

# SENG 350

## - Software Architecture & Design

Shuja Mughal

### **Design Patterns**

Fall 2024



# Milestone 2

## 1. Construct incrementally

This milestone is one of three major design and construction “sprints”. Your system is not expected to be complete at the end of milestone 2, but defined parts of it should be implemented (Coded).

Make sure you deliberately plan the parts you want to finish in this "sprint" and code, respectively. Before submission, report what was achieved regarding the Milestone 2 goals and which adjustments you had to shift to Milestone 3 goals (if any). Be prepared to demo your Milestone 2 results and code in the lab.



# Milestone 2 (cont.)

## 2. Class Diagram

- Create a class diagram for your respective system.
- Create a section "Class Model" on your Wiki that includes the Class Diagram.
- Please add a textual explanation (not just the diagrams).



# Milestone 2 (cont.)

## 3. Collaboration Diagram

- Create a Collaboration Diagram for your respective system.
- Create a “Collaboration Model” section on your Wiki that includes the Collaboration Diagram.
- Please add a textual explanation (not just the diagrams).



# Milestone 2 (cont.)

## 4. Sequence Diagrams

- Create a minimum of five extensive and detailed sequence diagrams to illustrate the interactions within the system.
- Create a “Sequence Models” section on your Wiki that includes the Sequence Diagrams.
- Please add a textual explanation (not just the diagrams).



# Milestone 2 (cont.)

## 5. Activity Diagram

- Specify the major process activity of your application with an activity diagram with regions (swim lanes).
- On your Wiki, create a section "Activity Model" where you include the Activity Diagrams.
- Please add a textual explanation (not just the diagram).



# Milestone 2 (cont.)

## 6. State Machine Diagrams

- Describe the stateful behaviour with a UML Statemachine.
- Make sure that the Statemachines can communicate with appropriate events (or messages) that are also consistent with your Sequence Diagrams and Activity Diagrams.
- There should be a minimum of 5 State Models.
- On your Wiki, create a section "State Models" where you include the Statemachine Diagrams.
- Please add a textual explanation (not just the diagrams).



# Milestone 2 (cont.)

## 7. Data Flow Diagrams (DFD)

- Develop a Level 0 DFD that illustrates the overall flow of information across the platform, providing a high-level view of how data moves between different components.
- Then, break down the primary processes into a more detailed Level 1 DFD to show each process's specific interactions and finer details.
- On your Wiki, create a section "Data Flow Models" where you include the DFD.
- Please add a textual explanation (not just the diagrams).





# Milestone 2 (cont.)

## 8. Entity Relationship Diagram (ERD)

- Create an ERD to represent the system's data structure visually.
- Create a section "Entity Relationship Model" on your Wiki where you include the ERD.
- Please add a textual explanation (not just the diagrams).



# Milestone 2 (cont.)

## 9. Component Diagram

- Develop a Component Diagram to show the system's high-level structure, focusing on its main components and how they interact.
- Create a “Component Model” section on your Wiki where you include the Component/Package Diagram.
- Please add a textual explanation (not just the diagrams).



# Milestone 2 (cont.)

## 10. Deployment Diagram

- Create a Deployment Diagram to illustrate the physical architecture of the system.
- On your Wiki, create a section "Deployment Model" where you include the Deployment Diagram.
- Please add a textual explanation (not just the diagrams).



# Milestone 2 (cont.)

## 11. Contributions

- Summarize the contributions made by each team member.
- Point out the sections/parts each group member was "most responsible" for.

## 11. Submission

- Submit your milestone on GitHub and share the link on Brightspace.
- Also, remember to submit your Teamwork Assessment on Brightspace by the same deadline.



# Design Patterns



# What Is A Design Pattern?

- A general and reusable solution to a commonly occurring problem in software design.
- IT IS NOT a finished algorithm that can be directly translated into program code.
- IT IS a template for how to solve a problem that has been used in many different situations.
- The various Object-Oriented design patterns show interactions between classes and objects without the specific program code that implements the pattern.
  - e.g., Information hiding, inheritance etc.

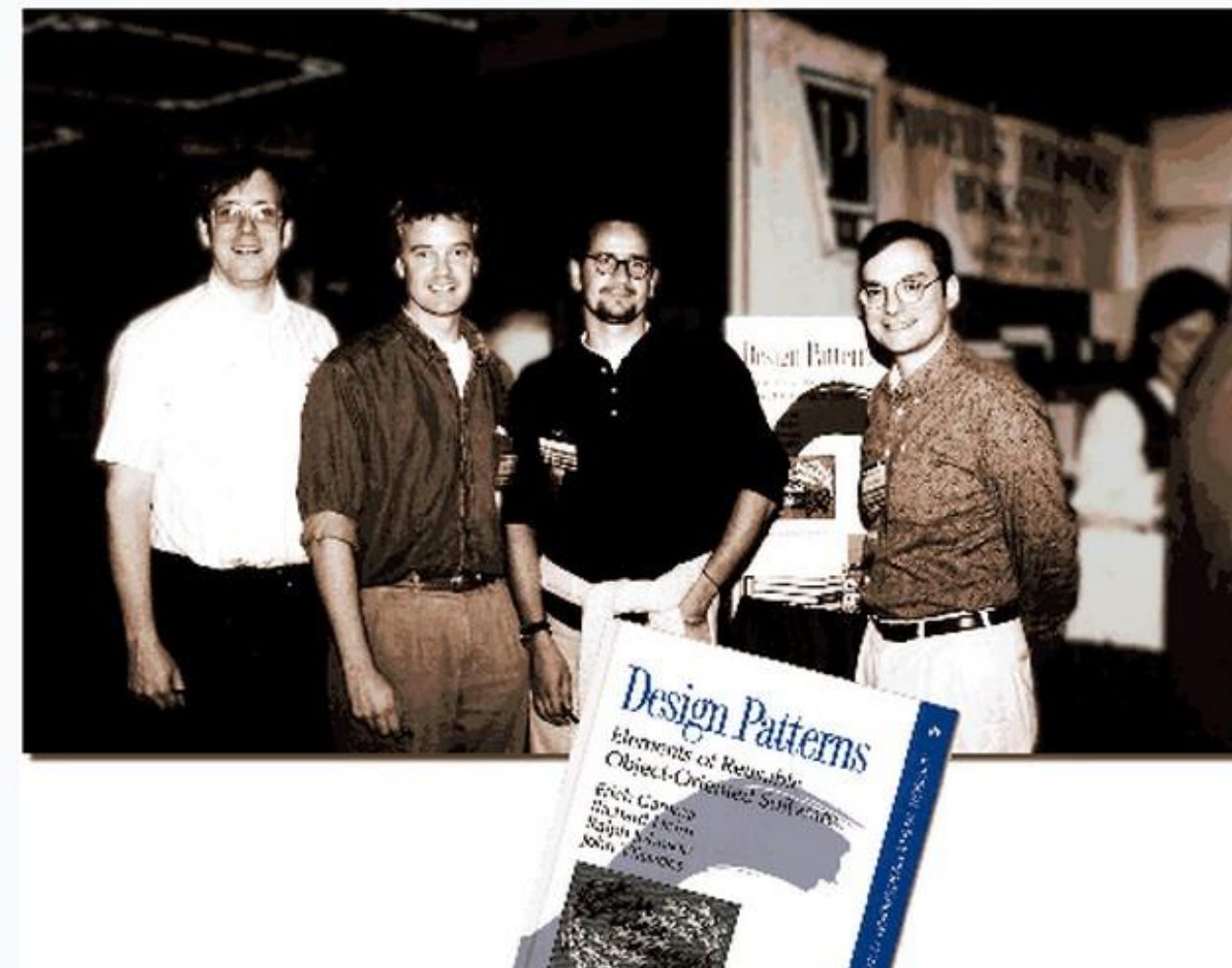


# Dr. Dobb's Journal's 1998 Excellence in Programming Awards

By Jonathan Erickson, March 01, 1998

[Post a Comment](#)

