COMP 504: Graduate Object-Oriented Programming and Design

Lecture 15: Visitor Design Pattern

Mack Joyner (mjoyner@rice.edu)

https://www.clear.rice.edu/comp504

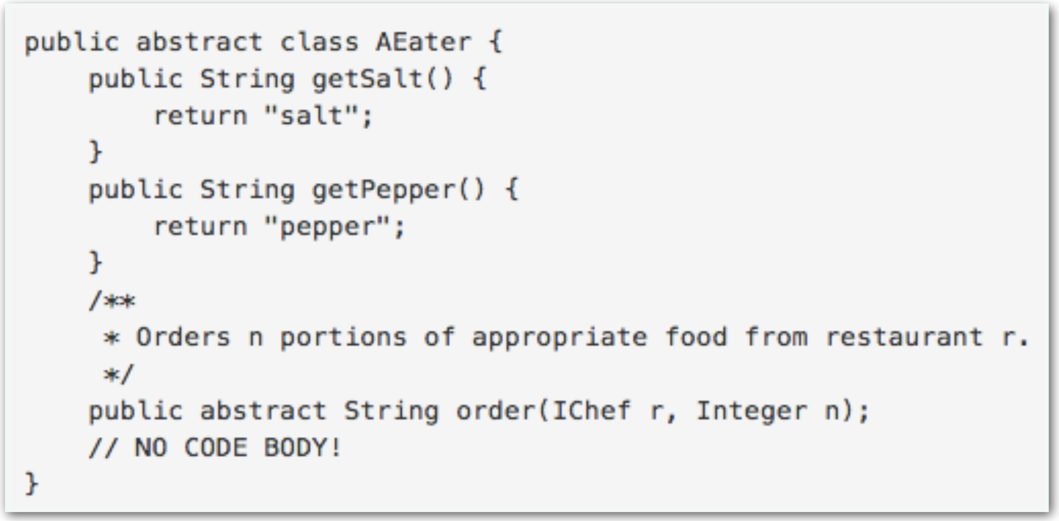
Announcements & Reminders

HW #3 due Wednesday, Oct 7th at 11:59pm

Food Consumers

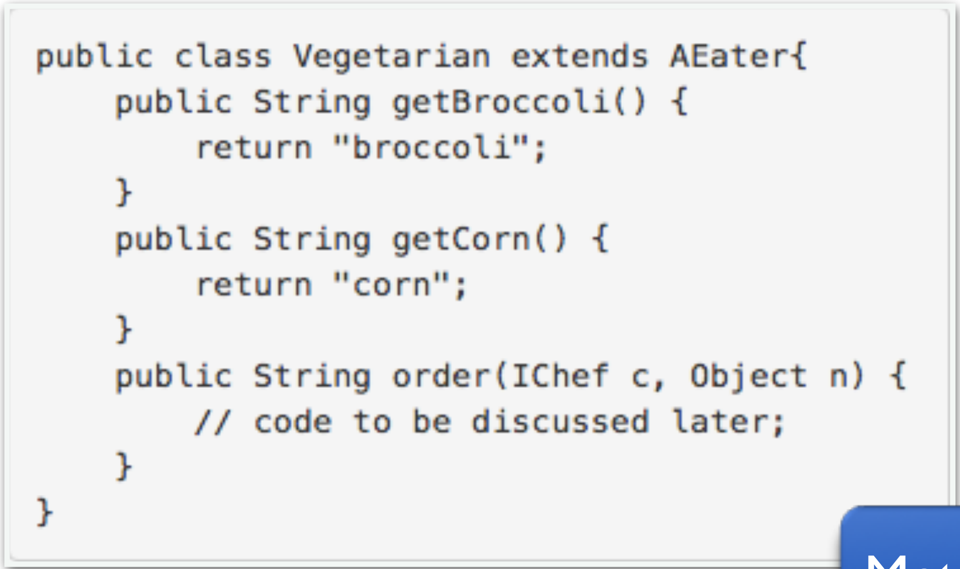
- Assume there are 2 types of food consumers
  - Vegetarians
  - Carnivores
- Vegetarians and carnivores cannot cook, they ask a chef to cook for them
  - Supply ingredients to chef and *order* cooked meal
- Model them as concrete subclasses of an abstract class *AEater*
  - *AEater* has 2 concrete methods *getSalt*, *getPepper*
  - *AEater* has 1 abstract method called *order*

