Objectives

- Design patterns
  - Reviewing Composition/Delegation, Singleton, MVC, Factory
  - New: Decorator

Review

1. What is the singleton design pattern?
   - When is it useful? How is it implemented?
2. What is the `instanceof` code smell? Why is it a smell?
   - What is the solution?
3. What is the process for evaluating an expression?
   - Consider `floor(y)` and `x+floor(y+cos(x))`
   - 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
   - Write unit tests for the various components processing the expression.

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` (rather than a `char`) as a parameter?
- Project goal: you're gaining ***experience***
  - You'll work with imperfect code bases in the future

Adding to the Workflow

- After you merge a pull request and pull main on your machine, test it out to see if things are still working as expected.
- Try to find issues before teammates do

Review: `instanceof` Code Smell

- Problem:
  - Code specific to each possible type → Hard to update as add new types
- Solution: Refactor! Add abstraction! (as usual)
  - Specifically: make a method for that functionality in the classes
  - Let dynamic dispatch call the appropriate method.

Design Pattern

General reusable solution to a commonly occurring problem in software design

- Not a finished design that can be transformed directly into code
- Description or *template* for how to solve a problem that can be used in many different situations
  - "Experience reuse", rather than code reuse

Review: Delegation/Composition Design Pattern

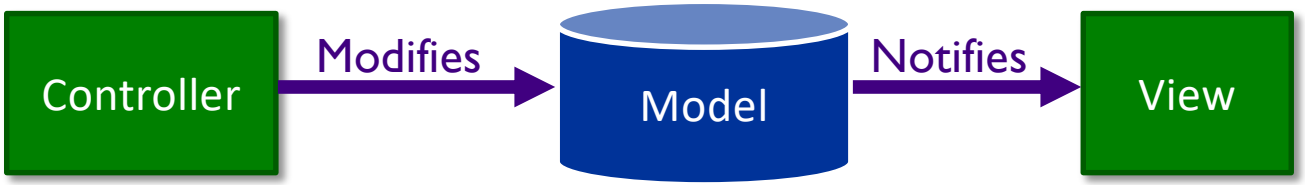
- Principle: Favor composition over inheritance
- Inheritance is a close coupling
  - If parent changes, then child is affected
- Instead of inheriting from a class ("is a" relationship), have a "has a" or "uses a" relationship with the class
  - The class is "*composed* of" the other object or it *delegates* to that other object

Review: Singleton Design Pattern

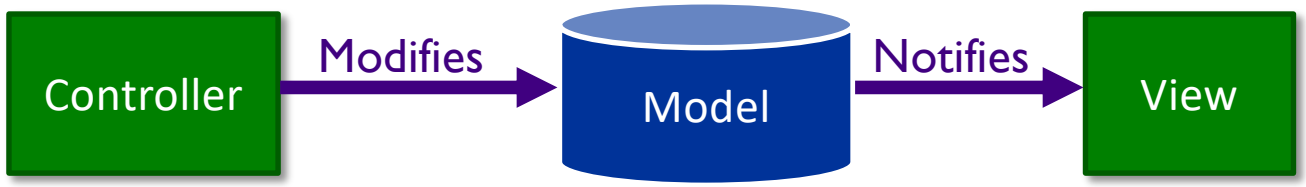
- Goal: Only one object of a class
- How to achieve
  - Make the constructor private
  - Make a public method for accessing the one and only instance

Model - Viewer - Controller (MVC)

- A common **design pattern** for GUIs
- Loosely coupled
  - Model: application data
  - View: graphical representation
  - Controller: input processing

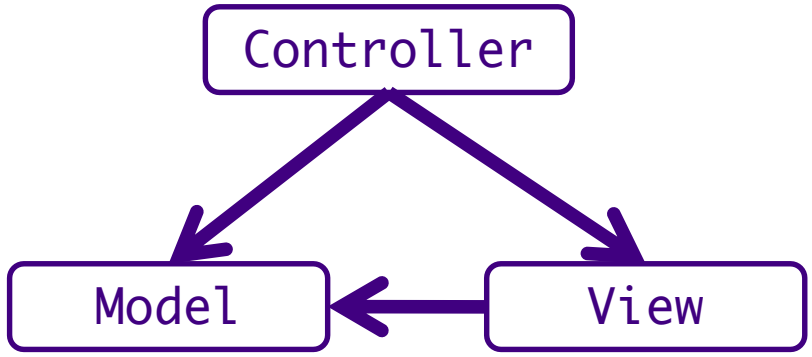


Model-Viewer-Controller



- Can have multiple viewers and controllers
- Goal: modify one component without affecting others