**The "Gang of Four" — Richard Helm, Erich Gamma, Ralph Johnson, and John Vlissides — are recipients of this year's annual award that honors achievement in the world of software development.**



By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	<ul style="list-style-type: none"> <li>Factory Method</li> </ul>	<ul style="list-style-type: none"> <li>Adapter (class)</li> </ul>	<ul style="list-style-type: none"> <li>Interpreter</li> <li>Template Method</li> </ul>
	Object	<ul style="list-style-type: none"> <li>Abstract Factory</li> <li>Builder</li> <li>Prototype</li> <li>Singleton</li> </ul>	<ul style="list-style-type: none"> <li>Adapter (object)</li> <li>Bridge</li> <li>Composite</li> <li>Decorator</li> <li>Façade</li> <li>Flyweight</li> <li>Proxy</li> </ul>	<ul style="list-style-type: none"> <li>Chain of Responsibility</li> <li>Command</li> <li>Iterator</li> <li>Mediator</li> <li>Memento</li> <li>Observer</li> <li>State</li> <li>Strategy</li> <li>Visitor</li> </ul>



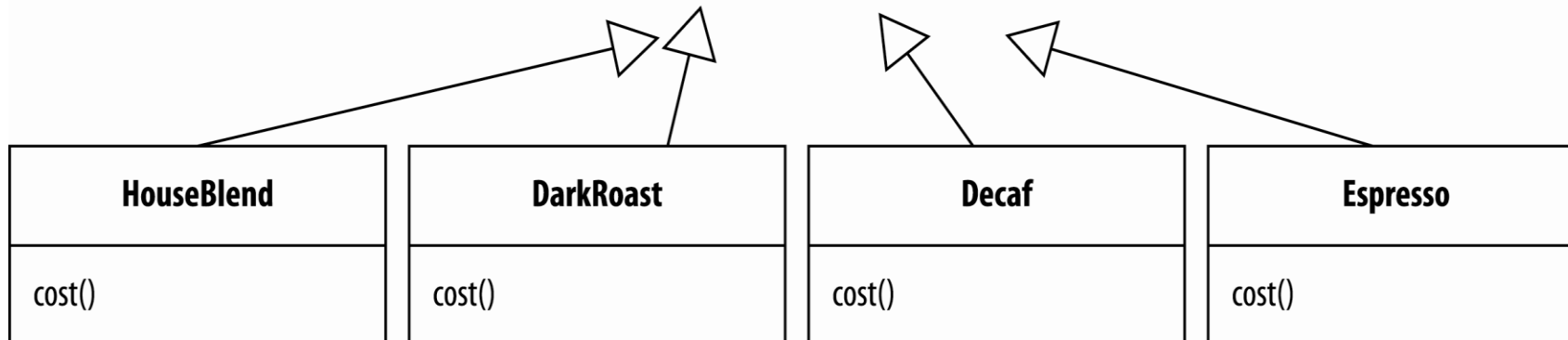
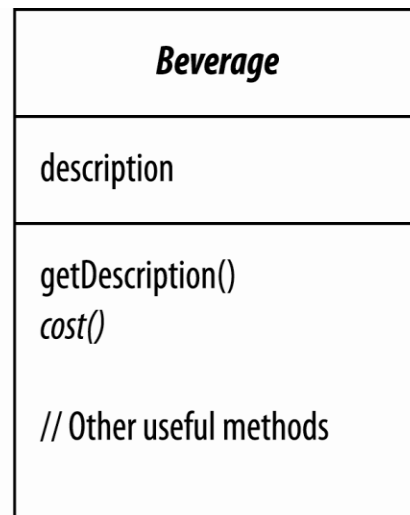
# The Decorator Pattern

*Composition over inheritance*



# Case Study: *Starbuzz PoS Software System* (a first design)

Starbuzz Coffee:  
Their first  
computer system  
implemented  
“beverages” as  
shown.



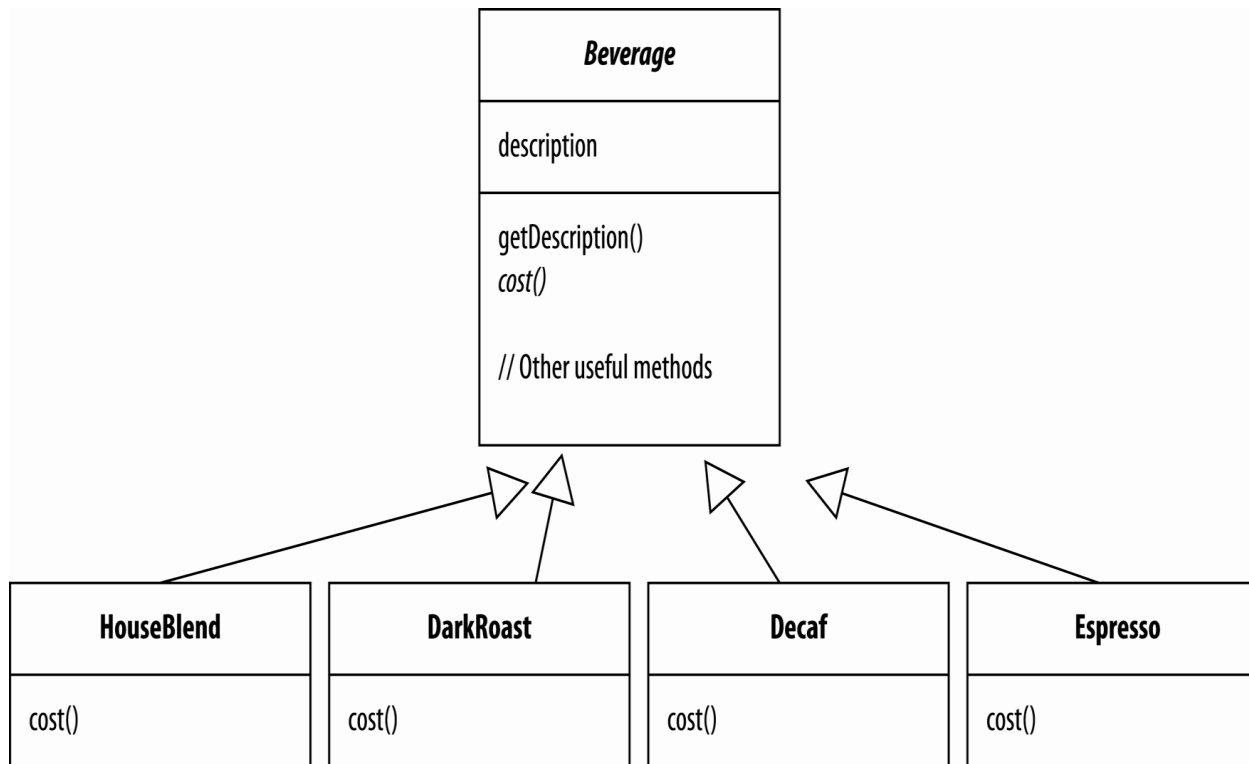
# *Changing business requirements... charge for condiments!*



Syrup: vanilla, caramel, hazecream, extra shot, soy, milk...

