More Design Techniques

CSCI 4448/5448: Object-Oriented Analysis & Design
Lecture 24 — 11/15/2012

Goals of the Lecture

- Cover the material in Chapters 15 & 16 of our textbook
  - Commonality and Variability Analysis
  - The Analysis Matrix
- And, jump ahead and cover a design pattern
  - Decorator from Chapter 17

More Design Techniques

- You may not always be able to use design patterns to start your designs
    - You can however apply the lessons learned from studying design patterns in ALL of your designs
- For instance, you can apply a technique known as commonality and variability analysis to identify variation in your problem domains
    - You can then use design pattern principles to encapsulate that variation and protect your software from potential changes

Examine the Problem Domain (I)

- One key aspect of design is identifying *what* elements of the **problem domain** *belong* in your **solution domain**
  - You need to identify the right things to model (or track) in your software
  - You want to do this as accurately as possible because the next step is to identify the relationships between those concepts
  - Once you have the relationships defined, changes to the design become more difficult

Examine the Problem Domain (II)

- Once you have concepts and relationships defined, inserting new concepts and relationships is less easy
  - You have to decide where the new concepts “fit” and how they will be integrated into the existing design
  - Existing relationships may change or be removed and new ones will be inserted
- This is why maintenance is so hard, when you are asked to change existing functionality or add new functionality, you must deal with the web of concepts and relationships that already exist in the system

Use Commonality and Variability Analysis

- Commonality and Variability Analysis attempts to identify the commonalities (generic concepts) and variations (concrete implementations) in a problem domain
    - Such analysis will produce the abstract base classes that can be used to interface with the concrete implementations in a generic way that will enable abstraction, type encapsulation and polymorphism
- Our authors demonstrate/explain the technique by example by applying it to the CAD/CAM problem

Applying CVA to CAD/CAM

- The CAD/CAM problem consists of
  - Version 1 and Version 2 of the CAD/CAM System
  - Slots, holes, cutouts, etc.
  - Version 1 Models and Version 2 Models

- These are **concrete variations** of **generic concepts**
  - CAD/CAM system, Feature, Model
- Generically: Commonality C has variations a, b, c

Another technique

- Take any 2 elements of the problem domain
  - And ask
    - Is one of these a variation of the other?
    - Are both of these a variation of something else?

- Iterate until you start to converge on the commonalities
  - Each with their associated variations
    - which are just concrete elements of the domain

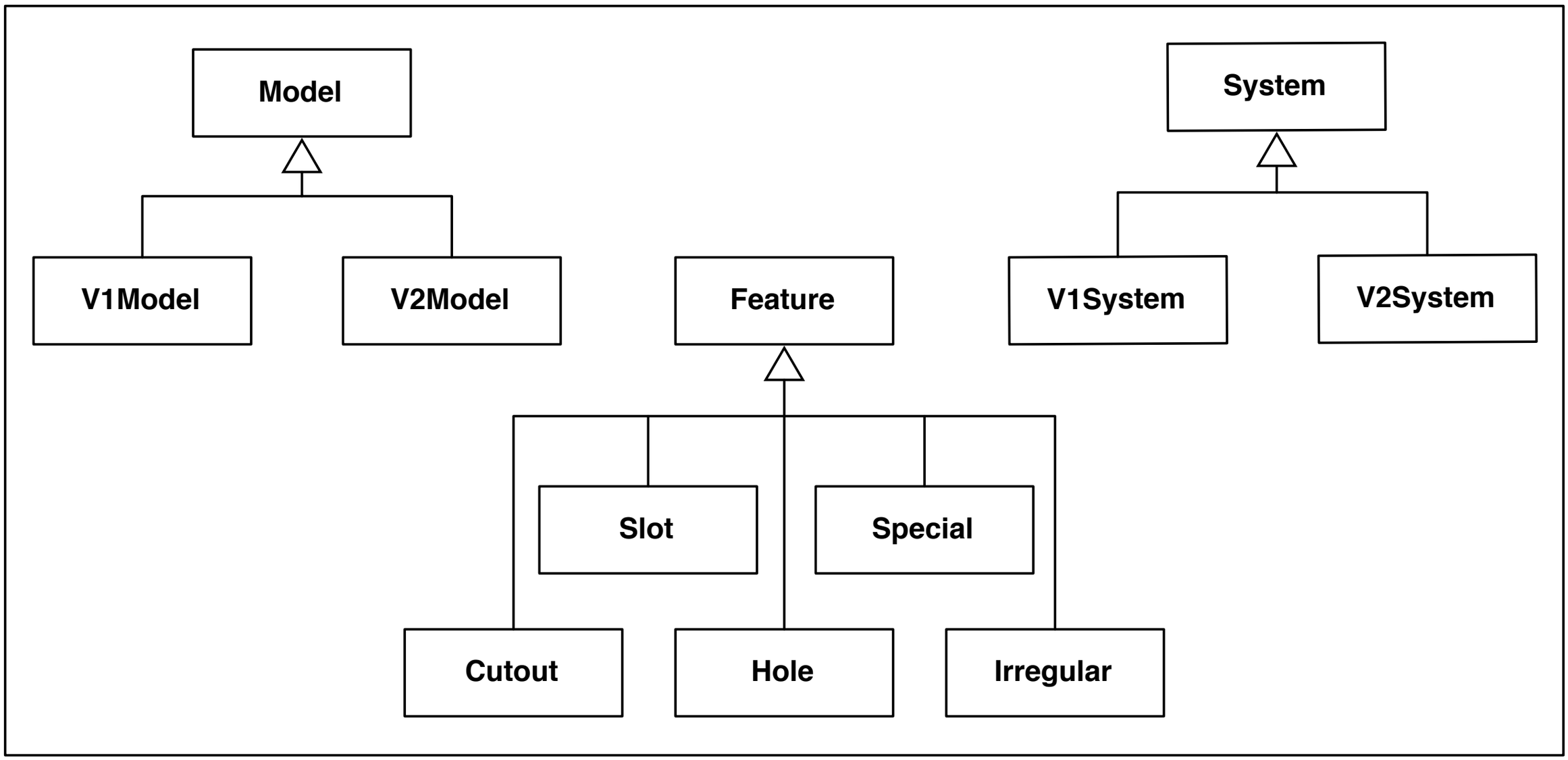
Potential Problem (I)

- Each commonality should be based on one issue
  - Beware collapsing two or more issues into a concept
- For the CAD/CAM domain, you might do something like
  - CAD/CAM Features
    - V1Slot, V2Slot, V1Cutout, V2Cutout
- Here you have collapsed “feature” and “version” into a single concept

Potential Problem (II)

- What you need is
  - CAD/CAM System
    - V1 and V2 (versions)
  - Feature
    - Slots, cutouts, holes, etc.
- Now you have one issue per concept
  - and this will lead to more cohesive designs

