Objectives

- Decorator design pattern

1

Review

- What is the singleton design pattern?
  - When is it useful?
  - How is it implemented?
- What is the process for evaluating an expression?
  - Consider `floor(y)` and `floor( floor(y) )`
    - Resulting image will not be different
  - Name the components, methods called
    - Template: A calls B's c method, passing in d and e; the method returns f
  - Map back to what these components represent, as appropriate

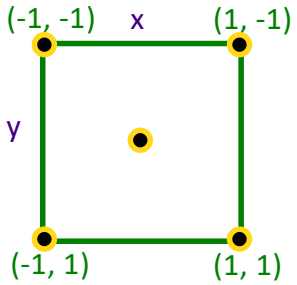
2

Review: Generating Images from Expressions

```
For all x:
    For all y:
        pixels[x][y] = expression.evaluate(x, y)
```

Consider evaluating expression as
f(x, y) = expression
at various points in the image

Example: expression is x+y

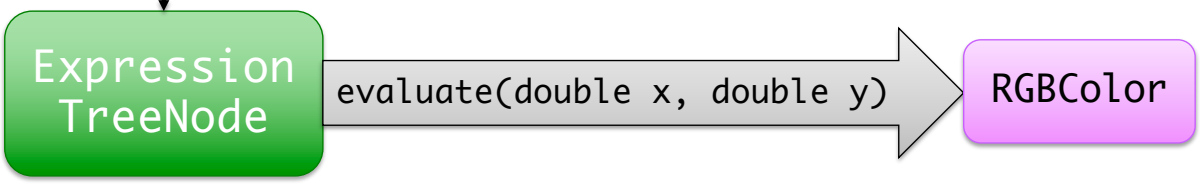


Expression Evaluation

Pixmap

- (x,y) coordinates

Evaluate expression at each x, y coordinate
Returns the RGBColor that should be displayed at that coordinate



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

Picasso Notes

- Given code base is not perfect but pretty good
- Example imperfections
  - Missing comments/Javadocs
  - Incorrect comments
  - Less-than-ideal naming
  - CharToken takes an `int` as a parameter? (rather than a `char`)
- Project goal: you're gaining ***experience***
  - You'll work with imperfect code bases in the future

Picasso: Your Team's Javadocs

- Automatically generated from `main` branch at 3:58 a.m. every day
- Linked from Documentation section of Picasso project page

FAQ for Picasso

- Linked from the specification page
- Updated as I get new questions

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 Friday's demo

http://cs.wlu.edu/~sprenkle/cs209/projects/final_proj.php#deliverables

9

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?
  - Next steps include the other/different types of expressions/functions, extensions
  - What could be designed better (i.e., make it easier to add these other parts)?
- An hour of thinking about the design and changing the code to improve the design will be worth hours of time later.

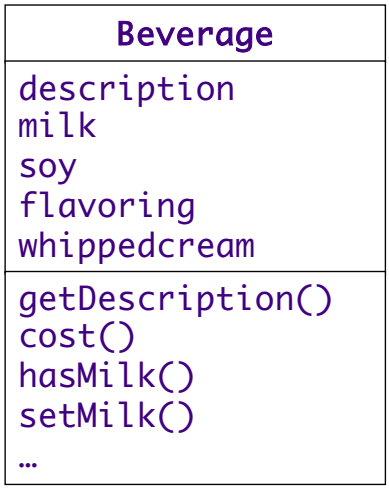
10

DECORATOR DESIGN PATTERN

What's Your Drink?

- You go into a coffee shop: what is your drink?
- How can we represent the various beverages in code?
- What are the possible implementation issues?

What's Your Coffee Drink?

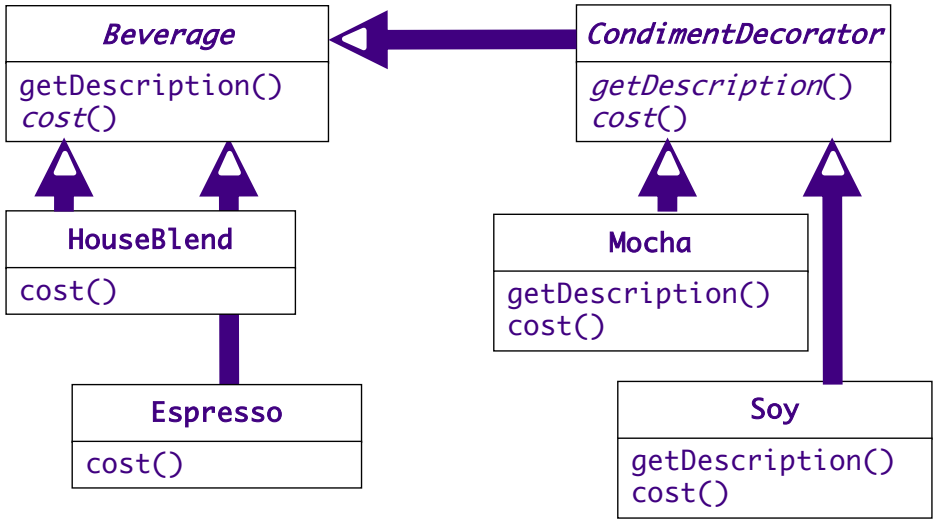


How many additional methods will we need to add to create a comprehensive beverage object?

How will we compute cost?

What happens when a new beverage feature is added?

One Solution: **Decorator**



UML Diagram

Latte's Implementation

```
public class Latte extends Beverage {

   private double cost;

   public Latte() {
      this.cost = 3.75;
   }

   public String getDescription() {
      return "Latte";
   }

   public double cost() {
      return this.cost;
   }
}
```

One possibility
(could keep state differently)

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?

17

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());
   }
}
```


18

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
  - Each class is responsible for just one thing
- Possible drawback:
  - Could add many small classes → less than straightforward for others to understand

Have we seen decorators used in practice?

Change in Requirements

- Beverage class has two new methods: `setSize(…)` and `getSize()`
- Condiments should be charged according to size
  - Example: Soy costs 10¢, 15¢ and 20¢ respectively for small, medium, and large

How would you alter the decorator classes to handle this change in requirements?

Handling Change in Requirements

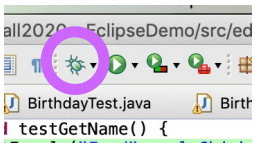
```
public double cost() {
   double cost = beverage.cost();

   if (getSize() == Beverage.SMALL) {
      cost += .10;
   } else if (getSize() == Beverage.MEDIUM) {
      cost += .15;
   } else if (getSize() == Beverage.LARGE) {
      cost += .20;
   }
   return cost;
}
```


Represent Thanksgiving?

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

ECLIPSE DEBUGGER

Eclipse Debugger



1. Set breakpoint
   - Near and BEFORE point of failure
2. Run program in debug mode
   - Program pauses when it hits a breakpoint
3. Inspect variables
4. Step through program, inspecting variables
   - Step into, over, and return

Commands

- Step Into
  - Executes the current line
  - If the current line is a method call, the debugger steps into the method's code
- Step Over
  - Executes a method without stepping into it in the debugger
- Step Return
  - Steps out to the *caller* of the currently executing method
  - Finishes the execution of the current method and returns to the caller of this method

Looking Ahead

- Friday: Preliminary Deadline and Demos
- Order of teams will be randomly generated on Friday
  - Schedule: 8:35, 8:47, 9:00, 9:14
  - Schedule: 11:05, 11:17, 11:30, 11:44
- Next steps:
  - How will you add reading expressions from a file?
  - How will you add other components?