HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

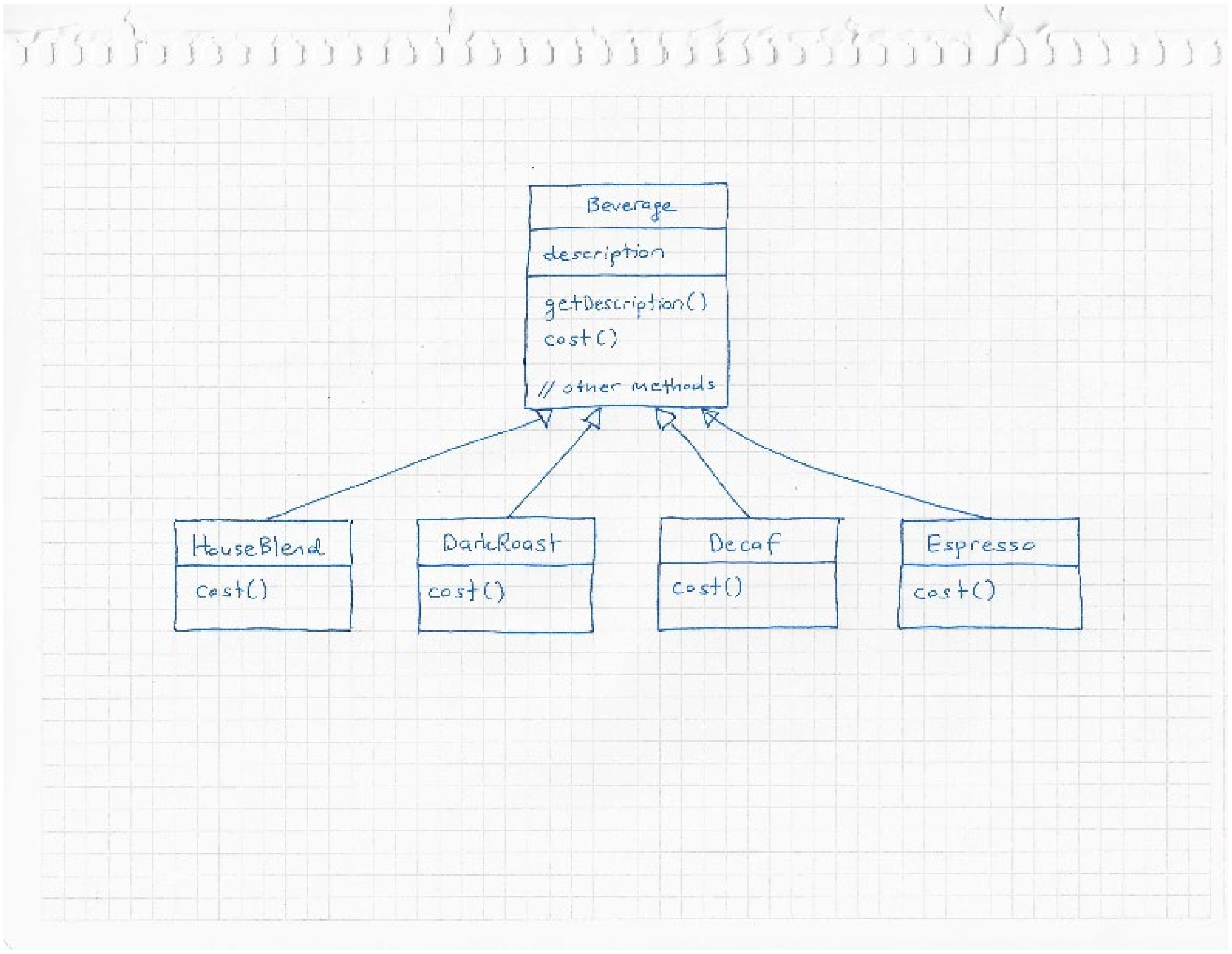
BBM 486 – DESIGN PATTERNS

4. THE DECORATOR PATTERN

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

We will illustrate the decorator pattern with as example. We are updating the order system of the fastest growing Starbuzz Coffee shop to match their beverage offerings.