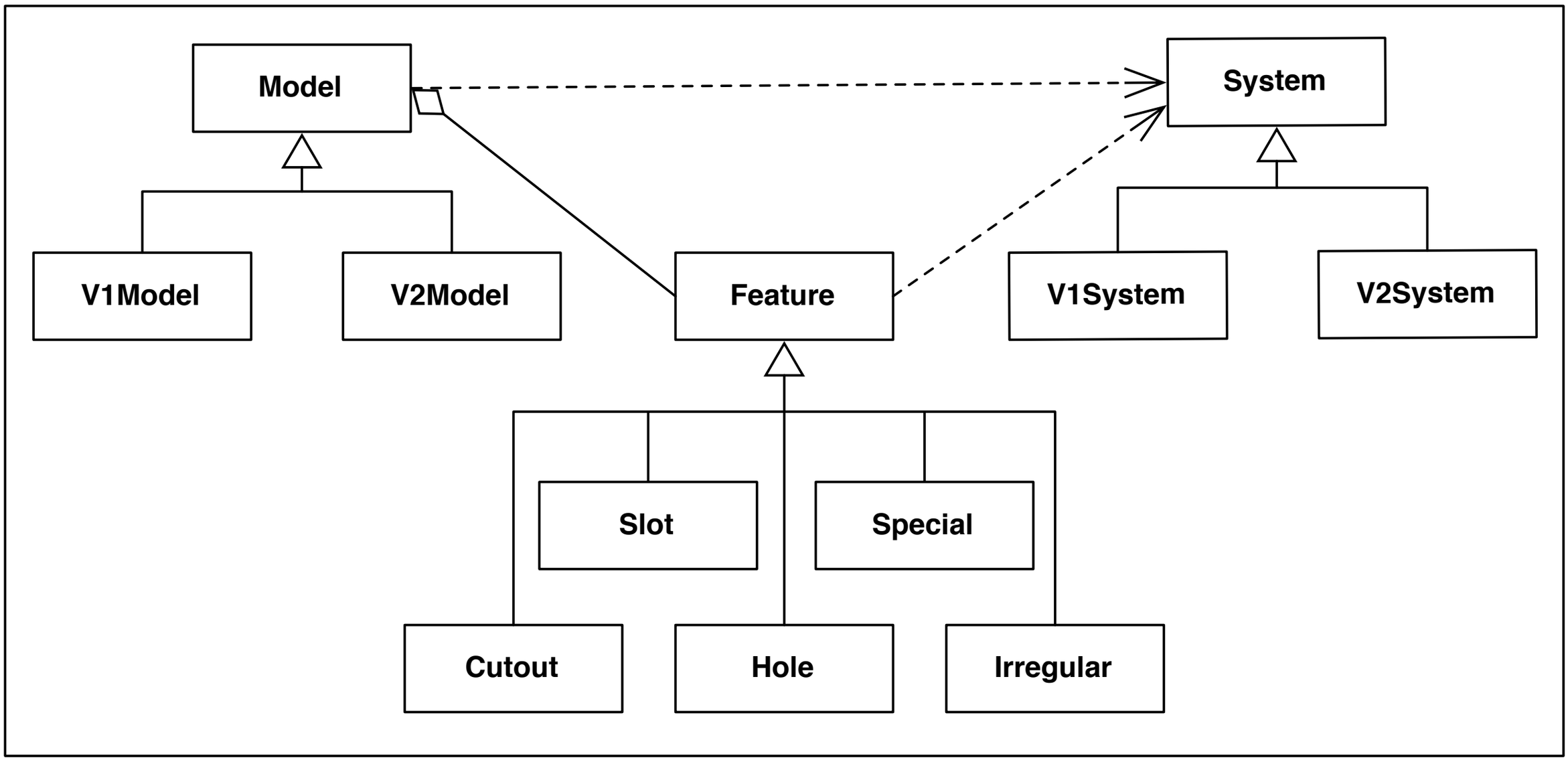
Translating to Classes



Identify Relationships

- If you are confident that you have identified the major concepts of the domain and their variations, you are then ready to identify the relationships that exist between them
  - Models are aggregations of Features
  - Models are generated from a CAD/CAM System
- Features are generated from a CAD/CAM System
  - We will represent “is generated from” as a uses relationship, since it is conceivable that once these concepts are instantiated, they access the CAD system from time to time

