ECE450 – Software Engineering II

Today: **Design Issues**

What's wrong with this?

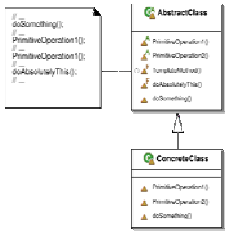
```
public class PizzaMaker {
  public void cookPizzas(List pizzas) {
    for (int i=0; i<pizzas.size(); ++i) {
      Object pizza = pizzas.get(i);
      if (pizza instanceof ThinCrustPizza) {
        ((ThinCrustPizza)pizza).cookInWoodFireOven();
      }
      else if (pizza instanceof PanPizza) {
        ((PanPizza)pizza).cookInGreasyPan();
      }
      else {
        // OH NO!  What is this thing?
      }
    }
  }
}
```


The Open-Closed Principle

- *Classes should be open for extension, but closed for modification*
  - You should be able to extend a system without modifying the existing code
- The type-switch in the example violates this
  - Have to edit the code every time the marketing department comes up with a new kind of pizza

Abstraction is the solution

- Solve the problem by creating a *Pizza* interface with a cook method
  - Or an abstract base class whose cook method must be overridden by every child
- The *Template Method* design pattern is used to set up the skeleton of an algorithm
  - Details then filled in by concrete subclasses



Cooking a generic pizza

```
public abstract class Pizza {
  public final void cook() {
    placeOnCookingSurface();
    placeInCookingDevice();
    int cookTime = getCookTime();
    letItCook(cookTime);
    removeFromCookingDevice();
  }
  protected abstract void placeOnCookingSurface();
  protected abstract void placeInCookingDevice();
  protected abstract int  getCookTime();
  protected abstract void letItCook(int min);
  protected abstract void removeFromCookingDevice();
}
```


Is this general enough?

- But what if someone wants to do something you didn't anticipate?
  - E.g. wants to add a PancakePizza that has to be flipped over halfway through the cooking process
- What are the options?

Override the Template Method?

```
public final void cook() {
  placeOnCookingSurface();
  placeInCookingDevice();
  int cookTime = getCookTime();
  letItCook(cookTime/2);
  flip();
  letItCook(cookTime/2);
  removeFromCookingDevice();
}
```

- But cook was final
- And it's storing up trouble for the future

Squeeze it somewhere else?

```
protected void removeFromCookingDevice() {
  flip();
  letItCook(cookTime);
  …remove from skillet…
}
```

- removeFromCookingDevice shouldn't be doing other things
  - Think about the documentation
- Once again, we're storing up trouble for the future

Leave space for future growth?

```
public final void cook() {
    beforePlacingOnCookingSurface();
    placeOnCookingSurface();
    beforePlacingInCookingDevice();
    placeInCookingDevice();
    beforeCooking();
    for (int i=0; i<getCookingPhases(); i++) {
      letItCook(getCookTime(i));
      afterCookingPhase(i);
    }
    beforeRemovingFromCookingDevice();
    removeFromCookingDevice();
    afterRemovingFromCookingDevice();
  }
```


And the answer is...

- Plan for *reasonable* future growth
  - "Reasonable" means "guided by your experience, and the experience of others"
- Note: compliance with the Open-Closed principle is a matter of judgment
  - No algorithm or code analyzer can make a definitive ruling

Liskov substitution principle

- *Anywhere you specify a base type, you should be able to use an instance of a derived type*
  - This one *is* checkable, if you provide the system with enough information
- Polymorphism that obeys the rules of Design by Contract
  - Allowed to weaken preconditions and strengthen postconditions, but not viceversa

Is Vegan Pizza really pizza?

```
public interface Pizza {
  public Cheese getCheese();
}

public class SimplePizza implements Pizza {
  public Cheese getCheese() {
    return new Mozzarella();
  }
}

public class VeganPizza implements Pizza {
  public Cheese getCheese() {
    return null;
  }
}
```

But then...

```
class Customer {
 public boolean decideToBuy(Pizza p) {
   Cheese c = p.getCheese();
   return c.smellsGood();
}
```

- Returning `null` violates the contract
- The problem is, that contract was *implicit*
  - Which is why languages like Ada allow programmers to make contracts explicit

This is **not** a solution!

```
class Customer {
  public boolean decideToBuy(Pizza p) {
    Cheese c = p.getCheese();
    if (c != null) {
      return c.smellsGood();
    }
    else {
      return false;
    }
}
```

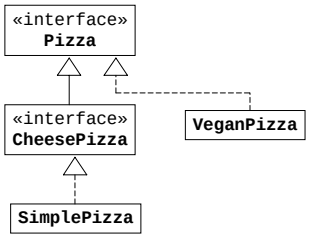
- Why is this bad?

Possible solutions

- Weaken or remove the supertype contract
  - Only if you have a strategy for updating existing code
  - Remember, the contract was there for a reason...
- Remove or replace the operation
  - Add a `smellsGood` method to `pizza`

Possible solutions (cont)

- Modify the inheritance hierarchy
  - Only works if the code that calls `getCheese` knows if it has a real pizza or a vegan pizza



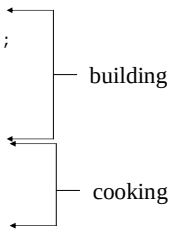
Possible solutions (cont)

- Modify subtype behaviour
  - In this case, use the *Null Object Pattern* to return an instance of `NullCheese`
    - *Null Object Pattern*: Having a non-null object whose job is to take the place of null, and whose methods all return 0, null, not-interesting, empty string, etc.
    - A bit of a hack
- Opinions?

It's not my job

- Single Responsibility Principle: *A class or interface should only be responsible for one thing*
  - Alternative phrasing: *A class or interface should have only one reason to change*
- Count responsibilities:

```
public interface Pizza {
  public List<Topping> getToppings();
  public void setToppings(List<Topping> tops);
  public PizzaSize getSize();
  public void setSize(PizzaSize size);
  public PizzaCrust getCrust();
  public void setCrust(PizzaCrust crust);
  pub void cook(int temp, int minutes);
  public TasteRating rateTaste();
  public SmellRating rateSmell();
  public boolean isBurnt();
}
```



Law of Demeter

- A method M of an object O may only invoke methods of:
  - Itself
  - Its parameters
  - Objects it creates
  - Its members
- In particular, methods should *not* invoke methods of other objects' members
- A violation:

```
public boolean buyPizza(Pizza p) {
  return p.getCheese().smellsGood();
}
```

- This method depends on Pizza having a Cheese member, and Cheese having a smellsGood method
  - All these things might change.
  - The more indirect connections the code has, the harder it is to understand, test, and change