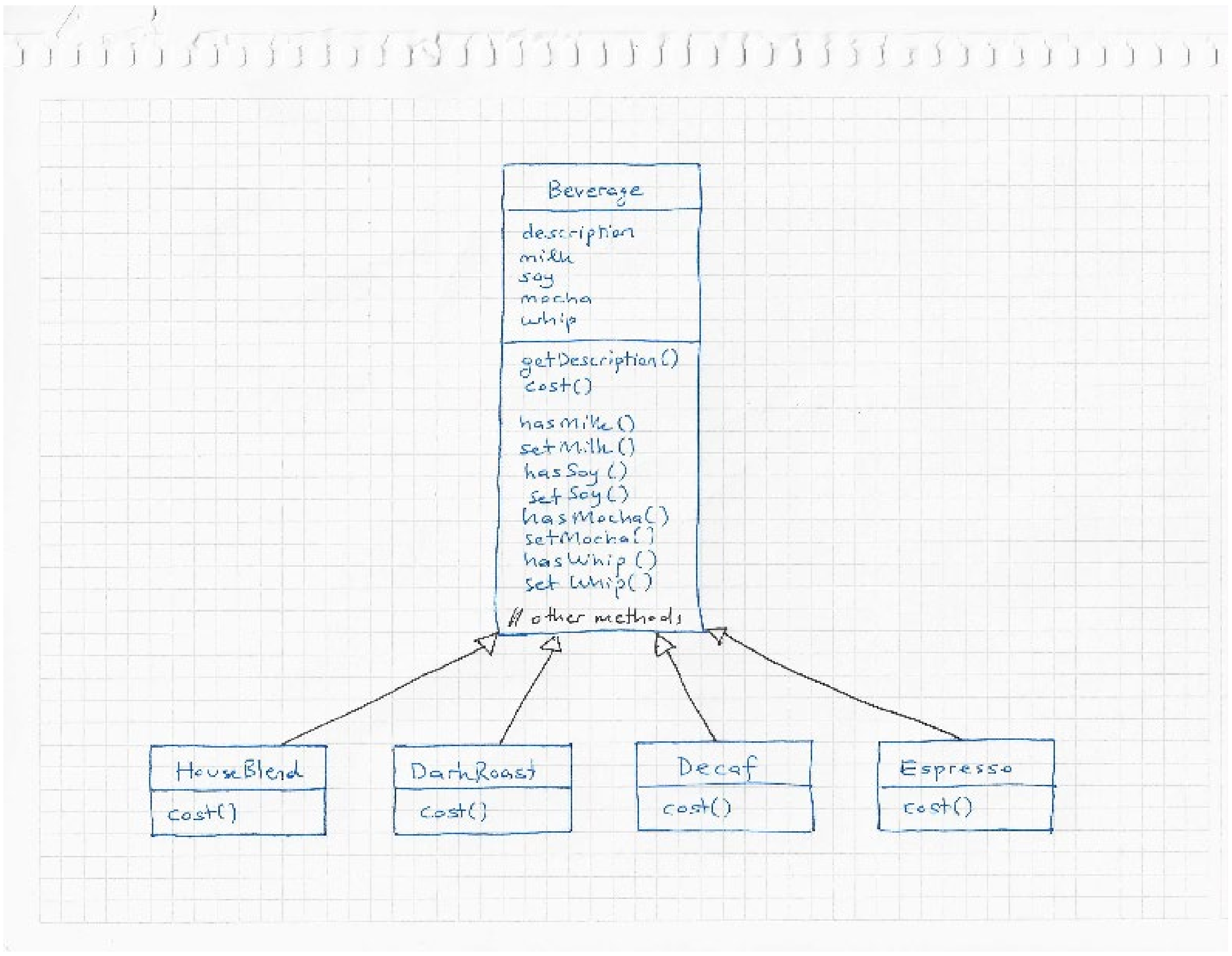
1. In the initial design of their system Beverage was an abstract class, which was subclassed by all beverages offered in the coffee shop: HouseBlend, DarkRoast, Decaf and Espresso. The description instance variable is set in each subclass. The cost() method is abstract; subclasses need to define their own implementation to return the cost of the beverage.



2. In addition to your coffee, you can also ask for several condiments like steamed milk, soy and mocha, and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system. This caused a class explosion, which created a maintenance nightmare. What happens when the price of milk goes up? What do they do when they add a new caramel topping?

HouseBlendWithSteamedMilkandMocha	HouseBlendWithWhipandSoy	…
DarkRoastWithSteamedMilkandMocha	DarkRoastWithWhipandSoy	…
DecafWithSteamedMilkandMocha	DecafWithWhipandSoy	…
EspressoWithSteamedMilkandMocha	EspressoWithWhipandSoy	…

3. Can't we just use instance variables and inheritance in the superclass to keep track of the condiments? Start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha and whip. The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.