Translating to Class Diagram

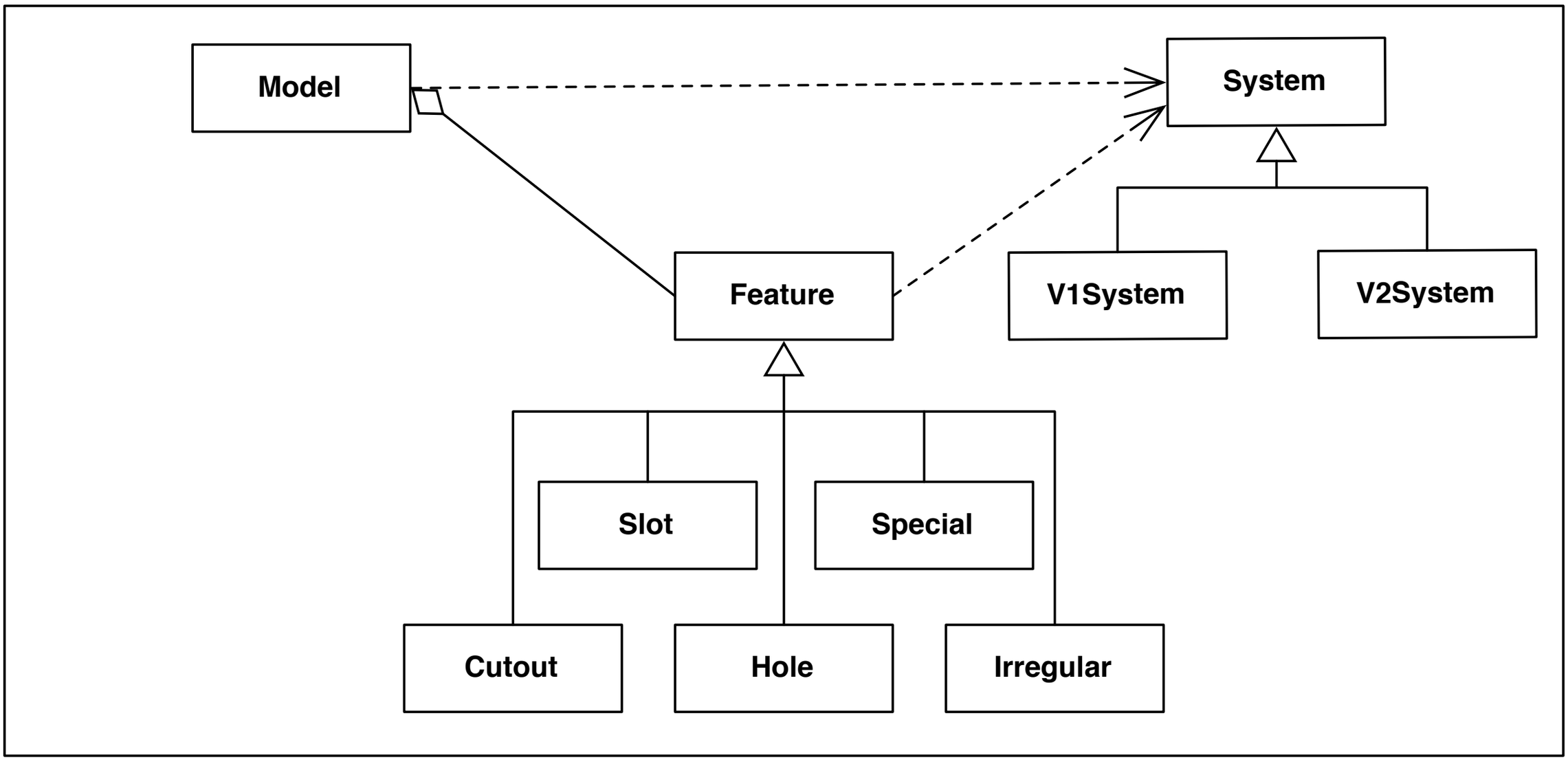


Reflect on the Variations

- We have a duplication in the current design
  - V1Model and V2Model
- AND
  - V1System and V2System

- We can remove this duplication by deciding that a model produced by V1 will have features that are different than a model produced by V2 and so we can get by with just the variation in the CAD/CAM system