Abstract Class AEater

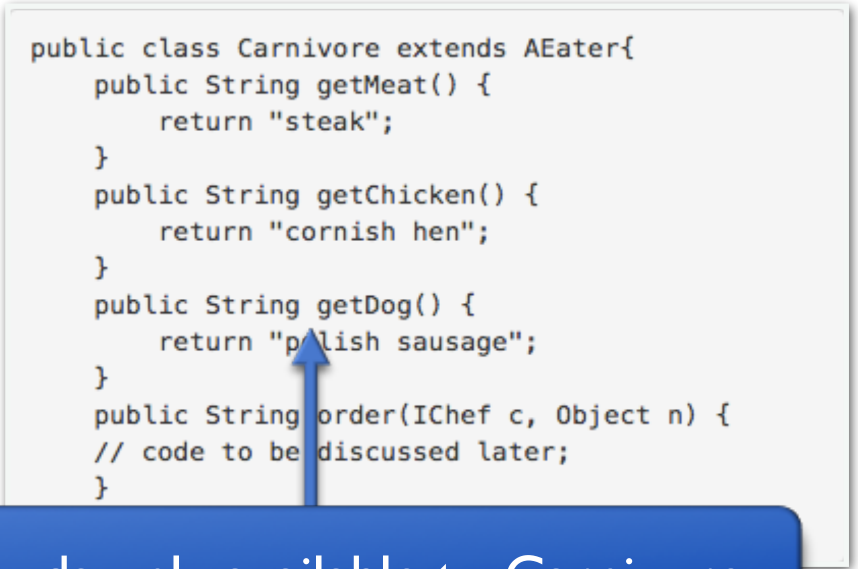


```
public abstract class AEater {
    public String getSalt() {
        return "salt";
    }
    public String getPepper() {
        return "pepper";
    }
    /**
     * Orders n portions of appropriate food from restaurant r.
     */
    public abstract String order(IChef r, Integer n);
    // NO CODE BODY!
}
```

Concrete Subclasses of AEater



```java
public class Vegetarian extends AEater{
    public String getBroccoli() {
        return "broccoli";
    }
    public String getCorn() {
        return "corn";
    }
    public String order(IChef c, Object n) {
        // code to be discussed later;
    }
}
```



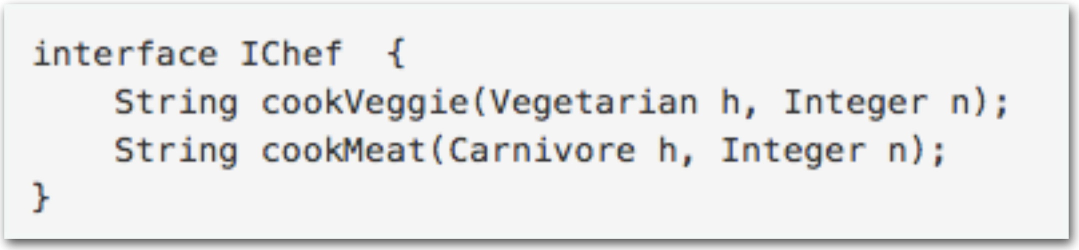
```java
public class Carnivore extends AEater{
    public String getMeat() {
        return "steak";
    }
    public String getChicken() {
        return "cornish hen";
    }
    public String getDog() {
        return "polish sausage";
    }
    public String order(IChef c, Object n) {
    // code to be discussed later;
    }
```

Methods only available to Carnivore

The Chef

The chef is an interface (IChef) with 2 methods

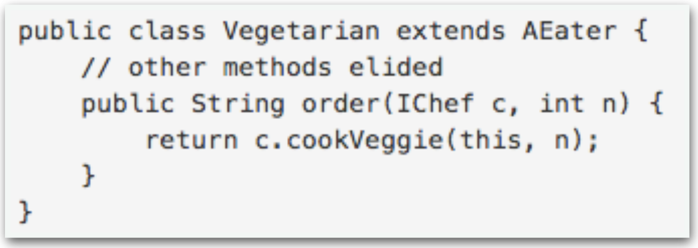
- cookVeggie for veggie dish
- cookMeat for meat dish



```
interface IChef  {
    String cookVeggie(Vegetarian h, Integer n);
    String cookMeat(Carnivore h, Integer n);
}
```

