Lecture Notes
December 3, 2020

Reminder “Assignment T5: Second Iteration” https://courseworks2.columbia.edu/courses/104335/assignments/490853, originally due this Friday Dec 4, extended to Sunday Dec 6

Define the equivalence partitions, both valid and invalid, for each of your major methods and devise corresponding unit tests for those methods.

For those equivalence partitions that have boundaries (not all do), determine what specifically are those boundaries and devise unit tests for exactly at the boundary, one less and one more, for each boundary.

Set up your branch coverage tool before running your test suite, since many branches will be covered while running regular blackbox tests. If you do not already achieve 100% branch coverage, try to devise new test cases that will cover missed branches.

Continuous integration - include configuration file(s) in your github repo and submit something as part of your iteration report to demonstrate that the CI works

Working as a team: improving your code

Static analysis tools and human code review can detect code smells (discussed on October 20)

Refactoring is a systematic approach to removing code smells = Modify code structure in very small steps that do not change system functionality (semantics-preserving)

Refactoring patterns are recipes for refactoring specific code smells. See https://sourcemaking.com/refactoring

Refactoring should not be conducted as part of the same commit as bug-fixes. Why not?

Refactoring tools detect code smells and suggest code changes that instantiate the refactoring patterns - or even automate the edits, often implemented as an IDE plugin

Need to re-run affected test cases (or entire test suite) after every small change to verify that there were no functional changes. System-level tests should all produce the same result (pass or fail) with pre-refactored and post-refactored versions of the system

But if any of the refactoring changes cross unit boundaries, or introduce new units/remove old units, then we cannot expect the exact same unit tests to pass - need to also “refactor” the test suite

Let’s say we have

```
veryLongMethod () {
    many lines of code
}
```

and refactor to

```
veryLongMethod (....) {
    shorterMethod(...;
    if ( someCondition(...) ) {
        anotherShorterMethod(..., aMethod(...), ...);
    }
}
```

Which test cases should be changed, added or removed?

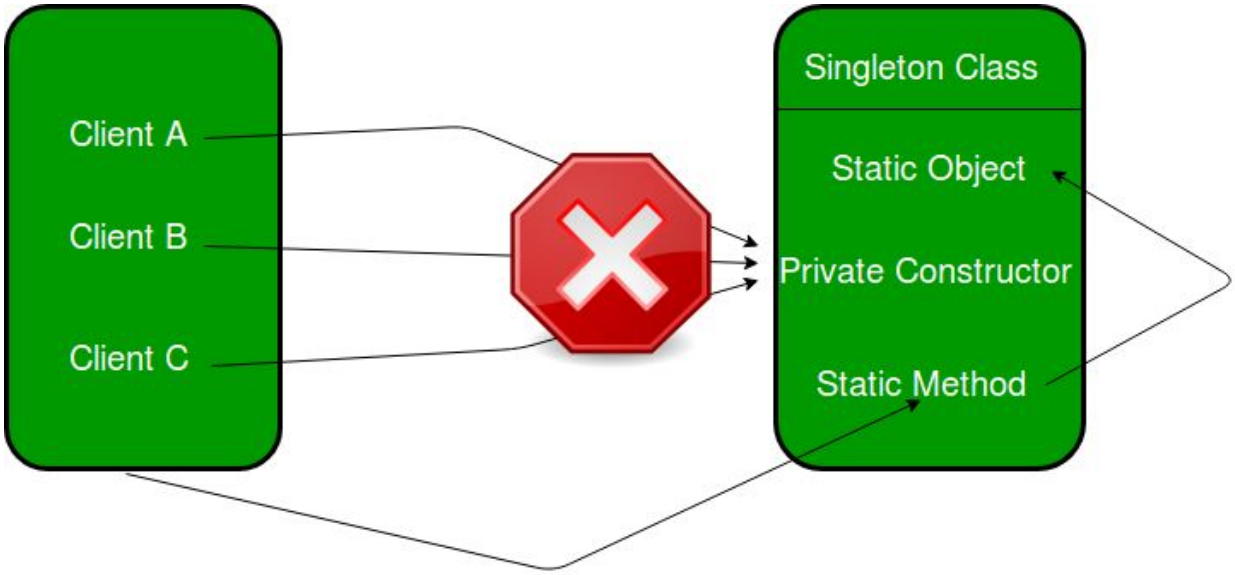
One common application of refactoring is to leverage design patterns.