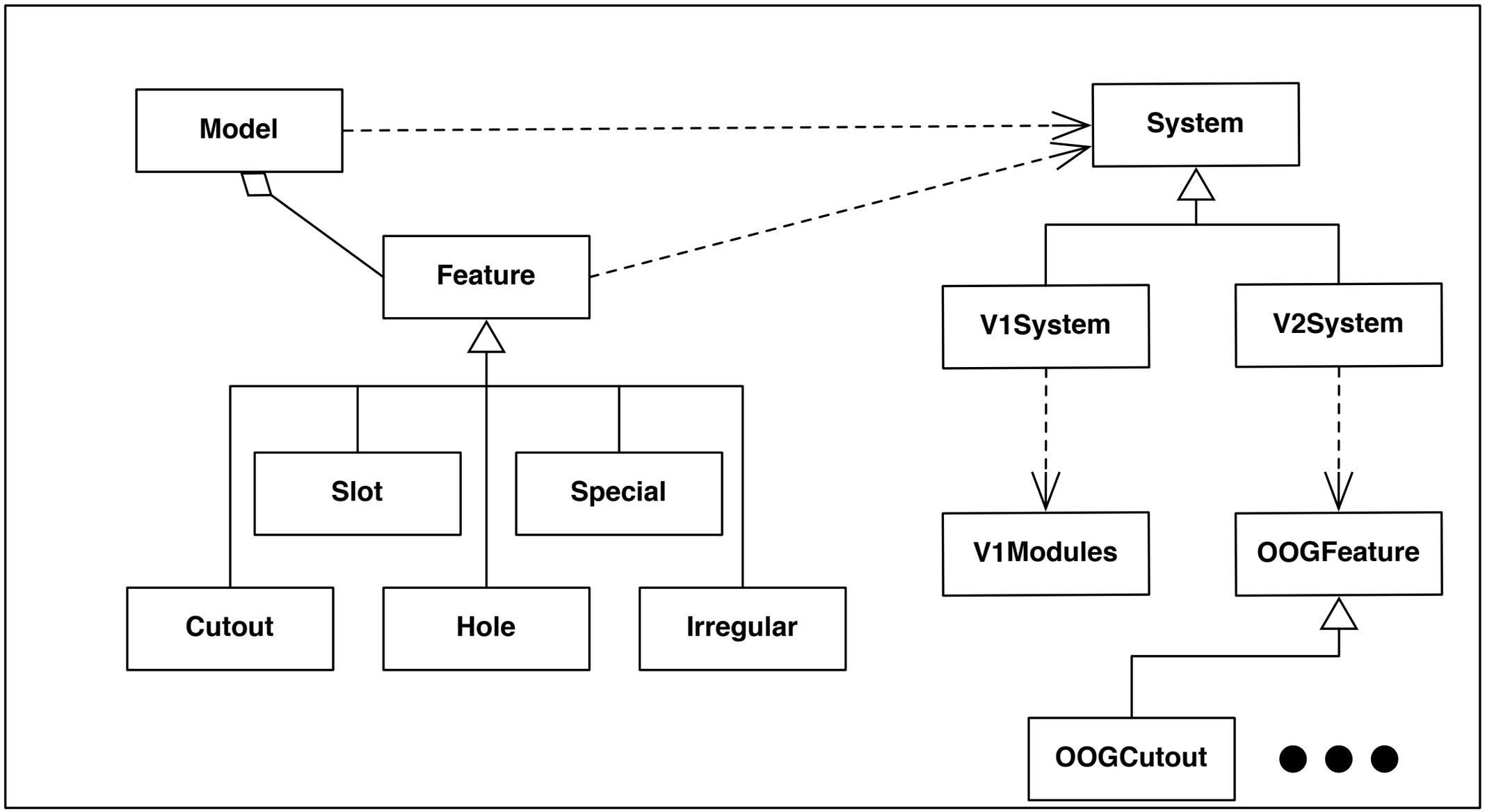
Updated Class Diagram



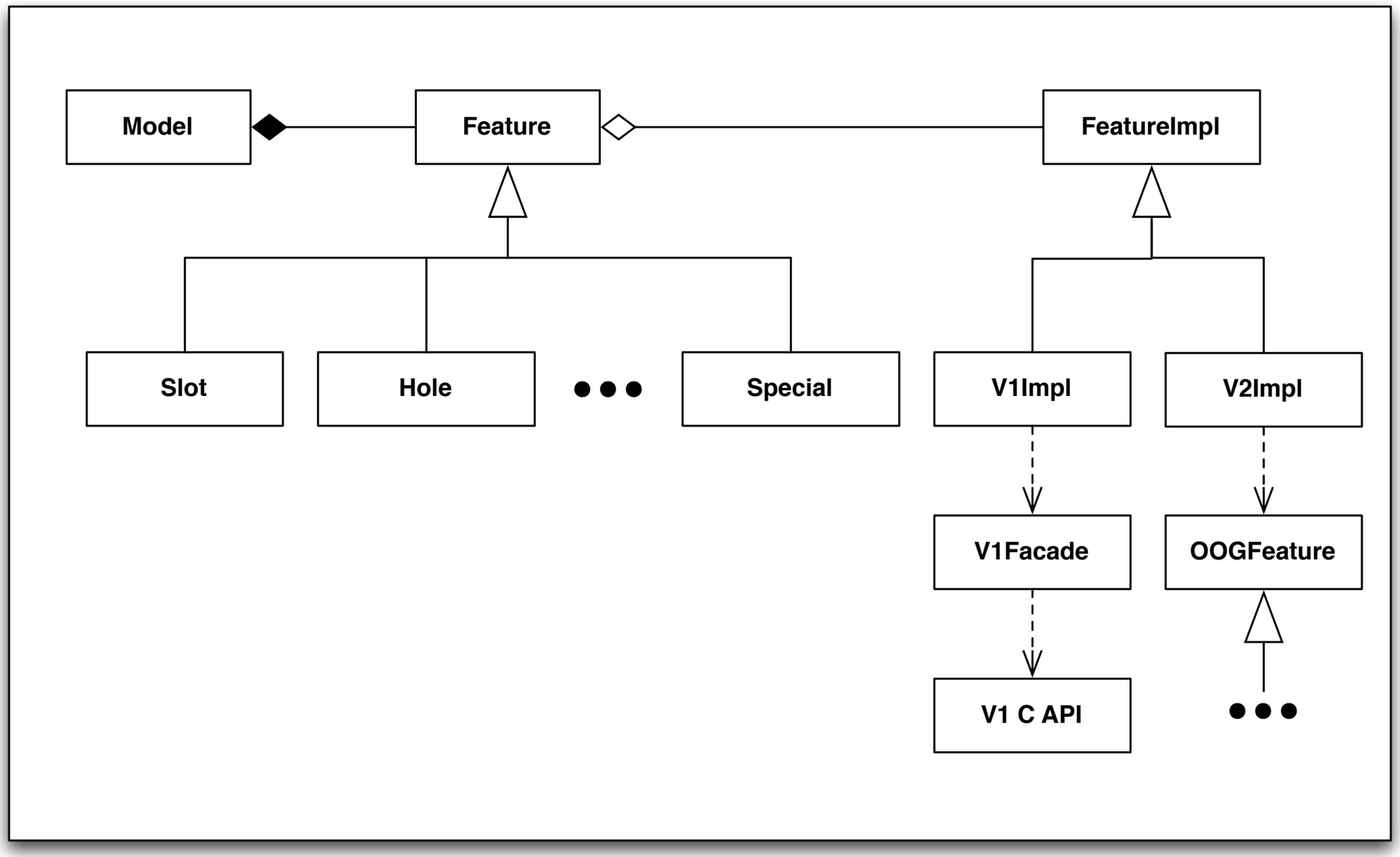
Connect to the Actual Systems

- We must now connect this design to the actual CAD/CAM Systems
  - CAD/CAM Version 1 has the C API
  - CAD/CAM Version 2 has the OO API

Updated Class Diagram



Compare with Previous Design



Comparison with Design Pattern Approach (I)

- CVA produced a design that is similar to the design we created in Lecture 22 using an approach that was driven by design patterns
  - Identify patterns that provide context for other patterns
- The two approaches are synergistic and can be used in tandem
  - Design Patterns establish relationships between entities in the problem domain
  - CVA identifies entities in the problem domain and whether one entity is a variation of another

Comparison with Design Pattern Approach (II)

- The difference is that CVA can be used in all design contexts
    - whereas the design pattern approach requires that you know of design patterns that match the relationships found in the problem domain

Another Technique: Analysis Matrix

- Chapter 16 presents another design technique
  - known as the Analysis Matrix
- to help designers deal with large amounts of variations in a problem domain
- Our authors use an e-commerce system example in which packages must be shipped to customers in response to orders
  - where different rules become active depending on the countries involved in the order

Variations

- Real world domains have a lot of variations
    - Patients always check in to a hospital before they are admitted
        - UNLESS it is an emergency
            - IN WHICH CASE they go to the emergency room to get stabilized and
                - THEN get admitted to the hospital
                    - (IF REQUIRED)

Just like in CVA, Find Concepts

- When dealing with lots of variations, you still ask the question
  - what concept is this “thing” a variation of
- To organize this work, you create a matrix
  - a concept becomes a “row header”
  - a variation is an entry in the matrix
  - related variations go into a column
    - and the column header groups the variations by a particular scenario relevant to the problem domain

Requirements are the Input

- The input to this process are the requirements gathered from the customer
  - For the e-commerce system, we might have reqs like:
    - Calculate shipping based on the country
    - In the US, state and local taxes apply to the orders
    - In Canada, use GST and PST for tax
    - Use U.S. postal rules to verify addresses
    - …

Organize by Matrix

	U.S. Orders	Canadian Orders
Calculate Freight	Use U.S. Postal Rates	Use Canadian Rates
Verify Addresses	Use U.S. Postal Rules	Use Canadian Rules
Calculate Taxes	Use State and Local Tax Rates	Use GST and PST
Money	U.S. Dollars	Canadian Dollars

Organize by Matrix

	U.S. Orders	Canadian Orders
Calculate Freight	CONCRETE IMPLEMENTATIONS OF SHIPPING RATES	
Verify Addresses	CONCRETE IMPLEMENTATIONS OF POSTAL RULES	
Calculate Taxes	CONCRETE IMPLEMENTATIONS OF TAX RULES	
Money	GENERIC CLASS THAT HANDLES CURRENCIES	

Organize by Matrix

	U.S. Orders	Canadian Orders
Calculate Freight	STRATEGY PATTERN	
Verify Addresses	STRATEGY PATTERN	
Calculate Taxes	STRATEGY PATTERN	
Money	GENERIC CLASS THAT HANDLES CURRENCIES	

Organize by Matrix

	U.S. Orders	Canadian Orders
Calculate Freight	THESE IMPLEMENTATIONS ARE USED WHEN WE HAVE A U.S. CUSTOMER	THESE IMPLEMENTATIONS ARE USED WHEN WE HAVE A CANADIAN CUSTOMER
Verify Addresses		
Calculate Taxes		
Money		

Organize by Matrix

	U.S. Orders	Canadian Orders
Calculate Freight	ABSTRACT FACTORY	ABSTRACT FACTORY
Verify Addresses		
Calculate Taxes		
Money		

Discussion (I)

- This technique gets more useful as the matrix gets bigger
  - If you have requirements for a new scenario that adds an additional row (concept) that you have not previously considered
    - this indicates that your previous scenarios were incomplete
    - you can now go back and fill in the missing pieces

Discussion (II)

