Objectives

- Decorator design pattern
- Eclipse debugger

1

Review

- What is a prototype?
  - What are some ways we can categorize prototypes?
  - What categories does the preliminary implementation fall into?
- What is the singleton design pattern?
  - When is it useful?
  - How is it implemented?

2

Preliminary Implementation

- Goals
  - Get your team working together
  - Find kinks in design
    - Rework now instead of later
- Tag your version
- Can keep working after that
  - Return to the tagged version for Monday's demo

Ungraded Objectives

- Think about what you need to complete for the final implementation.
- With your current design, how well does your design extend for the next steps, including the extensions? What could be designed better?
- An hour of thinking about the design and changing the code to improve the design will be worth hours of time later.

Review: Prototypes Overview

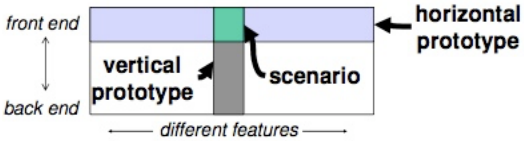
- Demonstrate one part/purpose
  - Focus on one thing, not everything else
- Purpose/Dimensions
  - Functionality
  - Interaction
  - Implementation

5

Review: Prototypes: Fidelity

- ***Fidelity***: how similar to finished product
- Low: omits details
- High: closer to finished project
- Multi-dimensional
  - Breadth: % of features covered
    - Low-breadth: Only enough features for certain tasks
  - Depth: degree of functionality
    - Low-depth: Limited choices, canned responses, no error handling

From Nielsen, *Usability Engineering*



6

Review: Singleton Design Pattern

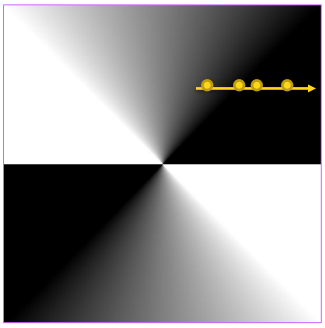
- Goal: Only one object of a class
- How to achieve
  - Make the constructor private
  - Make a public method for accessing the one and only instance

x/y is not the same as y/x

(placement of points is not exact in illustration)

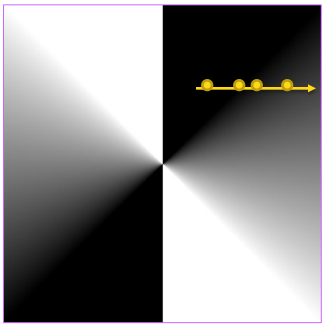
Consider points, holding y steady at -.5

x/y



Y	X	.3	.45	.55	.7
Y=-.5		-.6	-.9	-1.1	-1.4
Color:		Mid-gray	Dark gray	Black	Black

y/x



Y	X	.3	.45	.55	.7
Y=-.5		-1.67	-1.11	-.91	-.71
Color:		Black	Black	Dark gray	Mid dark gray

Picasso GUI Size Change

- Make the image size a square
  - Make width and height the same size
  - Example:

```
public class Main {
    public static final Dimension SIZE =
              new Dimension(600, 600);
```

- Size doesn't matter – code should work regardless of the size

DECORATOR DESIGN PATTERN

What's Your Drink?

- You go into a coffee shop: what is your drink?

- How can we represent the various beverages?
- What are the possible implementation issues?

11

What's Your Coffee Drink?

Beverage
description milk soy flavoring whippedcream
getDescription() cost() hasMilk() setMilk() …

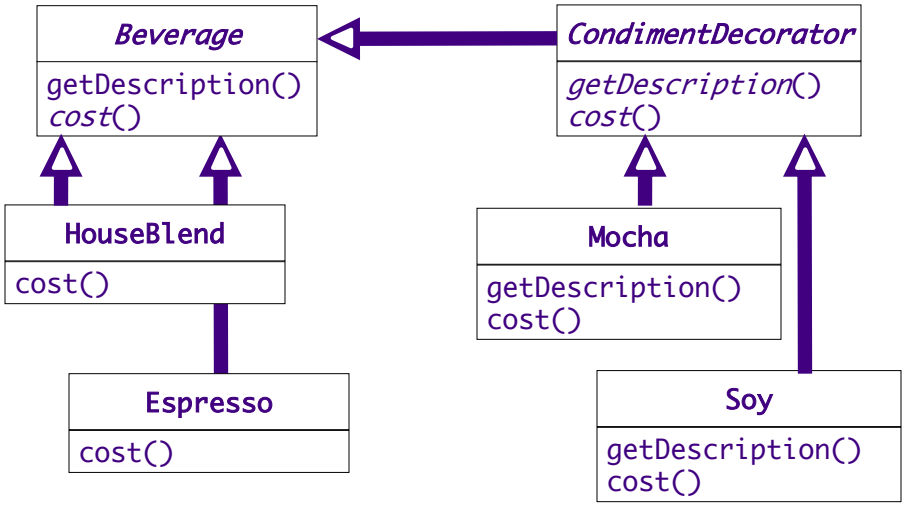
How many additional methods will we need to add to create a comprehensive beverage object?

How will we compute cost?

What happens when a new beverage feature is added?

12

One Solution: **Decorator**



UML Diagram

Mocha's Implementation

```
public class Mocha extends CondimentDecorator {

   private Beverage beverage;

   public Mocha(Beverage beverage) {
      this.beverage = beverage;
   }

   public String getDescription() {
      return beverage.getDescription() + ", Mocha";
   }

   public double cost() {
      return .20 + beverage.cost();
   }
}
```

What design patterns are used within this class?
How would we use this class?
How would we create other beverages?

Mocha's Implementation

```
public class Mocha extends CondimentDecorator {

   private Beverage beverage;

   public Mocha(Beverage beverage) {
      this.beverage = beverage;
   }

   public String getDescription() {
      return beverage.getDescription() + ", Mocha";
   }

   public double cost() {
      return .20 + beverage.cost();
   }
}
```

Handles part it knows about, **Delegates** rest to Beverage; Example of OCP

Generalize: when to use the Decorator pattern, tradeoffs of this design pattern

15

Using Beverages

```
public class CoffeeGeneral {

   public static void main(String[] args) {
        Beverage b = new DarkRoast();
        System.out.println(b.getDescription() +
             " $" + b.getCost());

        Beverage b2 = new DarkRoast();
        b2 = new Mocha(b2);
        b2 = new Mocha(b2);
        b2 = new Whip(b2);
        System.out.println(b2.getDescription() +
             " $" + b2.getCost());
   }
}
```


16

Design Pattern: **Decorator**

- Adds behavior to an object dynamically
  - Typically added by doing computation before or after an existing method in the object
- Benefits:
  - Alternative to inheritance
  - Can add any number of decorators
- Possible drawback:
  - Could add many small classes → less than straightforward for others to understand

Have we seen decorators used in practice?

17

Design Pattern: **Decorator**

- Adds behavior to an object dynamically
  - Typically added by doing computation before or after an existing method in the object
- Benefits:
  - Alternative to inheritance
  - Can add any number of decorators
- Possible drawback:
  - Could add many small classes → less than straightforward for others to understand

Have we seen decorators used in practice?

18

Represent Thanksgiving?

```
dinner = new Turkey( new Duck( new Chicken() ) );
```