"Design patterns" are at a much finer granularity than architectural styles like client/server, instead operating at the class and method level. Defined as a reusable solution or template, but not (usually) a library or framework where you can reuse code

Example design pattern: Singleton, when we want to create exactly one (or at most one) instance of a class

Useful for logger, hardware driver, cache, thread pool, database connection, configuration file, java.lang.Runtime

Why would we want no more than one of these?




```
// Classical Java implementation of singleton
class Singleton
{
    private static Singleton obj;

    // private constructor to force use of
    // getInstance() to create Singleton object
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

Three categories of design patterns:

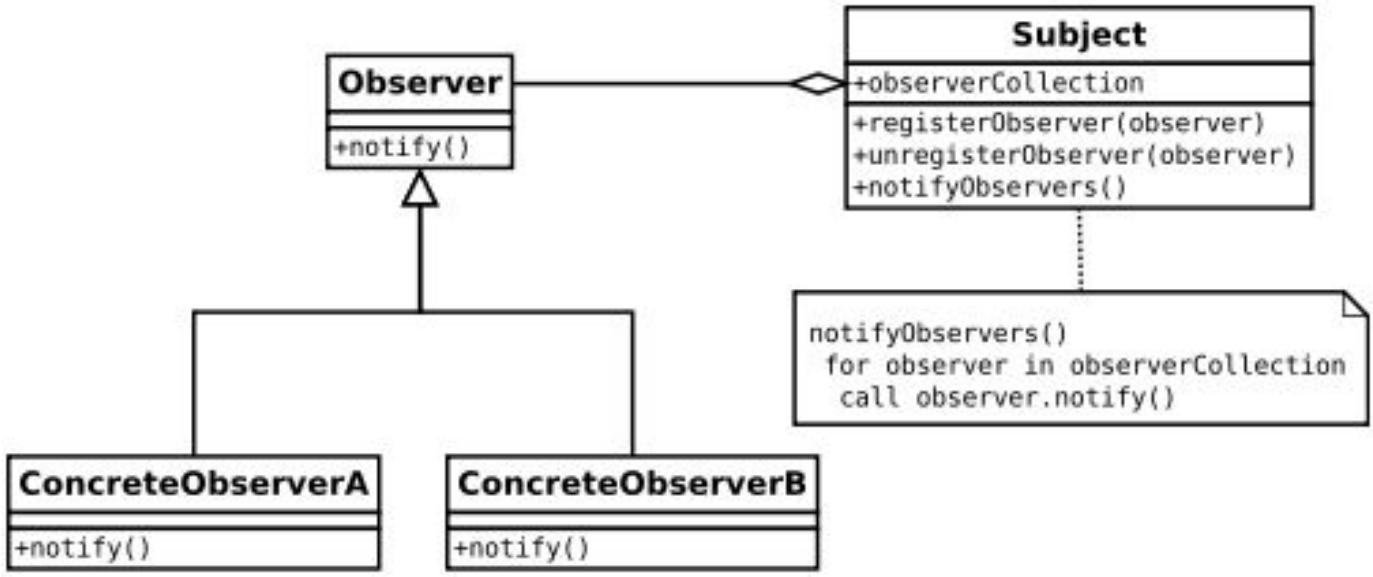
1. Creational design patterns: class instantiation or object creation

2. Structural design patterns: organize multiple classes and objects to form larger structures

3. Behavioral design patterns: realize common communication patterns among objects

Which group does Singleton belong to?

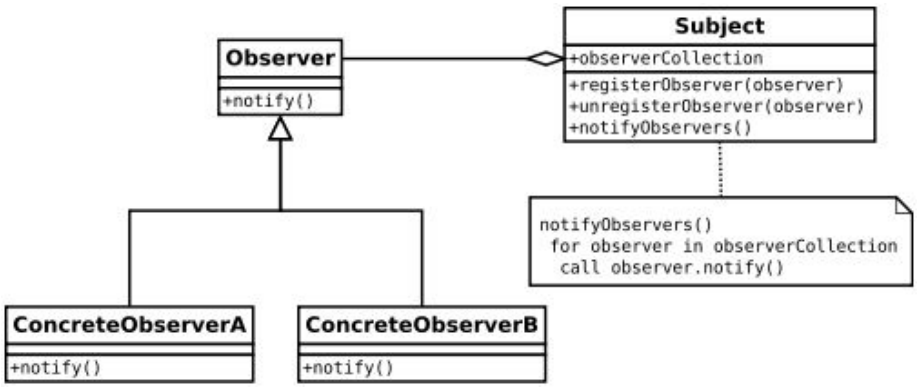
Observer pattern (behavioral) - defines a one to many dependency between objects so that when the one *subject* object changes state, all its dependent *observer* objects are notified (via callbacks)



This graphical notation, called a “class diagram”, is an example of UML (Unified Modeling Language). I do not plan to cover UML this semester, but if you are interested there is a nice introduction here.

Info from student (thank you!):

There's a good drawing program for UML at LucidChart: https://www.lucidchart.com/pages/
Is an online software that you can use. When you start a blank project, you will not have any UML shapes or connections available. You have to go to the bottom of the left-hand panel, there is a button that allow you to add shapes. Then, there is a popup menu. One of the first entries should be UML. If not, then you can just search for it. Then, you go from there.



Observer and **Subject** are interfaces or abstract classes. Concrete observer classes need to implement the **Observer** interface - the `notify()` callback. Then they call `registerObserver()` (and, potentially, `unregisterObserver()`) for specific subject objects

Subject objects keep a collection referencing all their observers. Classes that implement the **Subject** interface (and do other application-specific things) can change without affecting any Observer classes and vice versa

Example: GUI event listeners (onClick)