- Sometimes your special cases will have special cases
  - In the U.S. different shippers may have different requirements and different fees
  - You can capture this information in another analysis matrix that shares some of the columns and rows of the original but which add additional concepts just for those special situations

Discussion (III)

- We've now seen four design techniques
  - The OO A&D method of Lecture 6
  - Design Pattern-Driven Design
  - Commonality and Variability Analysis
  - Analysis Matrix
- Which one should we use?
  - Whatever helps you make progress! You might use more than one and switch between them until they converge

Decorator Pattern

- The Decorator Pattern provides a powerful mechanism for adding new behaviors to an object at run-time
    - The mechanism is based on the notion of “wrapping” which is just a fancy way of saying “delegation” **but with the added twist that the delegator and the delegate both implement the same interface**
        - You start with object A that implements interface C
        - You then create object B that also implements interface C
        - You pass A into B’s constructor and then pass B to A’s client
        - The client thinks its talking to A (via C’s interface) but its actually talking to B
        - B’s methods augment A’s methods to provide new behavior

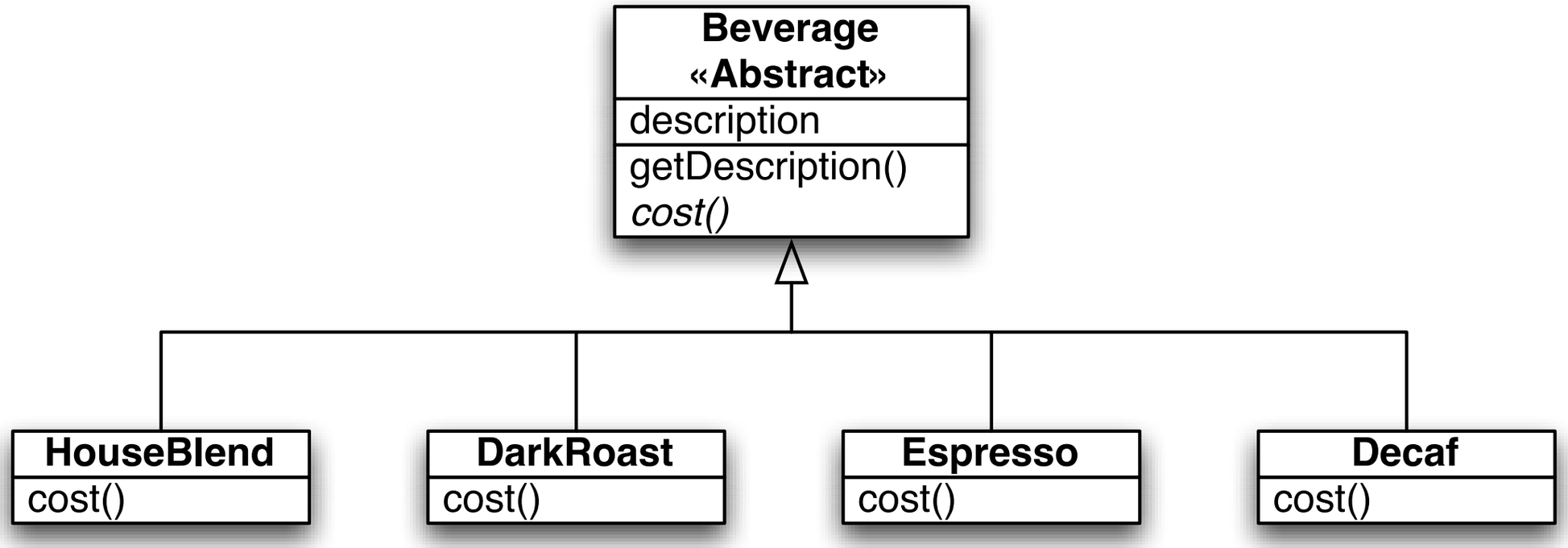
Why? Open-Closed Principle

- The decorator pattern provides yet another way in which a class's runtime behavior can be extended without requiring modification to the class
- This supports the goal of the open-closed principle:
  - Classes should be open for extension but closed to modification
    - Inheritance is one way to do this, but composition and delegation are more flexible (and Decorator takes advantage of delegation)
    - As the Gang of Four put it: "Decorator lets you attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."
- Our "Starbuzz Coffee" example, taken from Head First Design Patterns, clearly demonstrates why inheritance can get you into trouble and why delegation/composition provides greater run-time flexibility

Starbuzz Coffee

- Under pressure to update their “point of sale” system to keep up with their expanding set of beverage products
  - Started with a Beverage abstract base class and four implementations: HouseBlend, DarkRoast, Decaf, and Espresso
    - Each beverage can provide a description and compute its cost
- But they also offer a range of condiments including: steamed milk, soy, and mocha
  - These condiments alter a beverage’s description and cost
    - The use of the word “Alter” here is key since it provides a hint that we might be able to use the Decorator pattern