Foamy, wet, flat, dry, whipped shot, raspberry, irish cream...

# How to extend the Design?

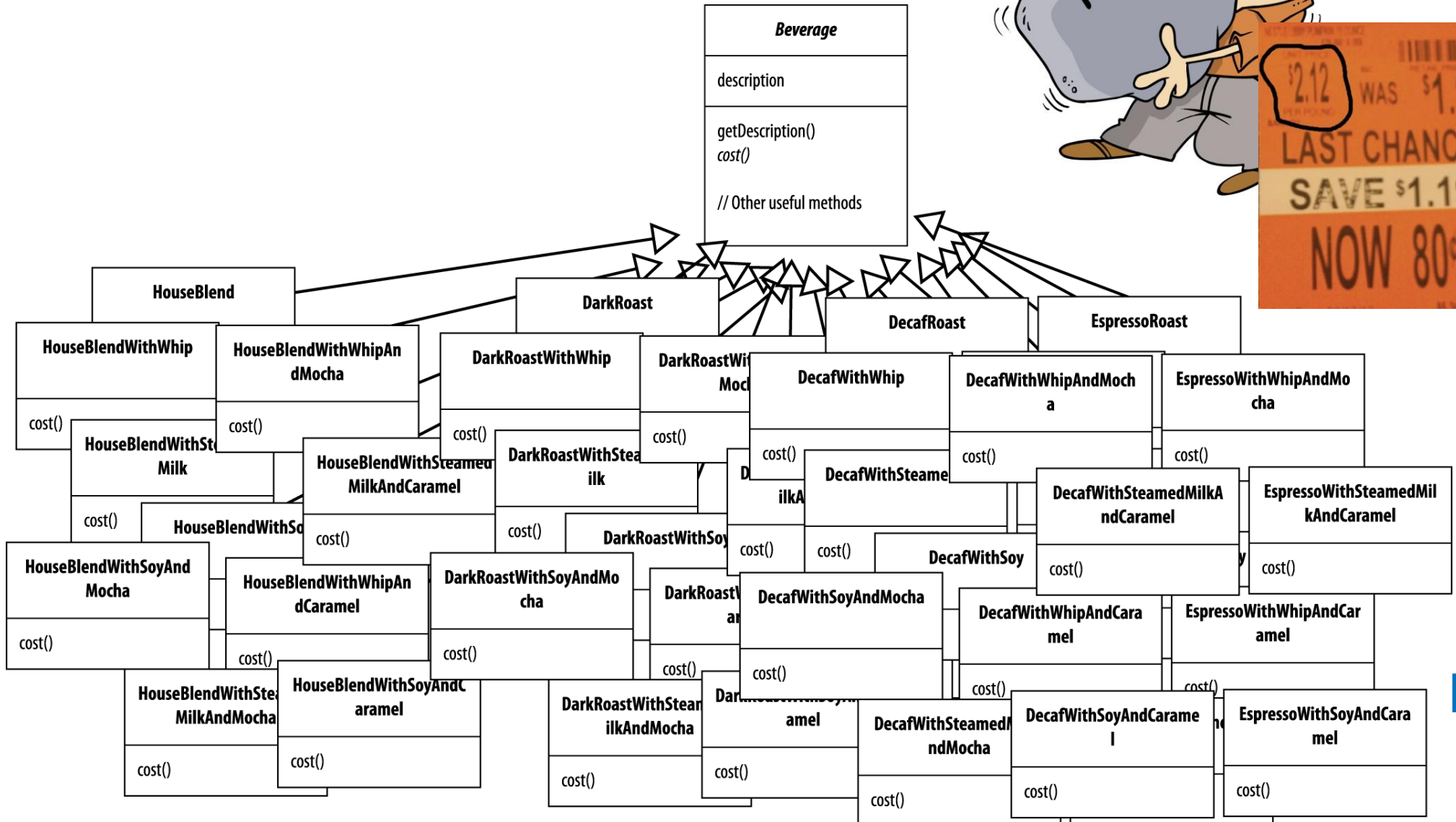


# More specialization?

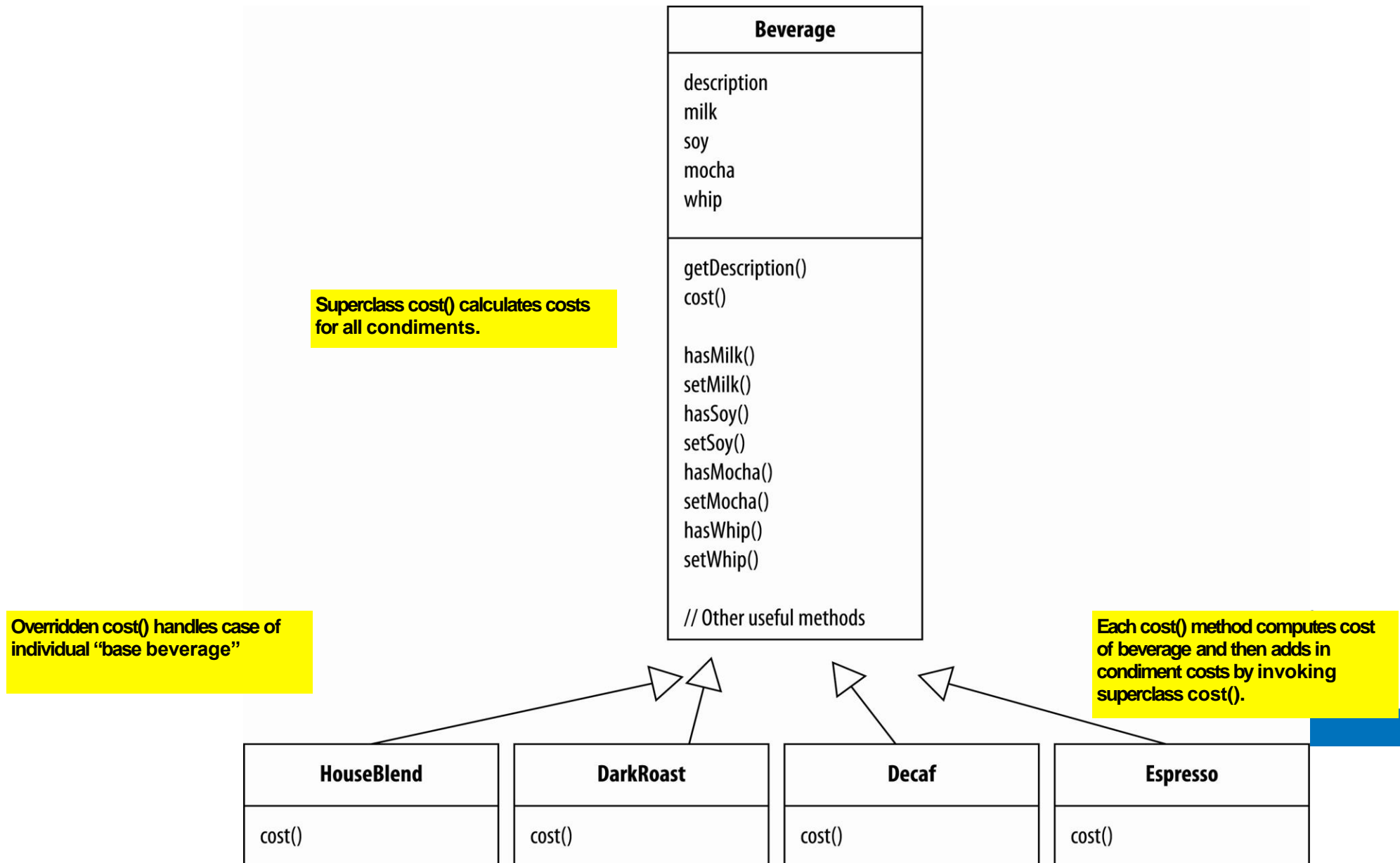
## Starbuzz v.2



2.12 WAS \$1.99  
LAST CHANCE  
SAVE \$1.19  
NOW 80¢



# A better solution? Starbuzz v.3



# Considerations

- What happens when condiment costs change?
- What happens when new condiments are added?
- What happens when we have new beverages for which condiments are inappropriate?
- What if a customer wants triple whip? Double mocha?