```
abstract class Observer {
    protected Subject subject;
    public abstract void notify();
}

class Subject {
    private List<Observer> observers = new
ArrayList<>();
    private int state;

    public void add(Observer o) {
        observers.register(o);
    }

    public int getState() {
        return state;
    }

    public void setState(int value) {
        this.state = value;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.notify();
        }
    }
}
```

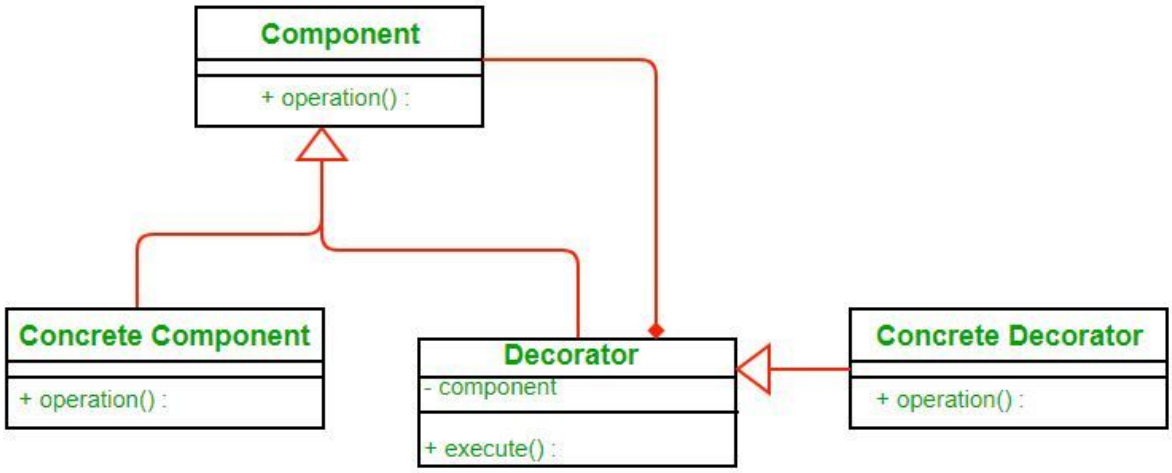
```
class HexObserver extends Observer {
    public HexObserver(Subject subject) {
        this.subject = subject;
        this.subject.register(this);
    }

    public void update() {
        System.out.print(" " +
Integer.toHexString(subject.getState()));
    }
}

class OctObserver extends Observer {
    public OctObserver(Subject subject) {
        this.subject = subject;
        this.subject.register( this );
    }

    public void update() {
        System.out.print(" " +
Integer.toOctalString(subject.getState()));
    }
}

class BinObserver extends Observer {
    public BinObserver(Subject subject) {
        this.subject = subject;
        this.subject.register(this);
    }

    public void update() {
        System.out.print(" " +
Integer.toBinaryString(subject.getState()));
    }
}
```