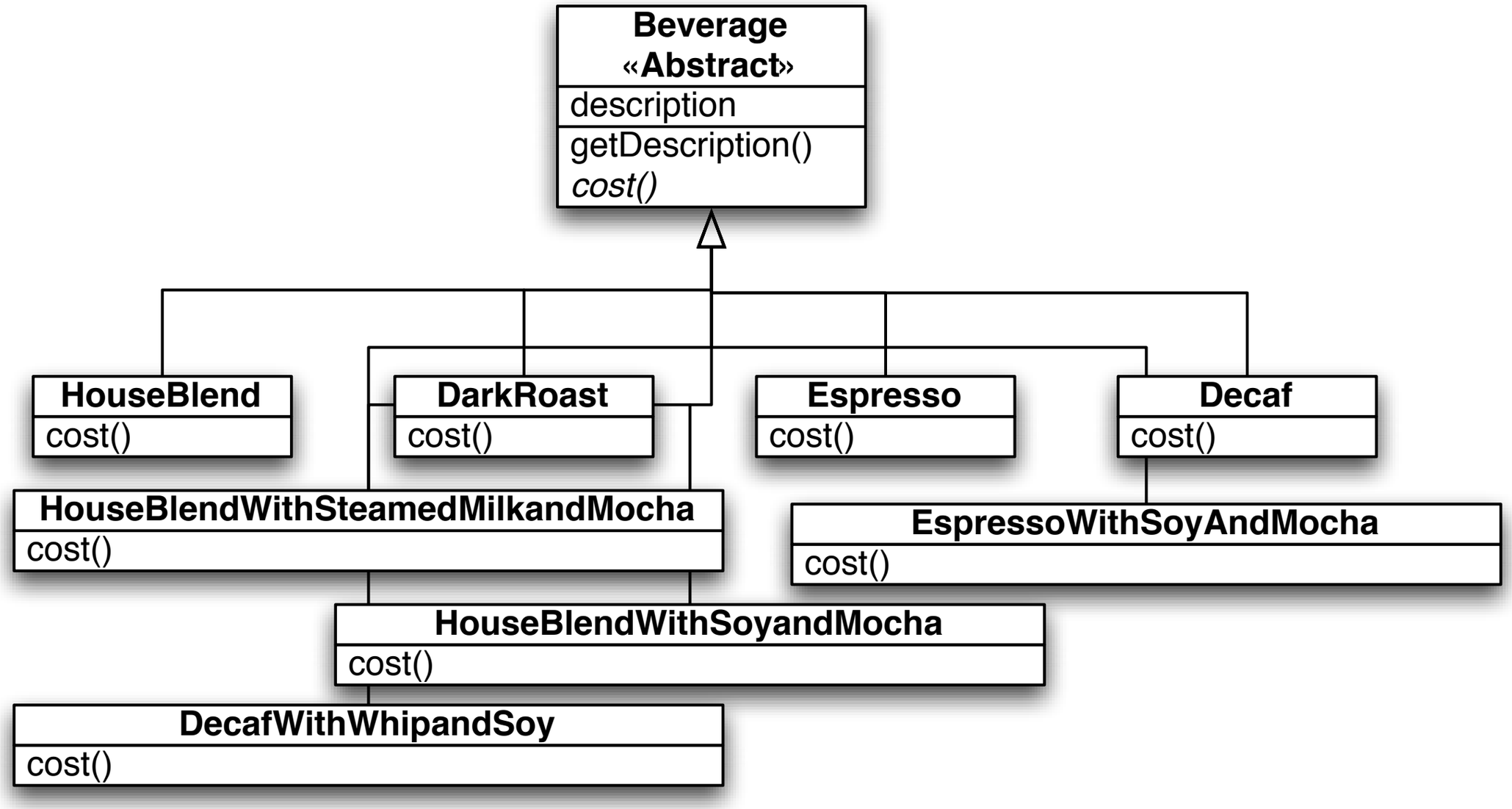
Initial Starbuzz System



With inheritance on your brain, you may add condiments to this design in one of two ways

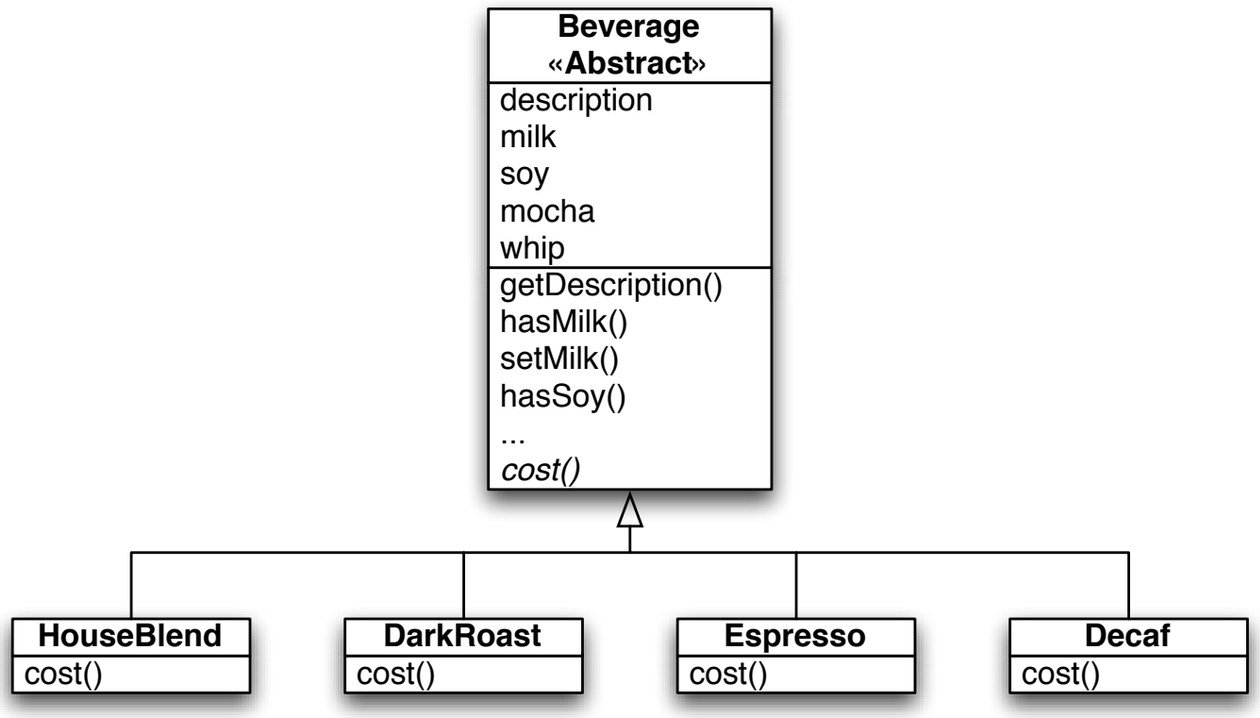
1. One subclass per combination of condiment (wont work in general but especially not in Boulder!)
2. Add condiment handling to the Beverage superclass