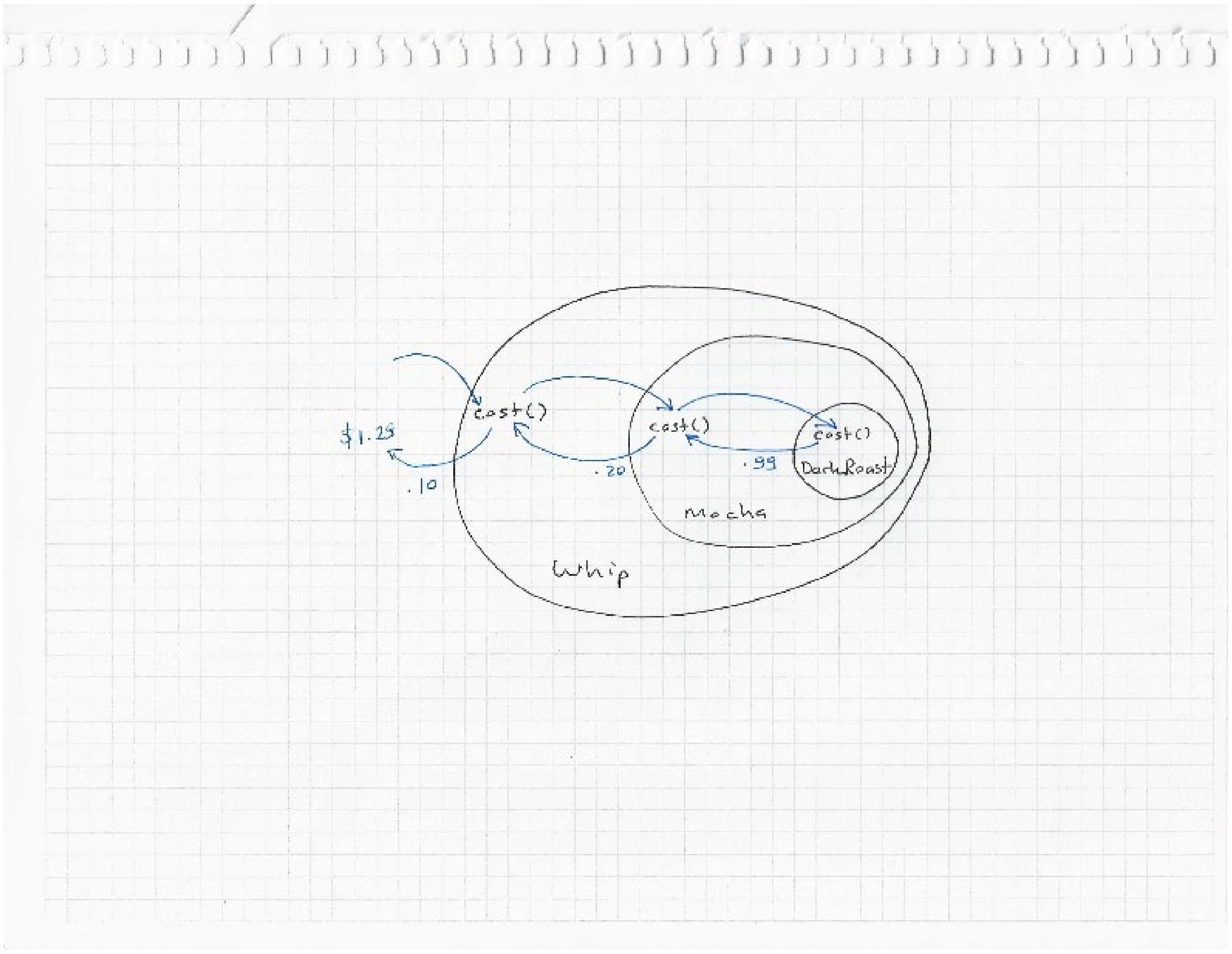
```
public class Beverage {

    // declare instance variables for milkCost, soyCost,
    // mochaCost and whipCost, and getters and
    // setters for milk, soy, mocha and whip.

    public double cost() {
            double condimentCost = 0.0;
            if (hasMilk()) {
                    condimentCost += milkCost;
            }
            if (hasSoy()) {
                    condimentCost += soyCost;
            }
            if (hasMocha()) {
                    condimentCost += mochaCost;
            }
            if (hasWhip()) {
                    condimentCost += whipCost;
            }
            return condimentCost;
    }
}

public class DarkRoast extends Beverage {

    public DarkRoast() {
            description = “Most excellent Dark Roast”;
    }

    public double cost() {
            return 1.99 + super.cost();
    }
}
```

4. Some potential problems with this approach:
    a. Price changes for condiments will force us to alter existing code.
    b. New condiments will force us to add new methods and alter the cost method in the superclass.
    c. We may have new beverages. For some of these beverages (iced tea?), the condiments may not be appropriate yet the Tea subclass will still inherit methods like hasWhip().
    d. What is a customer wants a double mocha?