# Inheritance can be problematic

- Issue: behaviour inherited by subclassing is static
  - i.e., happens at compile time

**OO Design Principle: *Classes should be open for extension, but closed for modification***

**OO Design Principle: *Favour composition over inheritance***







# Decorator Pattern

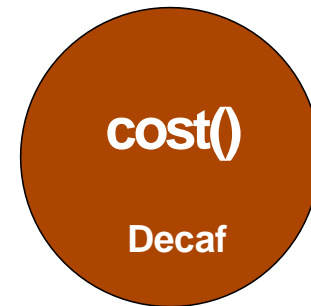
**Example:** Start with a beverage...

- ... and then dynamically (i.e, "at run time") decorate it with condiment (i.e., at runtime).
- Example: Decaf double Mocha with Whip.



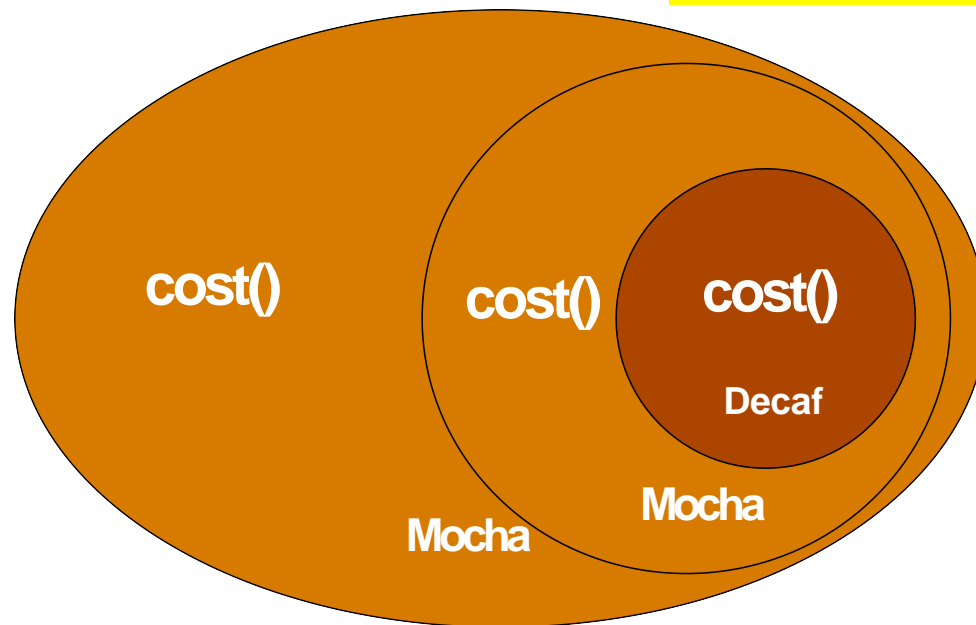
# Step 1: Start with Decaf object

**Decaf will inherit from Beverage so it will have a `cost()` method.**



# Steps 2 & 3: Customer wants double-mocha

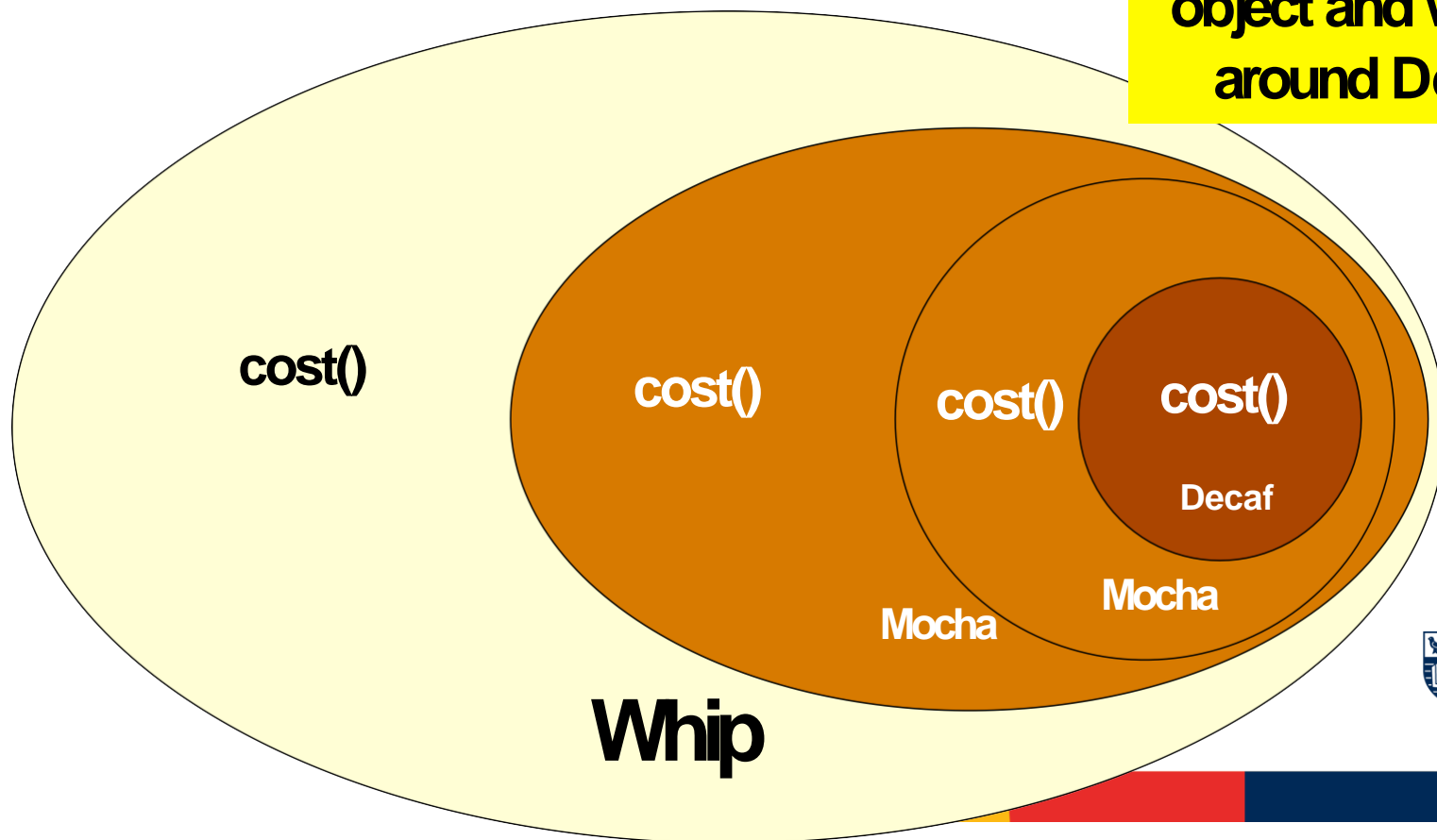
We create a Mocha object and wrap it around Decaf