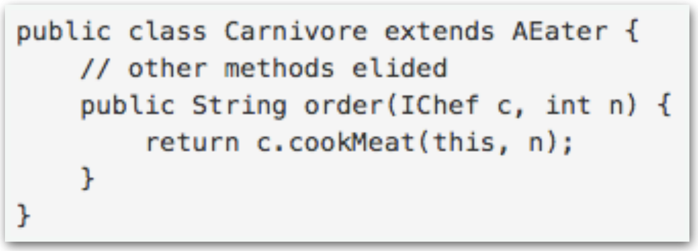
Ordering from IChef

- To order from an Chef
  - Vegetarian calls *cookVeggie()* passing itself as a parameter
  - Carnivore calls *cookMeat()* passing itself as a parameter



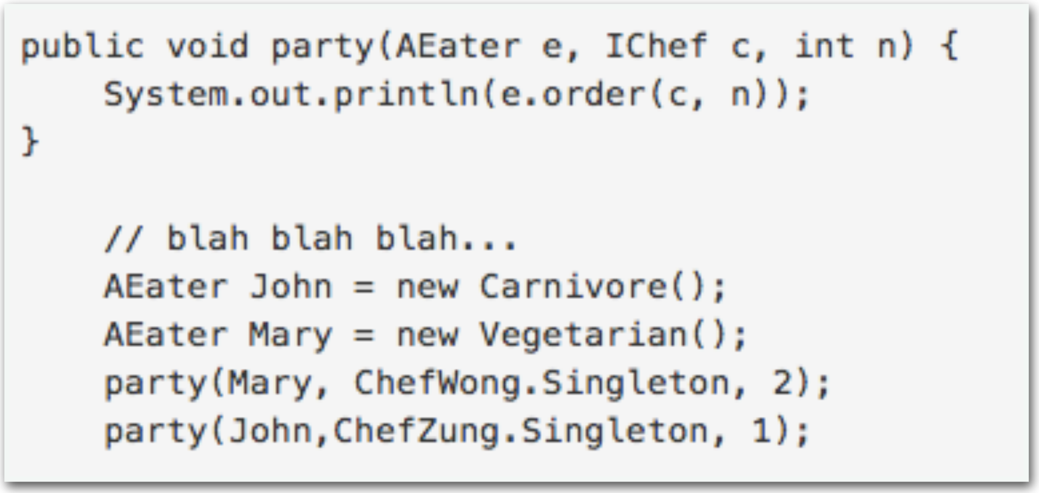
```
public class Vegetarian extends AEater {
    // other methods elided
    public String order(IChef c, int n) {
        return c.cookVeggie(this, n);
    }
}
```

- Concrete vegetarian, carnivore classes only deal with IChef
  - Don't care about IChef concrete classes
  - Polymorphism guarantees correct concrete IChef method call



```
public class Carnivore extends AEater {
    // other methods elided
    public String order(IChef c, int n) {
        return c.cookMeat(this, n);
    }
}
```

Client Code



```
public void party(AEater e, IChef c, int n) {
    System.out.println(e.order(c, n));
}

    // blah blah blah...
    AEater John = new Carnivore();
    AEater Mary = new Vegetarian();
    party(Mary, ChefWong.Singleton, 2);
    party(John,ChefZung.Singleton, 1);
```

Hosts

- This food consumers and chefs is an example of the visitor design pattern

- The abstract class *AEater* and concrete subclasses are called *hosts*

- The *order* method is called the hook method

- Concrete *AEater* subclasses know to call appropriate method on *IChef* parameter
    - Don’t need to know how concrete *IChef* performs task
    - Multiple ways to cook appropriate type of food

Visitors

- The chef interface (*IChef*) and all concrete implementations are called *visitors*

