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

Reptile reptile = new Turtle();
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;
}

We can expect Pizza types to continually change.

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.

This is probably going to stay pretty much the same.
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();
    }
    pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
return pizza;
}

Pull out the object creation code from the orderPizza Method
Put it into a PizzaFactory

orderPizza() method becomes a client of the factory.

Simple Factory Programming Idiom

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("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}

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

```java
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.box();
        return pizza;
    }
}
```

Fitting it all Together

This is the factory where we create pizzas. It should be the only part of the application that refers to concrete Pizza classes.

We could choose to define Pizza as an interface or as an abstract class.

Here are the concrete products.

Now we understand Factories – we can look at a real design pattern.
We make PizzaStore abstract.

The factory method is now abstract in PizzaStore.

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.

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()`;

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 instantiate. Factory
Method lets a class defer instantiation to subclasses.

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.
## Requirements

Currently 53 requirements implemented

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>A drone shall maintain sufficient voltage to return to base.</td>
<td>Safety Management</td>
</tr>
<tr>
<td>12</td>
<td>A drone shall monitor vertical velocity during takeoff and the targeted altitude has been reached.</td>
<td>Drone</td>
</tr>
<tr>
<td>13</td>
<td>The maximum number of drones flying in the flight zone shall be limited to two.</td>
<td>Flight Zone Management</td>
</tr>
<tr>
<td>14</td>
<td>While the system is in virtual mode, a flight simulator will compute the current location of each drone in flight.</td>
<td>Drone</td>
</tr>
</tbody>
</table>

Currently 53 requirements implemented
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.

**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...
Composite Pattern:

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

I used the Composite Pattern because…

Observer Pattern:

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

I used the Observer Pattern because…
Strategy Pattern:

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

I used the Strategy Pattern because…

Team Projects

1. An eclipse-based tool for supporting the creation of Safety Assurance Cases.
2. An interactive, GUI based application for crowdsourcing threat modeling activities for software projects. (Requires some data mining)
3. 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.