```
public class ObserverDemo {
    public static void main( String[] args ) {
        Subject sub = new Subject();
        // Client configures the number and type of
Observers
        new HexObserver(sub);
        new OctObserver(sub);
        new BinObserver(sub);
        Scanner scan = new Scanner(System.in);
        for (int i = 0; i < 5; i++) {
            System.out.print("\nEnter a number: ");
            sub.setState(scan.nextInt());
        }
    }
}
```

Enter a number: 55
37 67 110111
Enter a number: 12
c 14 1100
Enter a number: -10
fffffff6 37777777766
11111111111111111111111111110110
Enter a number: 112
70 160 1110000
Enter a number: 5
5 5 101

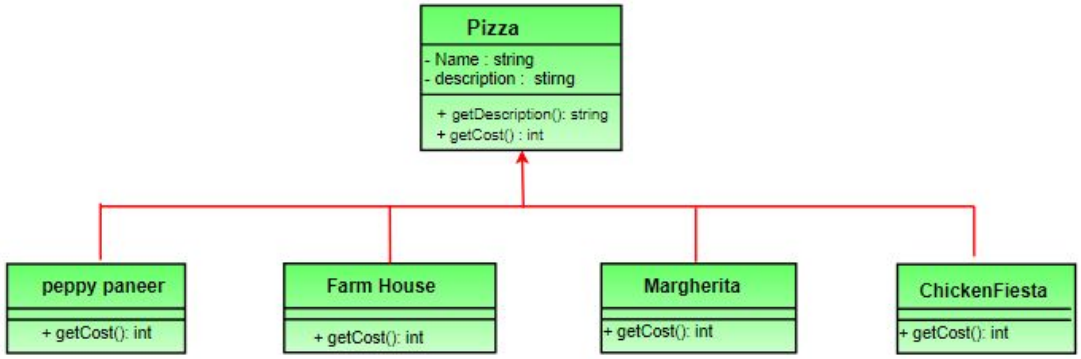
Decorator pattern (structural) - Attach additional responsibilities to an object dynamically (at runtime) via wrapping and delegation rather than statically via inheritance. Useful to extend the behavior of some, but not all, objects of a class