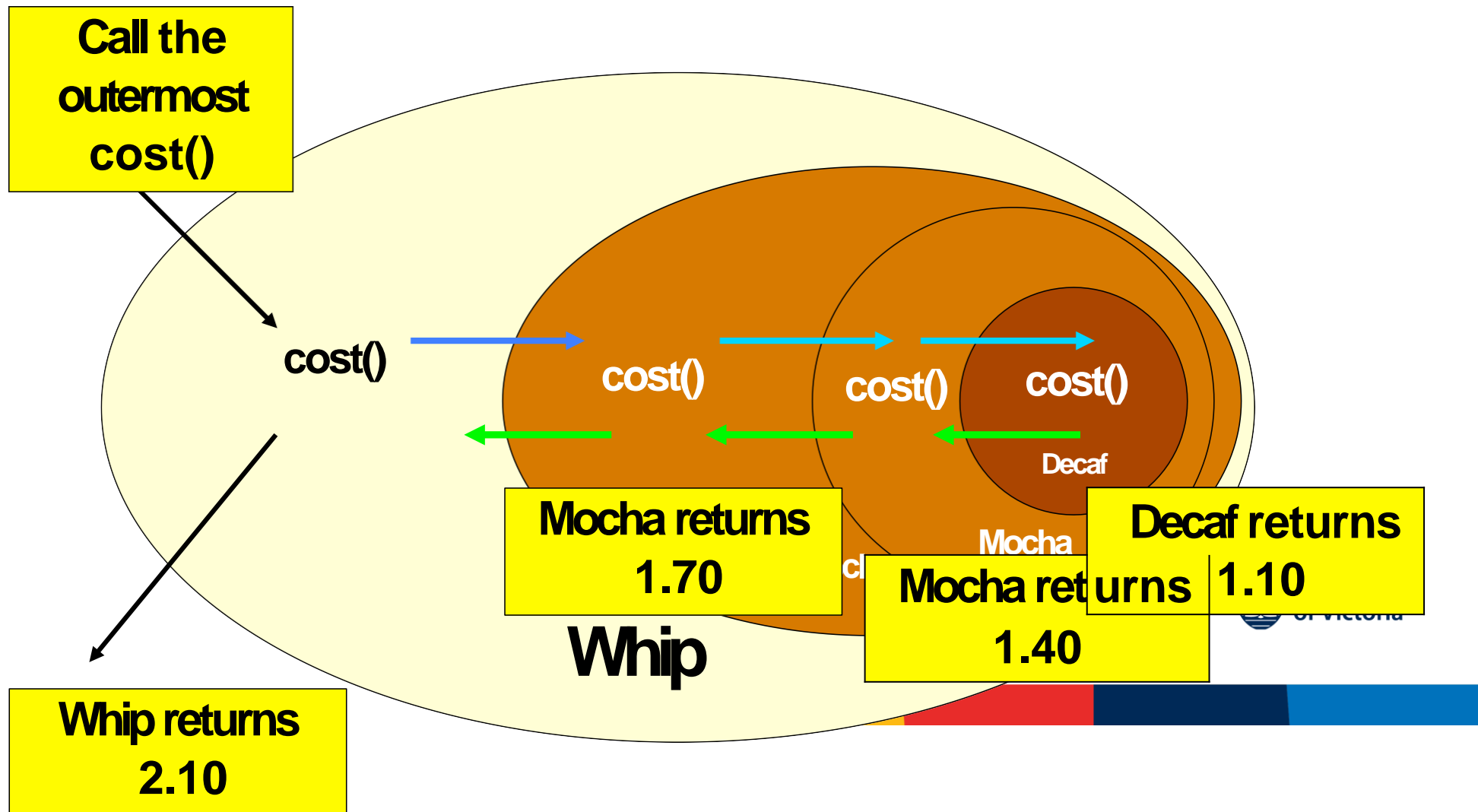
And then another around the first Mocha.