Direct associations



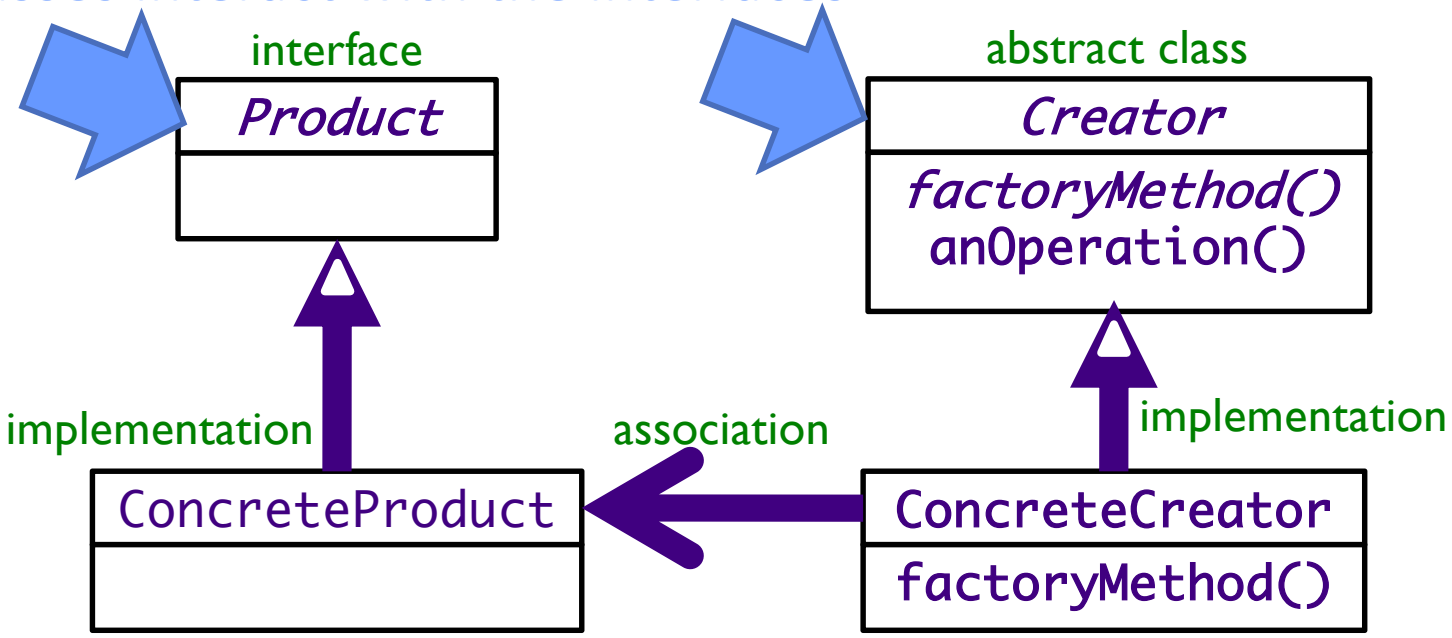
FACTORY DESIGN PATTERN

Design Pattern: **Factory Methods**

- Allows creating objects without specifying exact (concrete) class of created object
- Often used to refer to any method whose main purpose is creating objects
- How it works:
  1. Define a method for creating objects
  2. Child classes override method to specify the derived type of product that will be created

Factory Method Pattern

Client classes interact with the interfaces



Dependency Inversion Principle

Depend upon Abstractions

"Inversion" from the way you think

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?