Decorator objects have the same parent type as the objects to be decorated, so code can pass around a decorated object instead of the original object (before it was decorated). The decorator has an instance variable referencing the original. Decorators can stack (wrapping a wrapper)



Example: drawing program with shapes, such as circles and rectangles, where we want to add fill-color, line-color, line-thickness, etc. to some but not all shapes

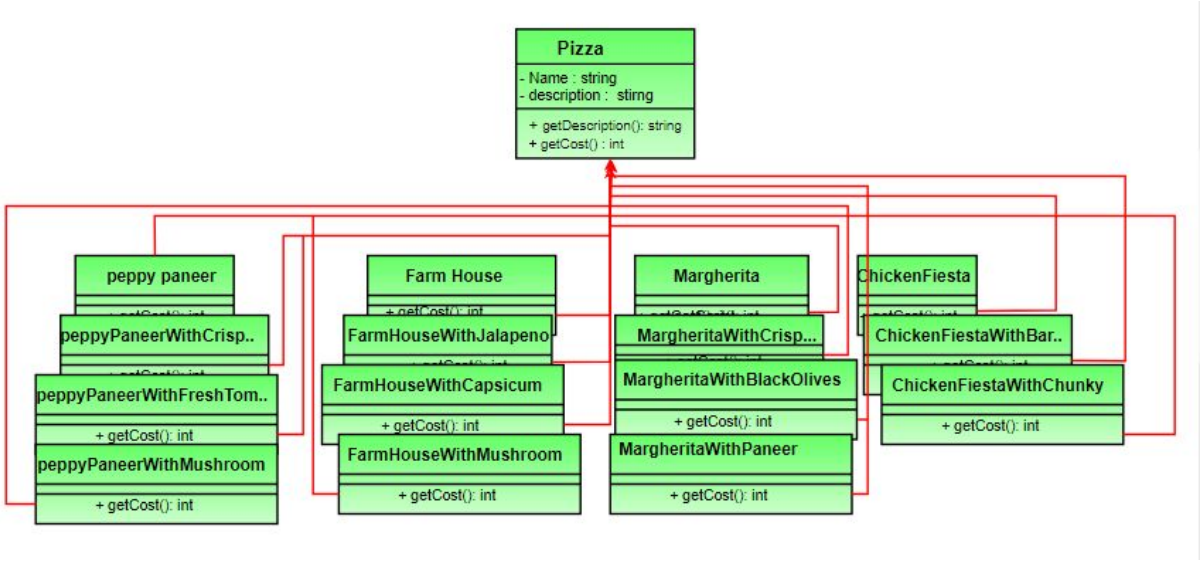
Consider this problem: we are building an app for a pizza store and need to model their classes of pizza. They have four types, so we use *inheritance* to abstract out the common functionality in a Pizza class



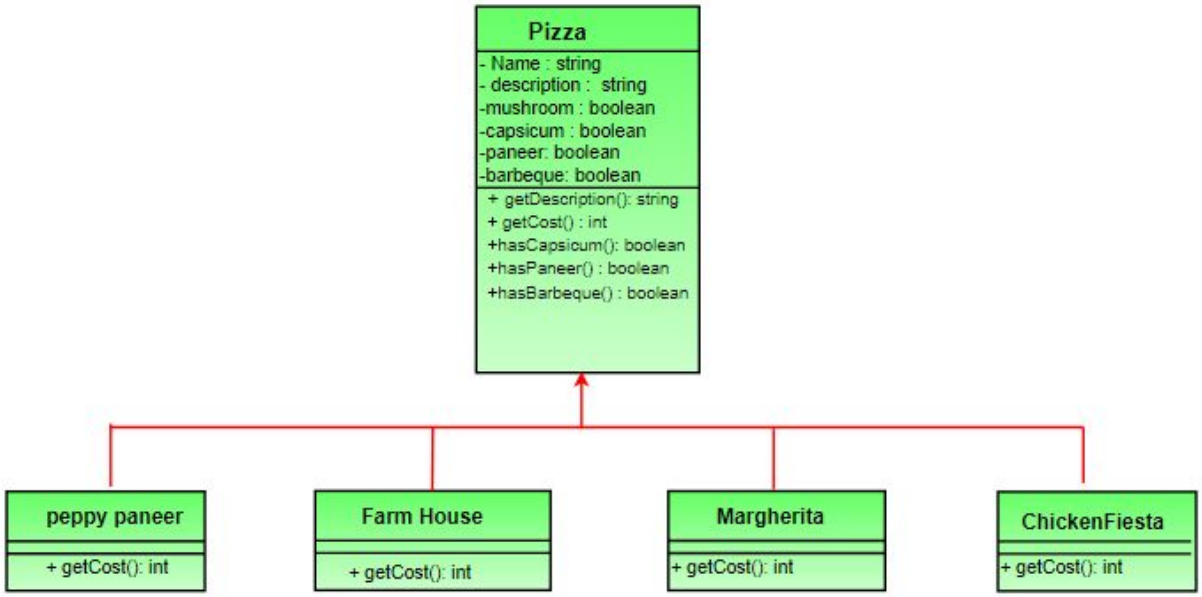
Each pizza has a different cost. We override `getCost()` in the subclasses to find the appropriate cost

Now suppose the customer can ask for additional toppings such as Fresh Tomato, Paneer, Jalapeno, Capsicum, Barbeque, etc.

If we create a new subclass for every topping, we get a big mess - and this only allows one additional topping!



Let’s instead add instance variables to the base class for each topping



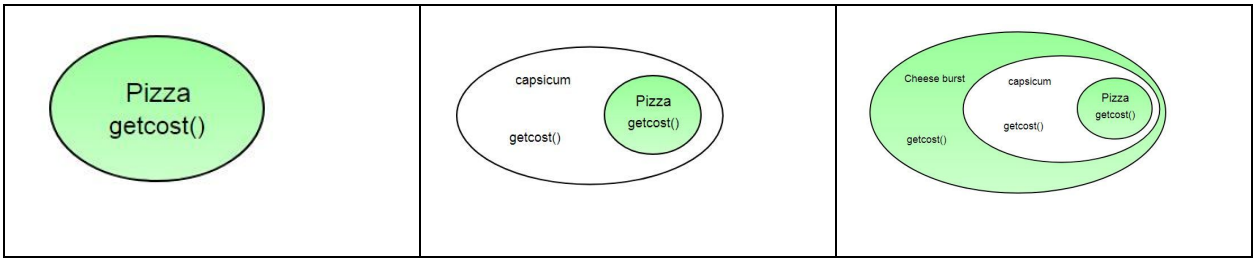

