight plan will be assigned to an available drone in the fleet.	Flight Plans
R9	Each flight plan will be in one of three states: pending, current, completed.	Flight Plans
R10	Flights will be assigned to drones in the order in which they are received.	Flight Plans
R11	The bounds of the flight zone will be specified in terms of latitude and longitude coordinates.	Flight Zone Management
R12	Drones will only be displayed on central command screens if they are located within flight zone bounds.	Display
R13	A drone shall regularly transmit its current GPS coordinates to central command.	Drone
R14	Central command will track the current coordinates of all drones in flight.	Fleet Management
R15	When a flight plan is assigned to a drone, the drone will fly to the targeted altitude of the starting location.	Drone
R16	During the flight the drone travel from one waypoint to another as ordered by the flight plan.	Drone
R17	Upon reaching the final waypoint the drone shall land.	Drone
R18	A drone will always be in one of five flight modes: grounded, awaitingTakeOff, taking-off, flying, or landing.	Drone
R44	A drone will always be in one of three safety modes: normal, diverted, halted	Safety Management
R19	Minimum separation distance shall always be maintained between all drones.	Safety Management
R20	A safety component shall direct the flight path of any drone which approaches the minimum separation distance of another drone.	Safety Management
R21	When drones take-off from a shared location, their flight shall be staggered to prevent violation of minimum separation distance.	Safety Management
R22	When two or more drones approach minimum separation distance violations will be avoided through forming an aerial roundabout.	Safety Management
R23	Drones must maintain sufficient voltage to return to base.	Drone
R24	Drones shall ascend vertically during takeoff until the targeted altitude has been reached.	Drone
R25	The maximum number of drones flying in the flight zone shall be limited to two.	Flight Zone Management
R26	While the system is in virtual model, a flight simulator will compute the current location of each drone in flight.	Drone

Currently 53 requirements implemented

16

High Level Architecture



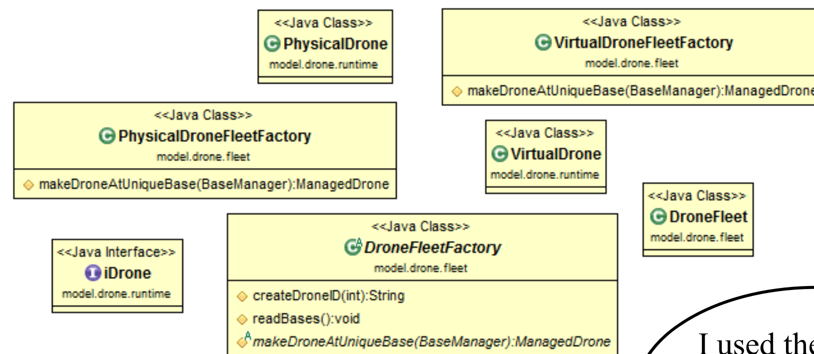
In this design activity we will focus on the Java-based Dronology system.

Your task is to design parts of a solution based on your current understanding of what the system does.

For this exercise your task is to think of ways to include some of the Design Patterns we've learned in this course so far.

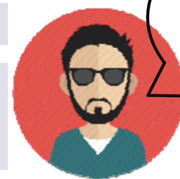
17

Example: Factory Method Pattern



These are some of the classes in the design.

Work in groups of 2-3 to sketch out the UML class diagram that organizes these classes into a factory method pattern.

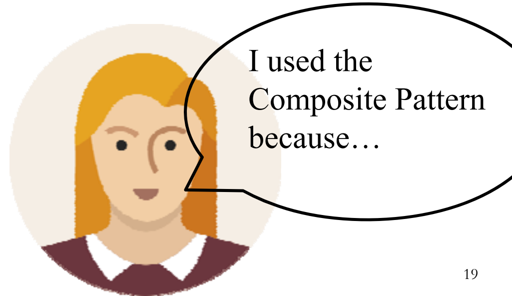


I used the Factory Method Pattern because…

18

Composite Pattern:

Decide where you might use the Composite Pattern in the Dronology System.
Sketch out a UML diagram showing its possible use.



19

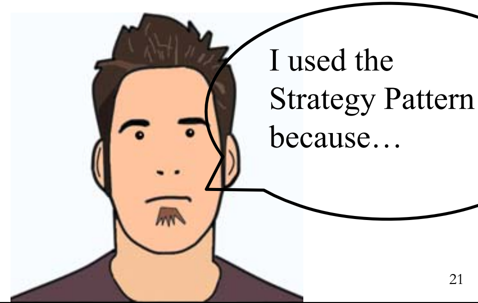
Observer Pattern:

Decide where you might use the Observer Pattern in the Dronology System.
Sketch out a UML diagram showing its possible use.

20

Strategy Pattern:

Decide where you might use the Strategy Pattern in the Dronology System. Sketch out a UML diagram showing its possible use.

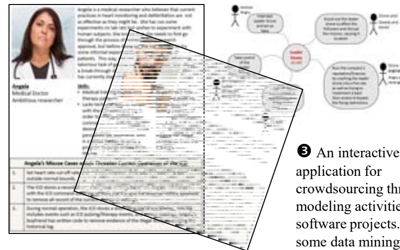


21

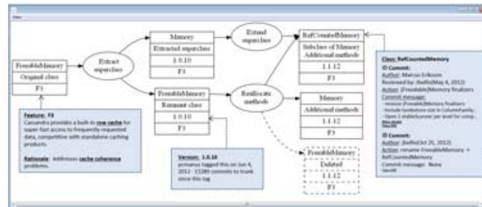
Team Projects

❶ An eclipse-based tool for supporting the creation of Safety Assurance Cases.

❷ An eclipse plugin for supporting impact analysis in Safety Critical Systems. Includes visualization.



❸ An interactive, GUI based application for crowdsourcing threat modeling activities for software projects. (Requires some data mining)



❹ A utility for interactively visualizing the evolution of requirements and source code. Sits on top of a Github repository

3-4 people per team.
Max three teams per project.