# Step 4: Add the whip

We create a Whip object and wrap it around Decaf



# Calculating the costs



# Decorator Pattern defined

**The Decorator Pattern attaches additional responsibilities to an object dynamically.**

**Decorators provide a flexible alternative to subclassing for extending functionality.**

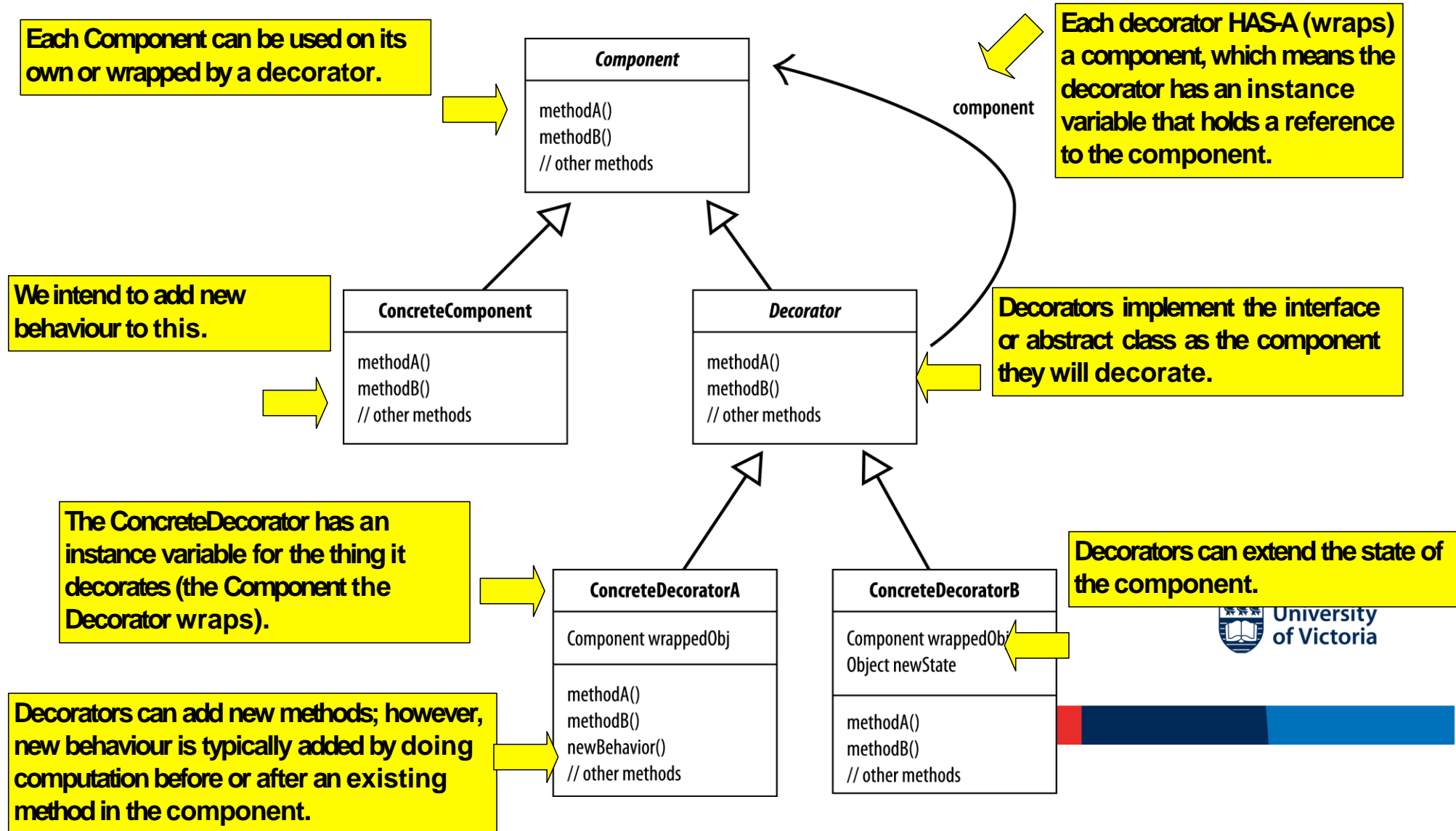


# Facts about Decorators

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- We can pass around a decorated object in place of the original (i.e., wrapped) object.
- The decorator adds its own behaviour before or after (or both!), & delegates to the object it decorates the rest of the job.
- Objects can be decorated at any time.



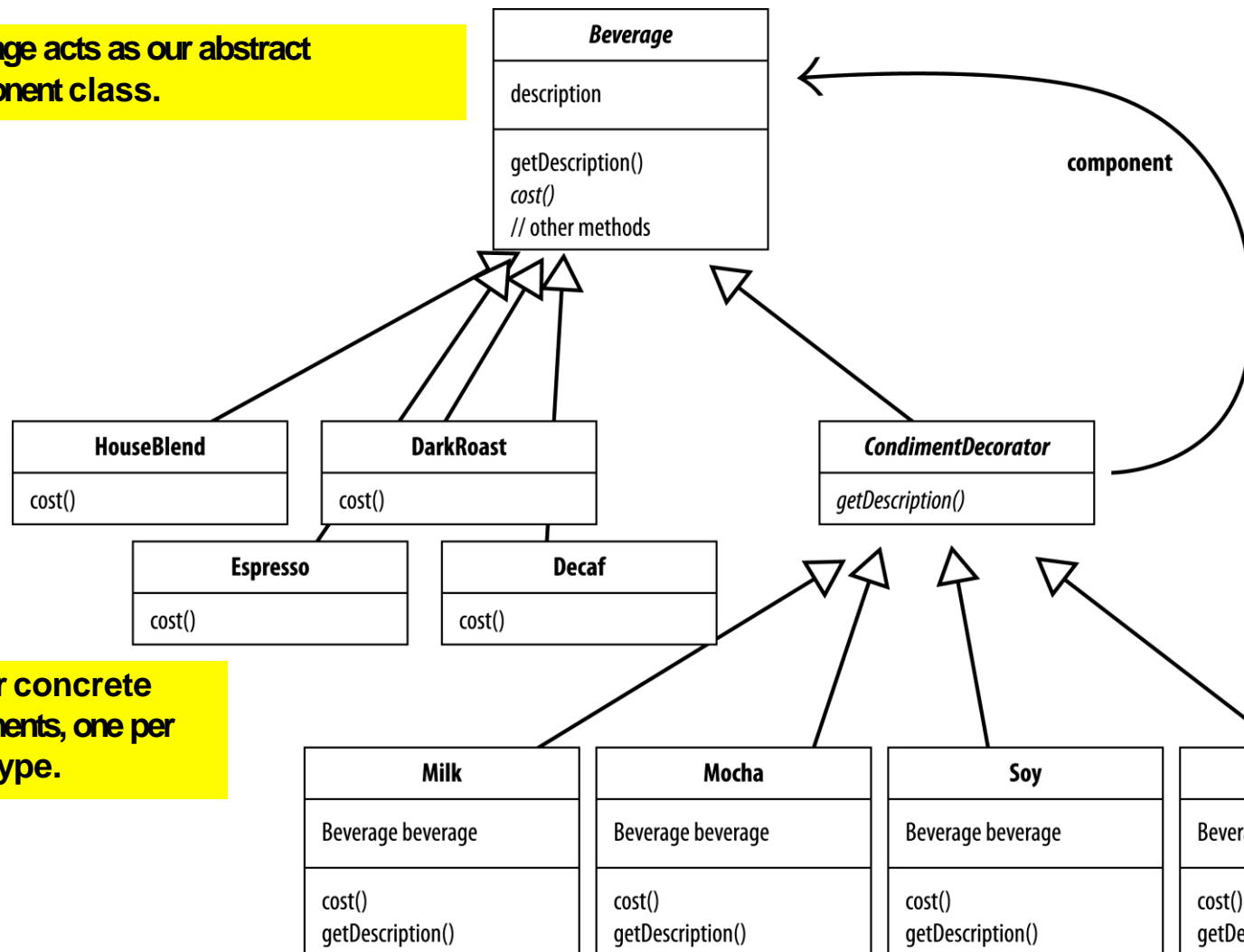
# Decorator Pattern in its general form





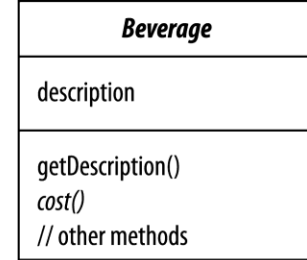
# Now applied to Starbuzz (v.3)

Beverage acts as our abstract component class.



The four concrete components, one per coffee type.

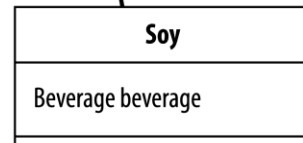
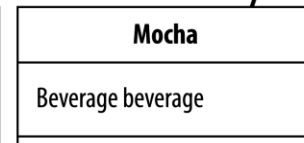
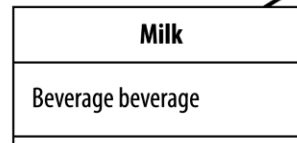
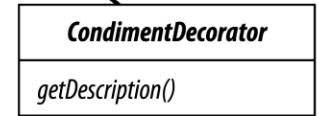
Condiment decorators; note which methods they need to implement – both `cost()` and `getDescription()`.



```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```



```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```

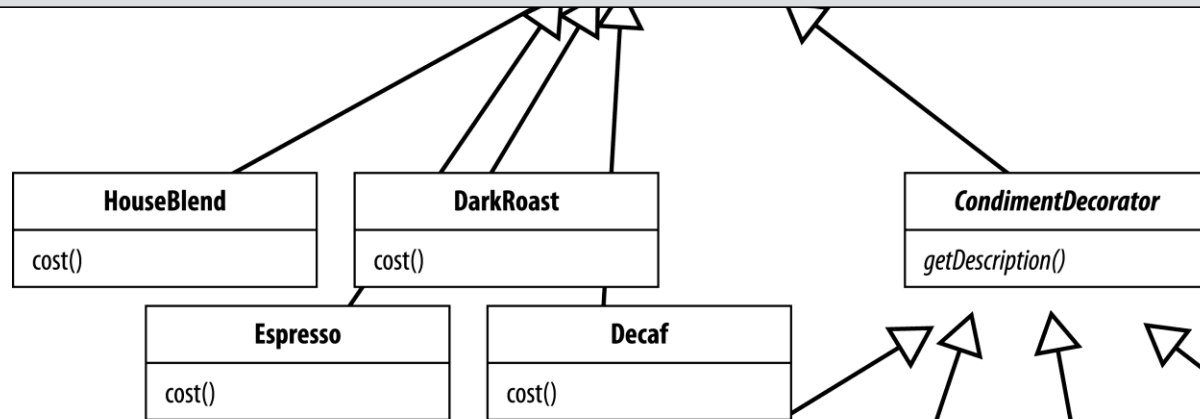


```

public class Decaf extends Beverage {
    public Decaf() {
        description = "Decaf relaxant";
    }

    public double cost() {
        return 1.10;
    }
}