Approach One: One Subclass per Combination



This is incomplete, but you can see the problem…

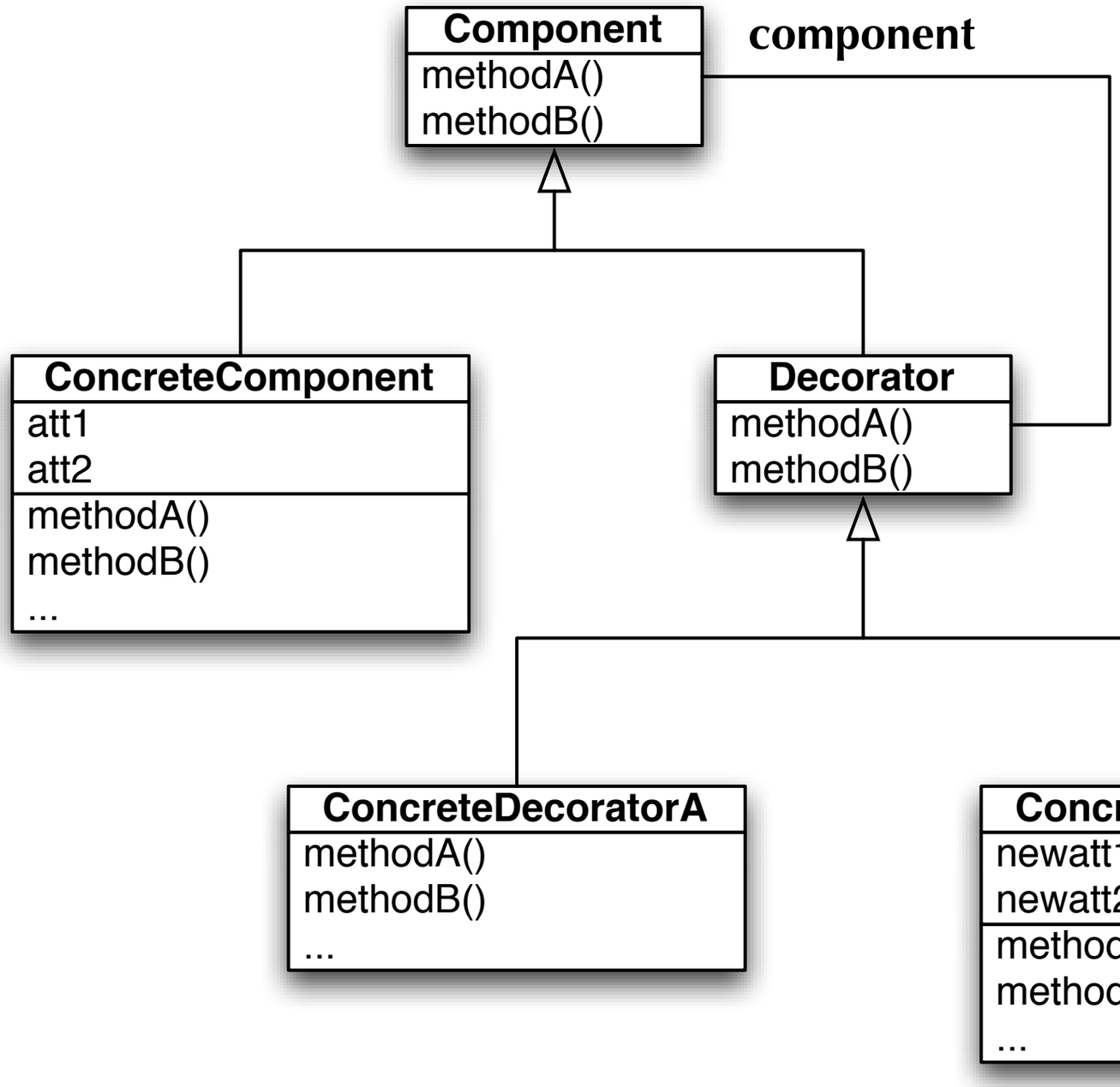
Approach Two: Let Beverage Handle Condiments



Houston, we have a problem…

1. This assumes that all concrete Beverage classes need these condiments
2. Condiments may vary (old ones go, new ones are added, price changes occur, etc.), shouldn't Beverage be encapsulated from this some how?
3. How do you handle "double soy" drinks with boolean variables?

Decorator Pattern: Definition and Structure



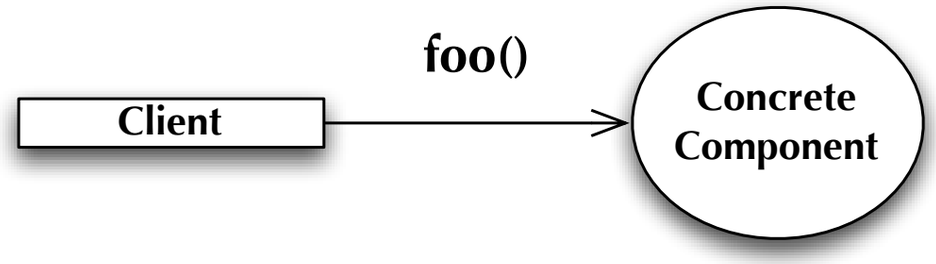
Inheritance is used to make sure that components and decorators share the same interface: namely the public interface of Component which is either an abstract class or an interface

At run-time, concrete decorators wrap concrete components and/or other concrete decorators

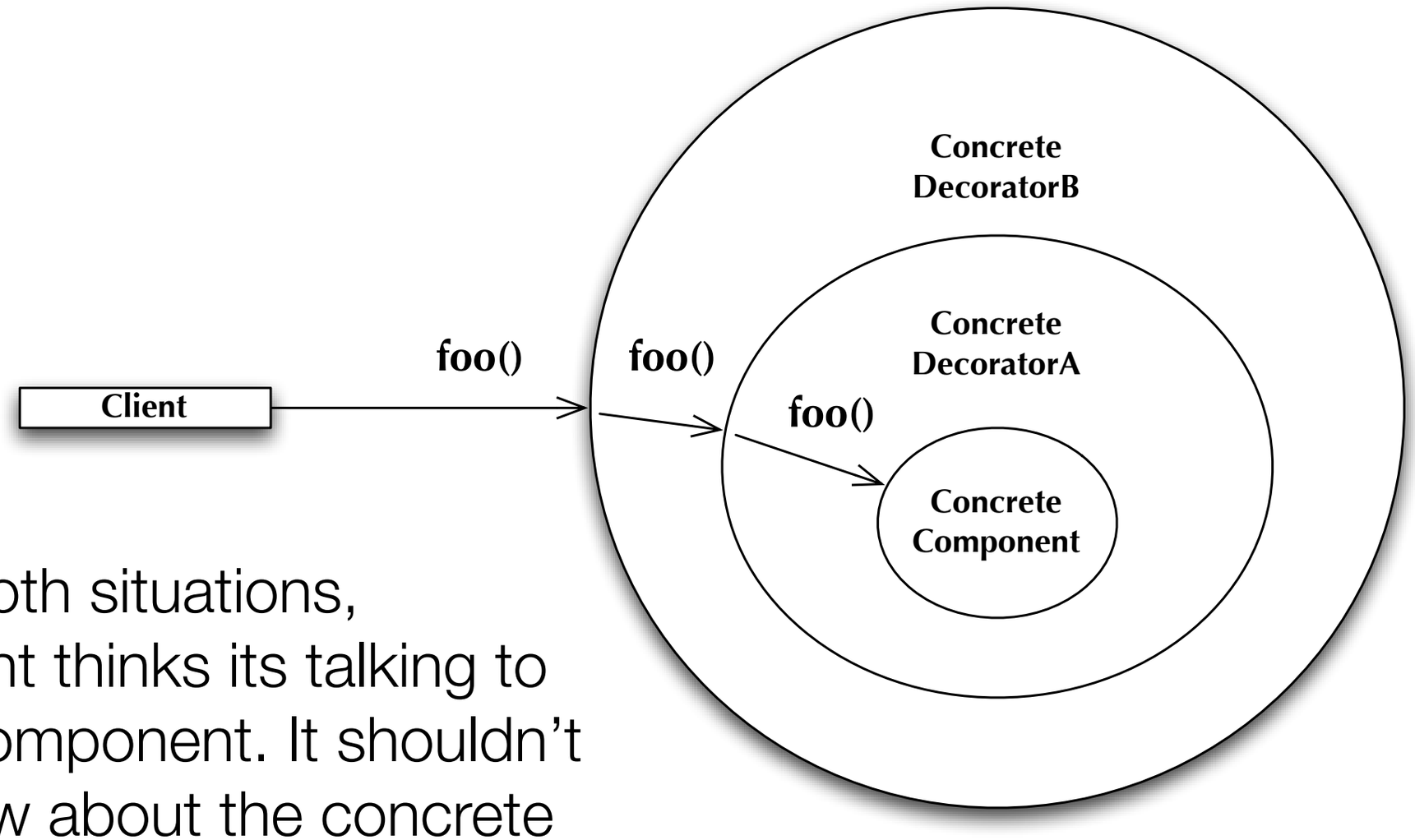
The object to be wrapped is typically passed in via the constructor