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

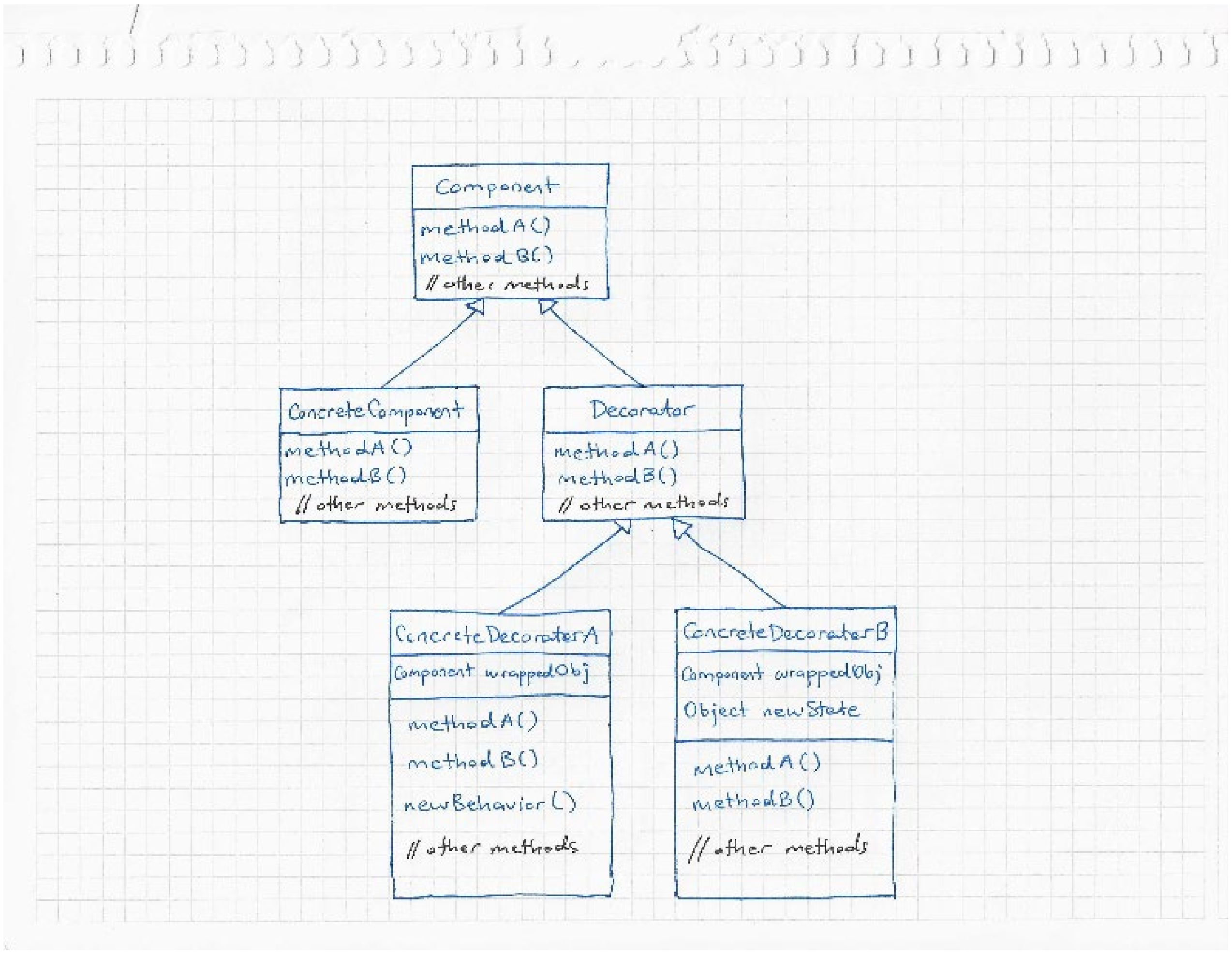
6. Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