```



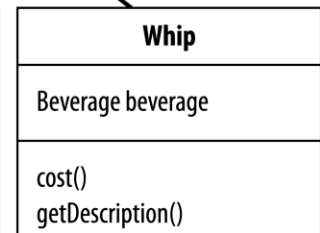
```

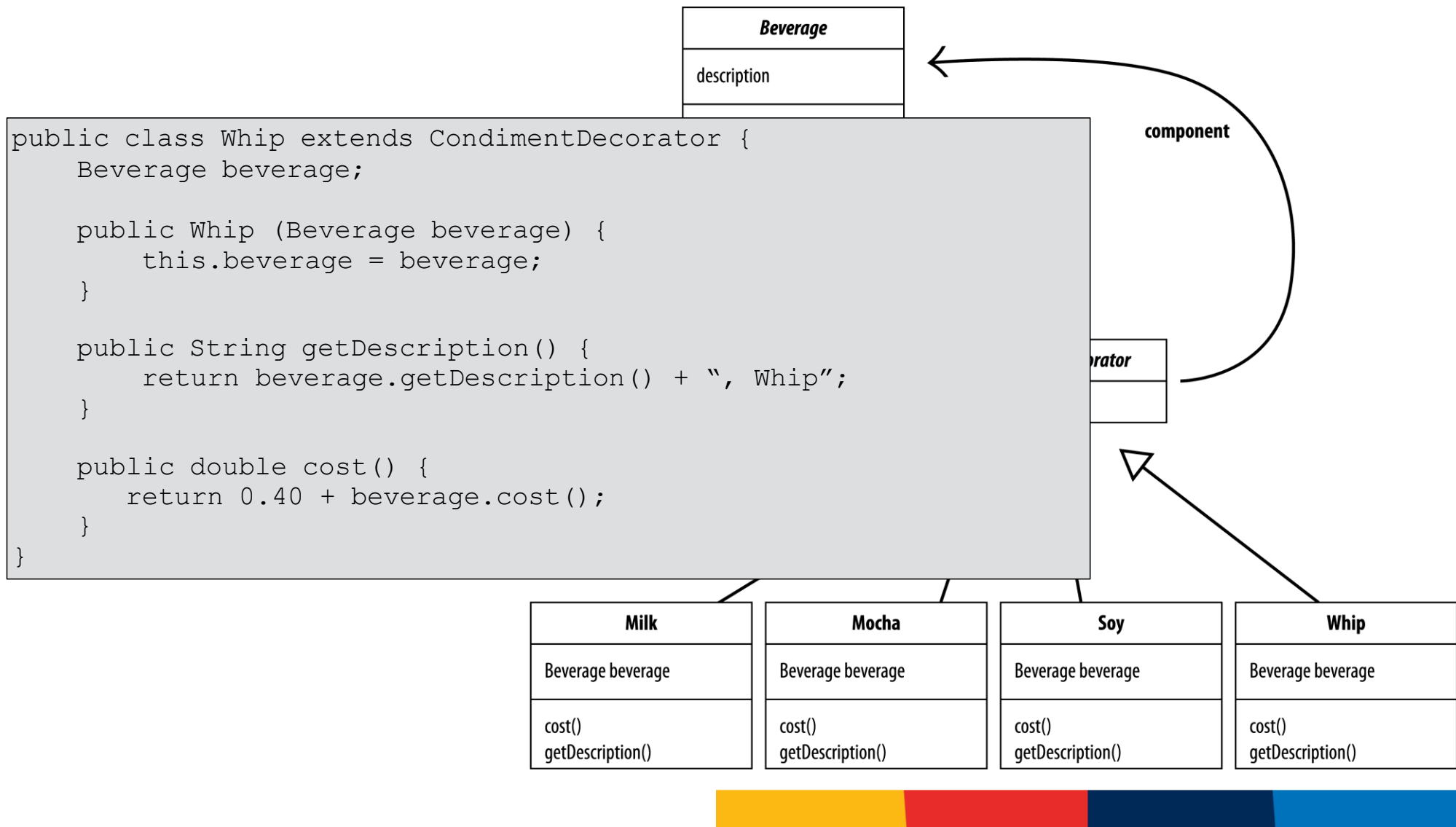
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "HouseBlend";
    }

    public double cost() {
        return 1.00;
    }
}

```

ent





- **Example use:** Composing the following drinks:
  - Ordinary Espresso.
  - Decaf with double Mocha, wrapped in whip.
  - HouseBlend with Soy, Mocha and Whip.



```

public class UVicKaffeeHaus {
    public static void main (String args[]) {
        Beverage beverage = new Espresso();
        System.out.println (beverage.getDescription() + " $" + beverage.cost());

        Beverage beverage2 = new Decaf();
        beverage2 = new Mocha (beverage2);
        beverage2 = new Mocha (beverage2);
        beverage2 = new Whip (beverage2);
        System.out.println (beverage2.getDescription() + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy (beverage3);
        beverage3 = new Mocha (beverage3);
        beverage3 = new Whip (beverage3);
        System.out.println (beverage3.getDescription() + " $" + beverage3.cost());
    }
}