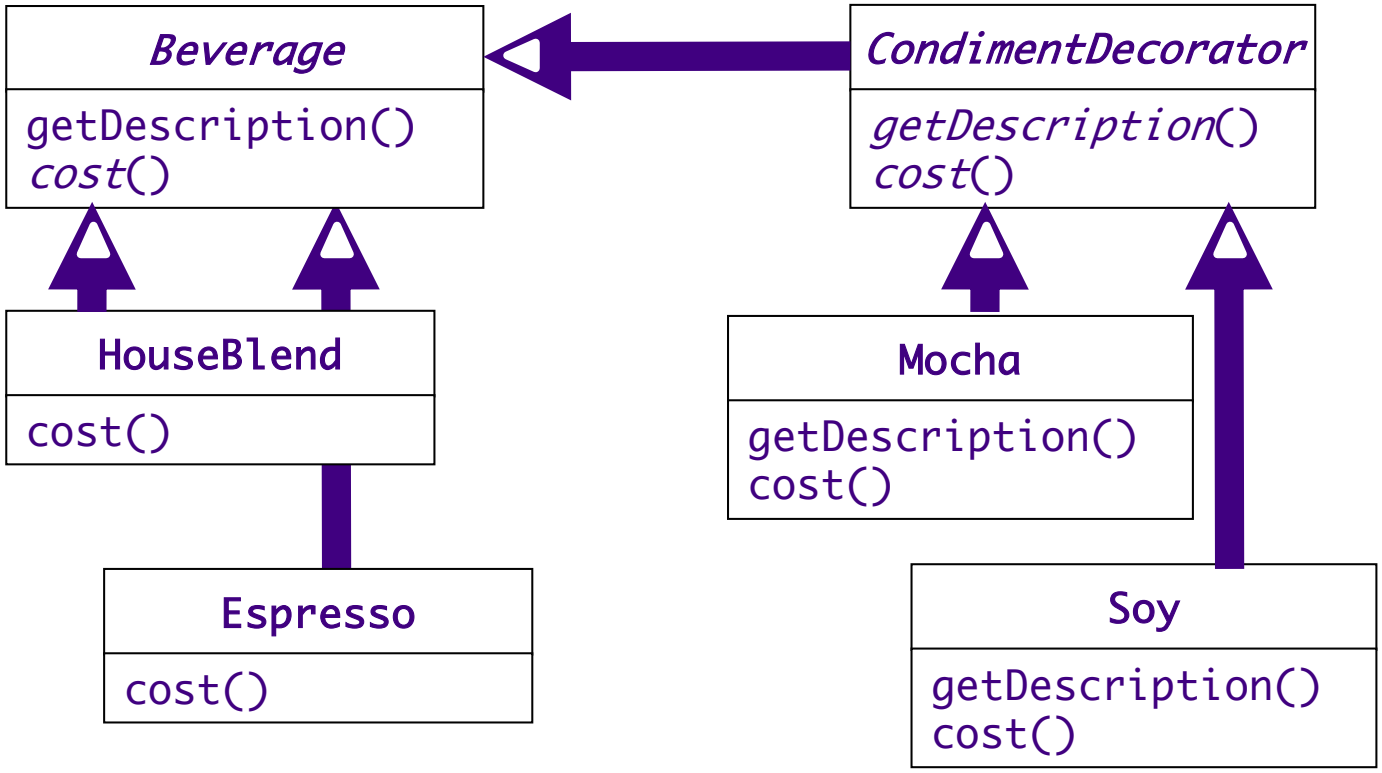
Beverage
description milk soy flavoring whippedcream
getDescription() cost() hasMilk() setMilk() …

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?

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?

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());
   }
}
```

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?

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?

Represent Thanksgiving?

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

Not-always-culturally-relevant: Christmas Tree

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

Reload the page to see changes/updates

FAQ for Picasso

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

Reload the page to see changes/updates

Preliminary Implementation

- Goals
  - Get your team working together, familiar/comfortable with pull requests
    - No one left out, no one dominating
  - 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

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 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.

Secondary Project Goals

- You're going to figure out that your final design isn't perfect—maybe not even good!
- Fix smaller and/or more critical things
  - Refactoring!
- Note larger things
  - Analysis/post-mortem due at end of finals week

Good judgment comes from experience.
How do you get experience?
Bad judgment works every time.

Looking Ahead

- Friday: Preliminary Deadline and Demos
- Order of teams will be randomly generated on Friday
  - Schedule: 9:15, 9:27, 9:40, 9:55
  - Schedule: 11:35, 11:47, 12:00, 12:15
- Next steps include
  - Learning more about
    - tokenization to implement the image-manipulation functions and assignment
    - Infix to postfix to handle order of operations
    - The GUI and streams to read input from a file
  - How will you add other components?