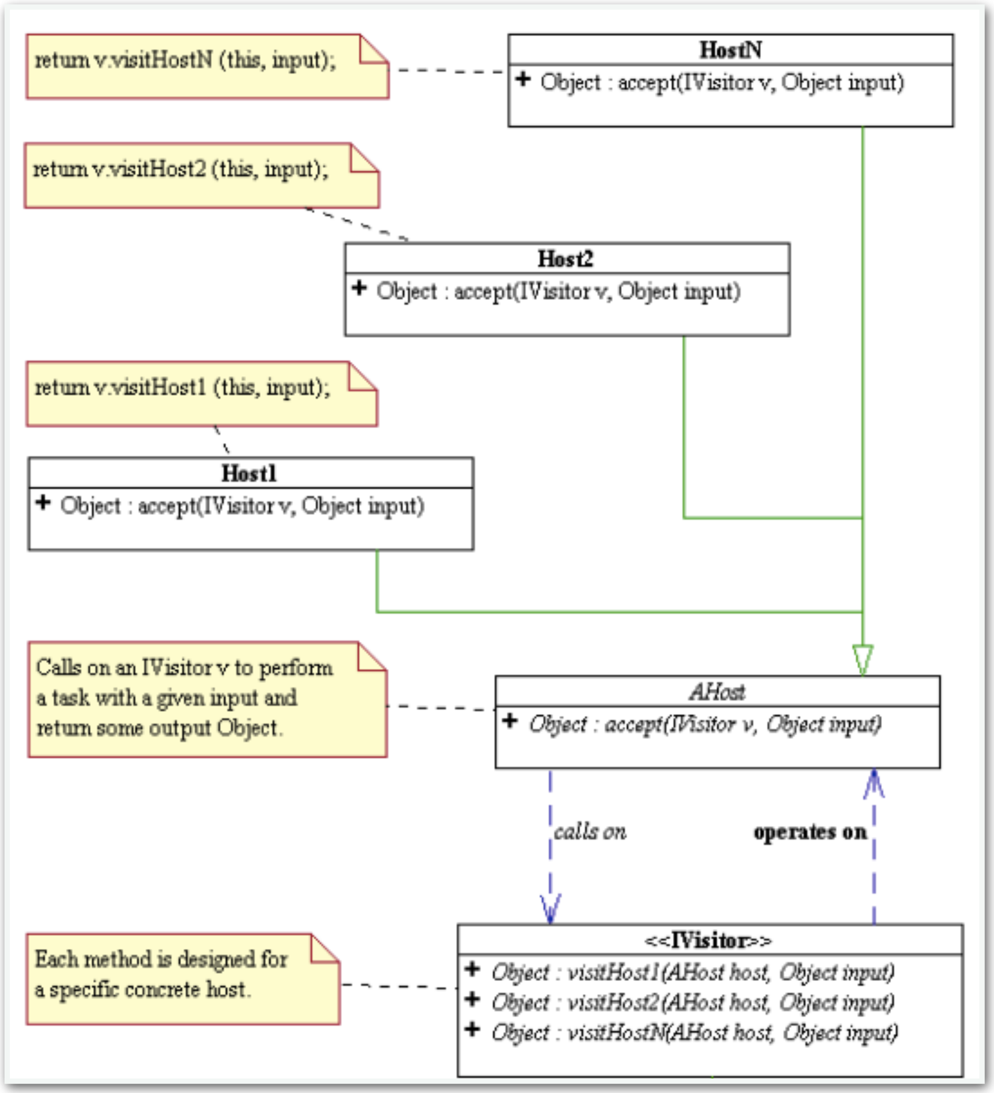
- *IChef* knows it's host is a Vegetarian or a Carnivore when performing *cookVeggie/cookMeat*
  - Can only call Vegetarian or Carnivore methods
  - Type checking flags an error if *getBroccoli* called in *cookMeat*

- Interactions with *hosts* (*AEater* and concrete subclasses) and *visitors* (IChef and concrete subclasses) are robust