```

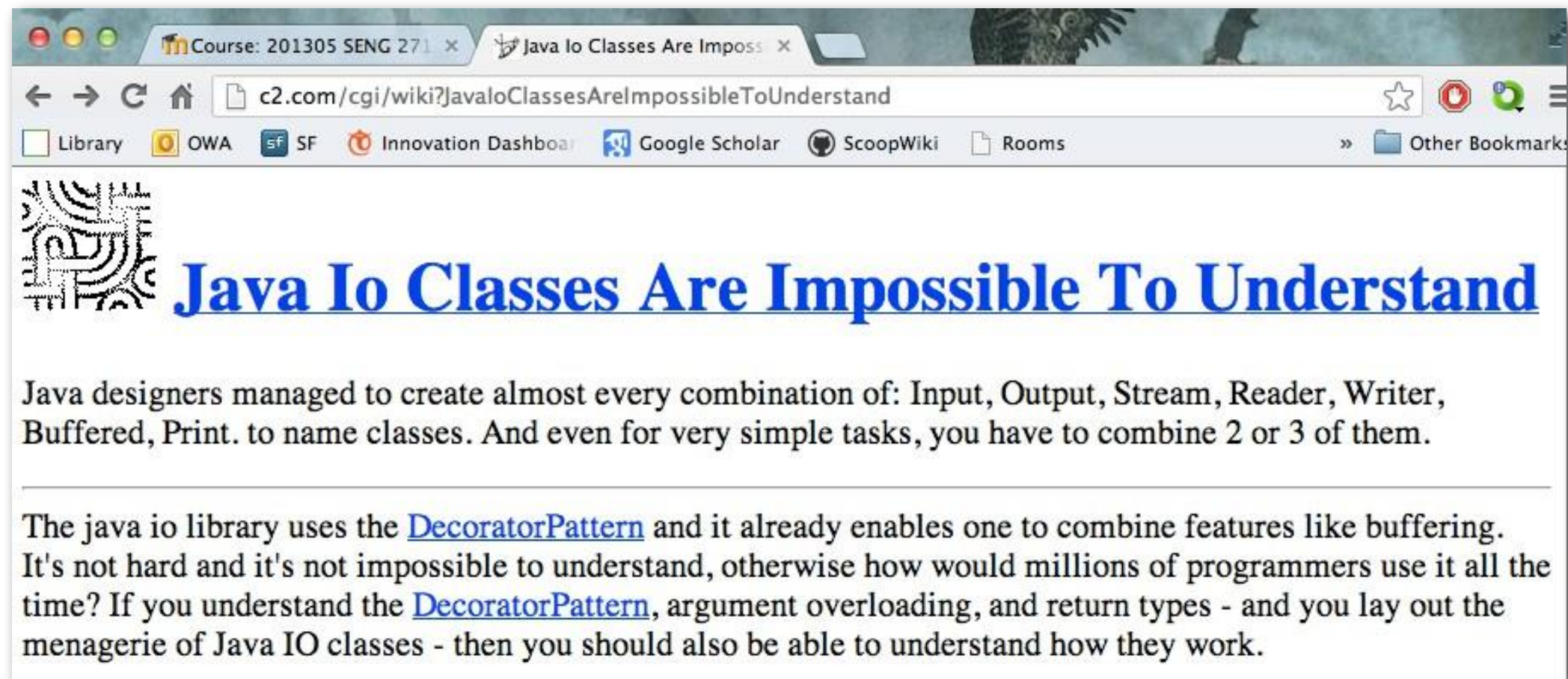
**Espresso**

**Decaf double Mocha with  
whip**

**HouseBlend with Soy,  
Mocha and Whip**



# The Decorator Pattern in Frameworks: Example



The screenshot shows a web browser window with two tabs: 'Course: 201305 SENG 271' and 'Java Io Classes Are Impossible To Understand'. The address bar shows the URL 'c2.com/cgi/wiki?JavaIoClassesAreImpossibleToUnderstand'. The browser's bookmark bar includes 'Library', 'OWA', 'SF', 'Innovation Dashboard', 'Google Scholar', 'ScoopWiki', and 'Rooms'. The page content features a decorative logo on the left and a main heading 'Java Io Classes Are Impossible To Understand' in blue. Below the heading, the text explains the complexity of Java IO classes and mentions the Decorator Pattern.

## Java Io Classes Are Impossible To Understand

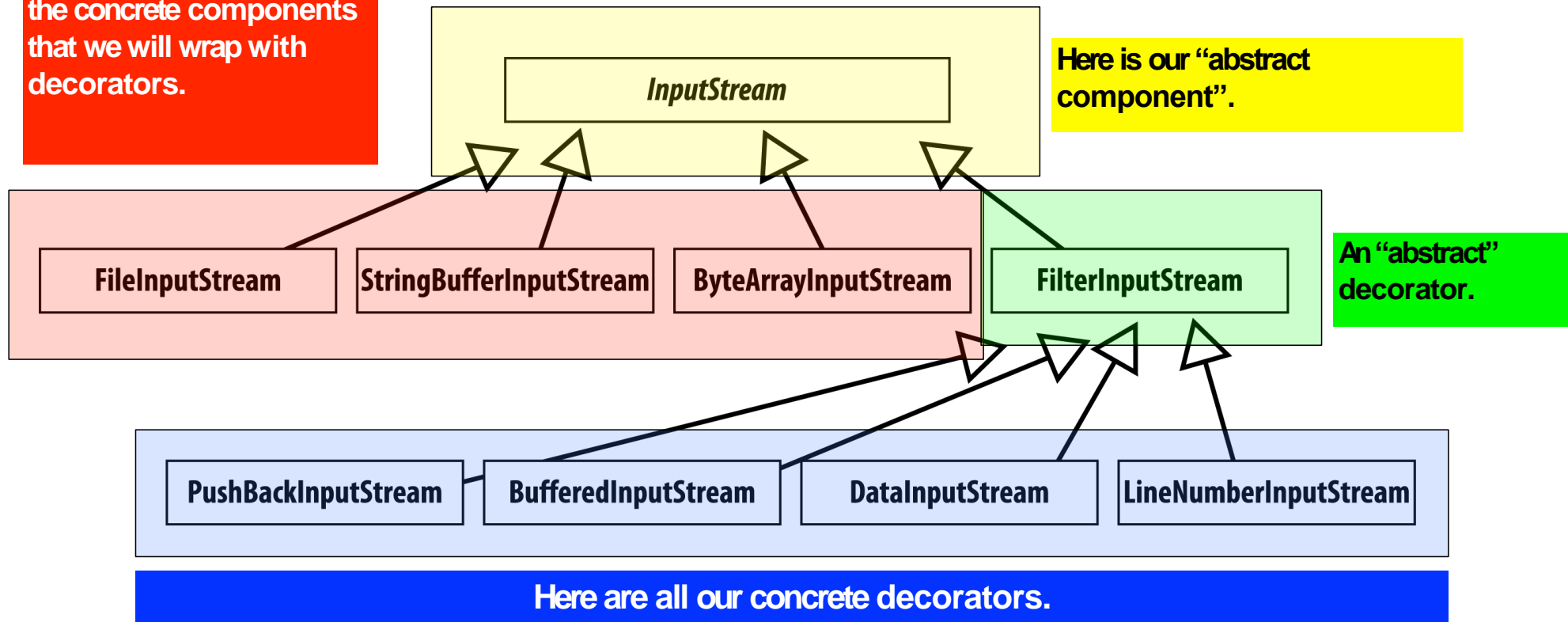
Java designers managed to create almost every combination of: Input, Output, Stream, Reader, Writer, Buffered, Print. to name classes. And even for very simple tasks, you have to combine 2 or 3 of them.

---

The java io library uses the [DecoratorPattern](#) and it already enables one to combine features like buffering. It's not hard and it's not impossible to understand, otherwise how would millions of programmers use it all the time? If you understand the [DecoratorPattern](#), argument overloading, and return types - and you lay out the menagerie of Java IO classes - then you should also be able to understand how they work.

# The Decorator Pattern in JavaIO

These input streams act as the concrete components that we will wrap with decorators.





# Using Java IO Decorators

```
public static void writeData (double[] data, String file)
    throws IOException
{
    OutputStream fout = new FileOutputStream (file);
    DataOutputStream out = new DataOutputStream(fout);
    out.writeInt(data.length);
    for (double d : data) {
        out.writeDouble(d);
    }
    out.close();
}
```

**Concrete component**

**Concrete decorator**



# Extending Java IO Decorators

**Example:** A decorator that converts all uppercase characters to lowercase in the input stream.



# New Decorator code

```
public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }
    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = super.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

# Using the new Decorator

```
public class InputTest {  
    public static void main(String[] args) throws IOException {  
        int c;  
        try {  
            InputStream in =  
                new LowerCaseInputStream(  
                    new BufferedInputStream(  
                        new FileInputStream("test.txt"))));  
            while((c = in.read()) >= 0) {  
                System.out.print((char)c);  
            }  
            in.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

# In-class Activity

- 10 minutes
- Find another example of a Decorator Pattern like the starbuzz.
- Implement the Decorator Pattern onto that example.
- Submit to Week 5 on Teams