Each decorator is cohesive, focusing just on its added functionality

Client Perspective



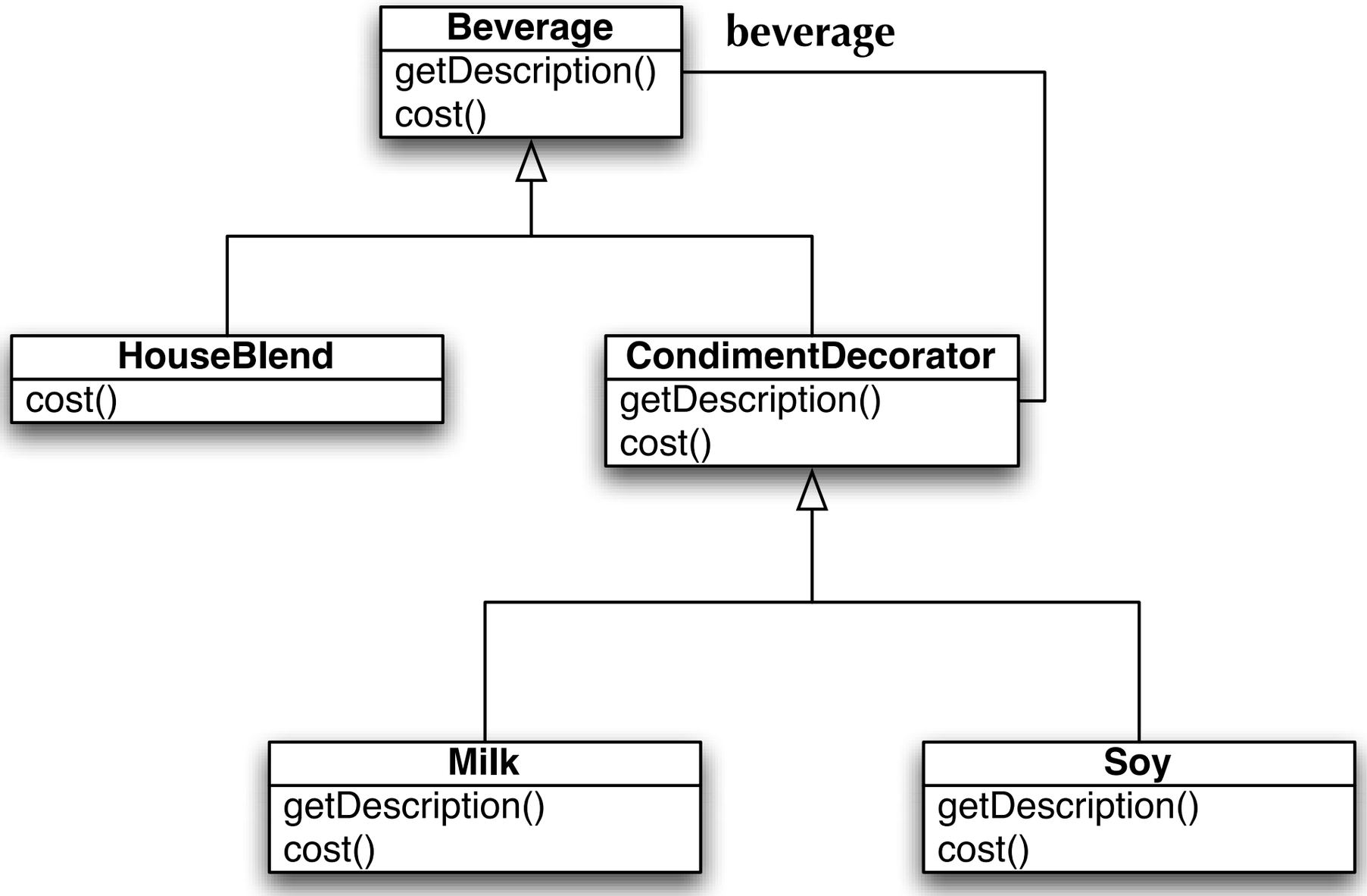
BEFORE



AFTER

In both situations, Client thinks its talking to a Component. It shouldn't know about the concrete subclasses.

StarBuzz Using Decorators (Incomplete)



Demonstration

- Starbuzz Example
- Use of Decorator Pattern in java.io package
  - InputStream == **Component**
  - FilterInputStream == **Decorator**
    - FileInputStream, StringBufferInputStream, etc. == **ConcreteComponent**
    - BufferedInputStream, LineNumberInputStream, etc. == **ConcreteDecorator**

The Textbook's Take (I)

- As we saw, Decorator offers another solution to the problem of rapidly multiplying combinations of subclasses
    - we saw examples of other solutions back in Chapters 8 and 10 where we made use of the strategy pattern and the bridge pattern
- The decorator pattern provides a means for creating different combinations of functionality by creating chains in which each member of the chain can augment or "decorate" the output of the previous member
    - Plus, it separates the step of building these chains from the use of these chains

The Textbook's Take (II)

- The Decorator pattern comes into play when there are a variety of optional functions that can precede or follow another function that is always executed
- This is a very powerful idea that can be implemented in a variety of ways (see the end of Chapter 17 for a discussion of some of the variations)
    - The fact that all of the classes in the decorator pattern hide behind the abstraction of Component enables all of the good benefits of OO design discussed previously

Wrapping Up

- Commonality and Variability Analysis
    - helps us identify generic concepts in our problem domains and their associated concrete implementations
- Analysis Matrix
    - helps us deal with variations in a problem domain and provide hints as to the concepts they are related to and how to implement them
- Decorator Pattern
    - Another technique for applying the open-closed principle

Coming Up Next

- Homework 5 and Presentations are due tomorrow

- FALL BREAK! Happy Thanksgiving!

- When we return
  - Lecture 25: Observer, Template Method
    - Chapters 18 and 19
    - Plus Two Bonus Design Patterns!