Visitor Design Pattern

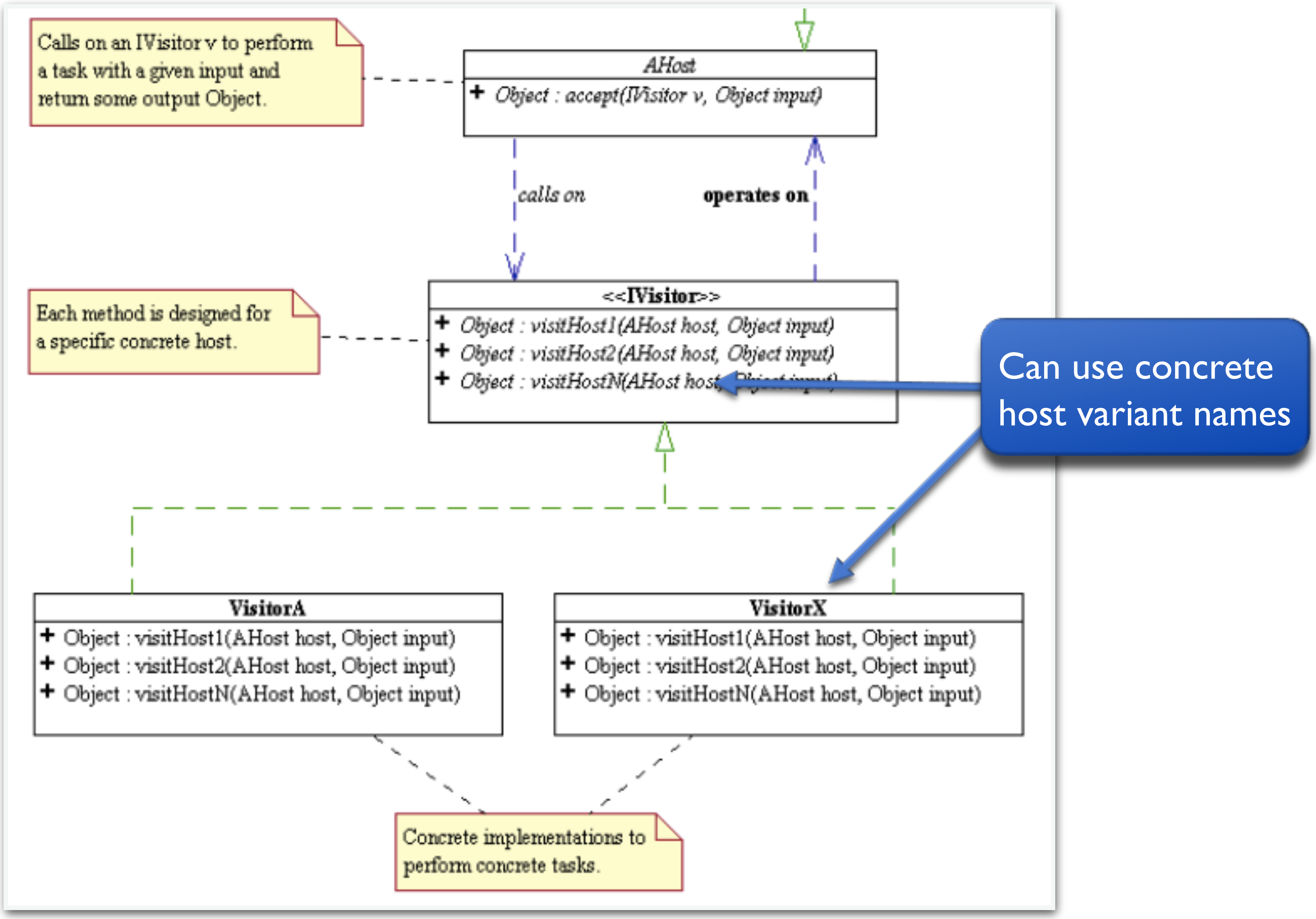
- A pattern of communication and collaboration between two unions patterns (hosts and visitors)
- Visitor interface has one method for each concrete host variant
- Abstract host has a method called the hook to accept the visitor
  - Concrete variant calls correct visitor method
- Only works with stable set of concrete host variants
  - Adding a variant to host requires changes to all visitors

UML Diagram (host)



return v.visitHostN (this, input);
HostN
+ Object : accept(IVisitor v, Object input)
return v.visitHost2 (this, input);
Host2
+ Object : accept(IVisitor v, Object input)
return v.visitHost1 (this, input);
Host1
+ Object : accept(IVisitor v, Object input)
Calls on an IVisitor v to perform a task with a given input and return some output Object.
AHost
+ Object : accept(IVisitor v, Object input)
calls on
operates on
Each method is designed for a specific concrete host.
<<IVisitor>>
+ Object : visitHost1(AHost host, Object input)
+ Object : visitHost2(AHost host, Object input)
+ Object : visitHostN(AHost host, Object input)

UML Diagram (visitor)


Calls on an IVisitor v to perform a task with a given input and return some output Object.
AHost
+ Object : accept(IVisitor v, Object input)
calls on
operates on
Each method is designed for a specific concrete host.
<<IVisitor>>
+ Object : visitHost1(AHost host, Object input)
+ Object : visitHost2(AHost host, Object input)
+ Object : visitHostN(AHost host, Object input)
Can use concrete host variant names
VisitorA
+ Object : visitHost1(AHost host, Object input)
+ Object : visitHost2(AHost host, Object input)
+ Object : visitHostN(AHost host, Object input)
VisitorX
+ Object : visitHost1(AHost host, Object input)
+ Object : visitHost2(AHost host, Object input)
+ Object : visitHostN(AHost host, Object input)
Concrete implementations to perform concrete tasks.

Worksheet #10: Visitor Design Pattern

Could you use the Visitor design pattern if you had a hw with commands, strategies and paint objects (moving circles, non-moving squares)? Explain why or why not?