```
// Sample getCost() in parent
class
public int getCost()
{
    int totalToppingsCost = 0;

    if (hasJalapeno() )
      totalToppingsCost +=
jalapenoCost;

    if (hasCapsicum() )
      totalToppingsCost +=
capsicumCost;

    // similarly for other
toppings
    return totalToppingsCost;
}
```

```
// Sample getCost() in child
class
public int getCost()
{
    // 100 for Margherita and
super.getCost()
    // for toppings.
    return super.getCost() +
100;
}
```

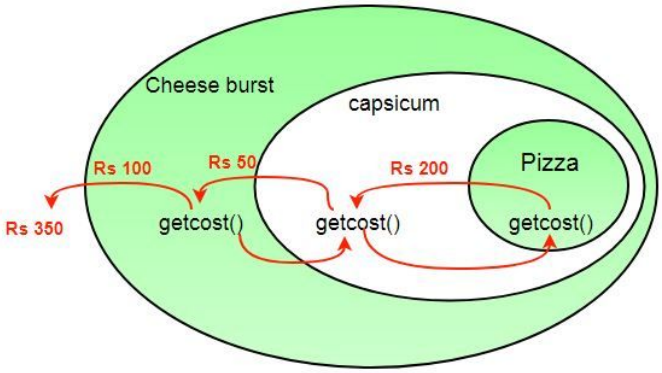
Problems with this design:

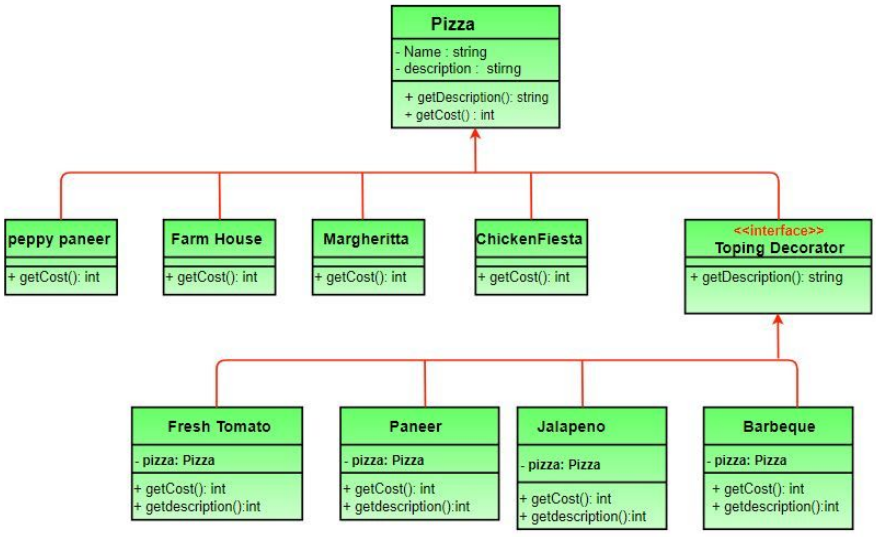
- Price changes in toppings will lead to altering existing code.
- New toppings will force us to add new methods and alter the `getCost()` method in the Pizza base class.
- For some pizzas, some toppings may not be appropriate yet the subclass inherits them.
- What if the customer wants double capsicum?

So instead, let's "decorate" instances of the Pizza class with toppings at runtime:



Use delegation to calculate toppings cost:





```
// Abstract Pizza class (All classes extend
// from this)
abstract class Pizza
{
    // abstract pizza
    String description = "Unknown Pizza";

    public String getDescription()
    {
        return description;
    }

    public abstract int getCost();
}

// The decorator class extends Pizza to be
// interchangable with it toppings decorator
// can also be implemented as an interface
abstract class ToppingsDecorator extends Pizza
{
    public abstract String getDescription();
}
```

```
// Concrete pizza classes
class PeppyPaneer extends Pizza
{
    public PeppyPaneer() { description =
"PeppyPaneer"; }
    public int getCost() {  return 100; }
}
class FarmHouse extends Pizza
{
    public FarmHouse() {  description = "FarmHouse";
}
    public int getCost() { return 200; }
}
class Margherita extends Pizza
{
    public Margherita()  { description =
"Margherita"; }
    public int getCost() { return 100;  }
}
class ChickenFiesta extends Pizza
{
    public ChickenFiesta() { description =
"ChickenFiesta";}
    public int getCost() { return 200; }
}
class SimplePizza extends Pizza
{
public SimplePizza() { description = "SimplePizza"; }
public int getCost() {  return 50;  }
}
```

```
// Concrete toppings classes
class FreshTomato extends ToppingsDecorator
{
    // we need a reference to the obj we are decorating
    Pizza pizza;

    public FreshTomato(Pizza pizza) { this.pizza = pizza; }
    public String getDescription() {
        return pizza.getDescription() + ", Fresh Tomato ";
    }
    public int getCost() { return 40 + pizza.getCost(); }
}
class Barbeque extends ToppingsDecorator
{
    Pizza pizza;
    public Barbeque(Pizza pizza) {  this.pizza = pizza;  }
    public String getDescription() {
        return pizza.getDescription() + ", Barbeque ";
    }
    public int getCost() { return 90 + pizza.getCost(); }
}
class Paneer extends ToppingsDecorator
{
    Pizza pizza;
    public Paneer(Pizza pizza)  {  this.pizza = pizza; }
    public String getDescription() {
        return pizza.getDescription() + ", Paneer ";
    }
    public int getCost()  {  return 70 + pizza.getCost(); }
}


// Other toppings can be coded in similar way
```

```
// Driver class and method
class PizzaStore
{
    public static void main(String args[])
    {
        // create new margherita pizza
        Pizza pizza = new Margherita();
        System.out.println( pizza.getDescription() +
                            " Cost :" +
pizza.getCost());

        // create new FarmHouse pizza
        Pizza pizza2 = new FarmHouse();

        // decorate it with freshtomato topping
        pizza2 = new FreshTomato(pizza2);

        //decorate it with paneer topping
        pizza2 = new Paneer(pizza2);

        System.out.println( pizza2.getDescription() +
                           " Cost :" +
pizza2.getCost());
   }
}
```

Output

Margherita Cost :100
FarmHouse, Fresh Tomato , Paneer Cost :310

Many more design patterns! This website gives examples of each pattern in C++, Java, Python and Typescript (~= Javascript). This website has true Javascript examples.

Original Gang of Four design patterns book (23 patterns)

Head First has a new design patterns book, the old one (which I've previously used for this class) is here

I'm starting to tabulate the class participation grade from posts on piazza. If you are using a different name in piazza than what courseworks thinks is your name, you need to let me know - you can post this on piazza, post to "instructors" or "kaiser". Also, if you're using a different name on zoom, let me know that too.

Assignment T6: Final Demo: https://courseworks2.columbia.edu/courses/104335/assignments/491047 still due December 10

Seeking volunteer teams to do demos in class on either Tuesday December 8th or Thursday December 10 (last two class sessions). Extra credit! You still need to do a separate demo for your IA mentor to give your IA a chance to “drive”.

More Extra Credit: Optional Demo Video https://courseworks2.columbia.edu/courses/104335/assignments/543277 still due December 20