7. We will start with a beverage and “decorate” it with the condiments at runtime.

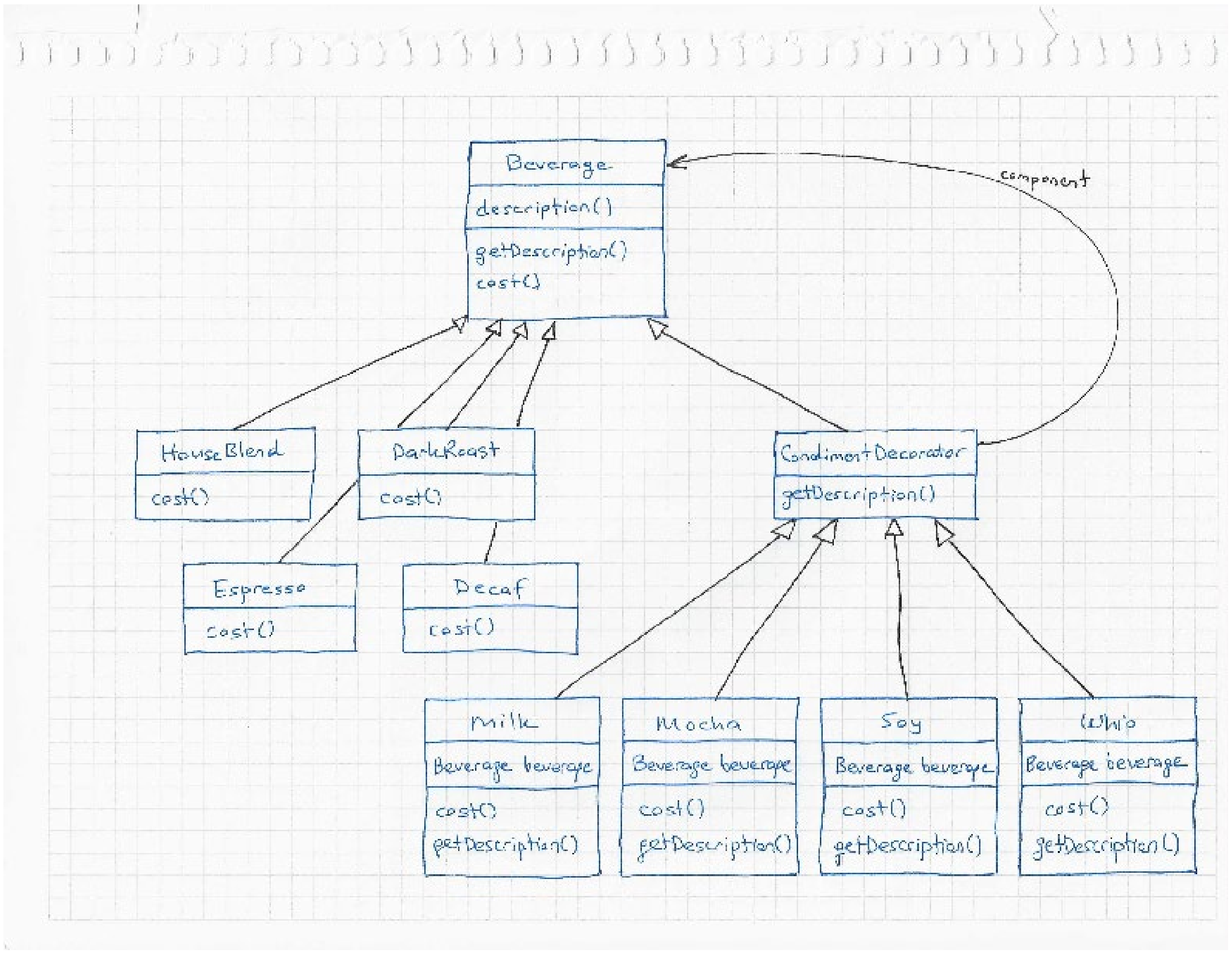


8. Decorators have the same supertype as the objects they decorate. You can use one or more decorators to wrap an object. Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.

9. The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job. Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.



10. Decorating our Beverages



11. Writing the Starbuzz code: Let’s start with the Beverage class, which does not need to change from the original design.

```
public abstract class Beverage {
    String description = “Unknown beverage”;

    public String getDescription () {
            return description;
    }

    public abstract double cost ();
}
```

Let’s implement the abstract class for the Condiments as well:

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

12. Coding beverages:

```
public class Espresso extends Beverage {

    public Espresso () {
            description = “Espresso”;
    }

    public cost () {
            return 1.99;
    }
}

public class HouseBlend extends Beverage {
    public HouseBlend () {
            description = “House Blend Coffee”;
    }

    public double cost () {
            return .89;
    }
}
```

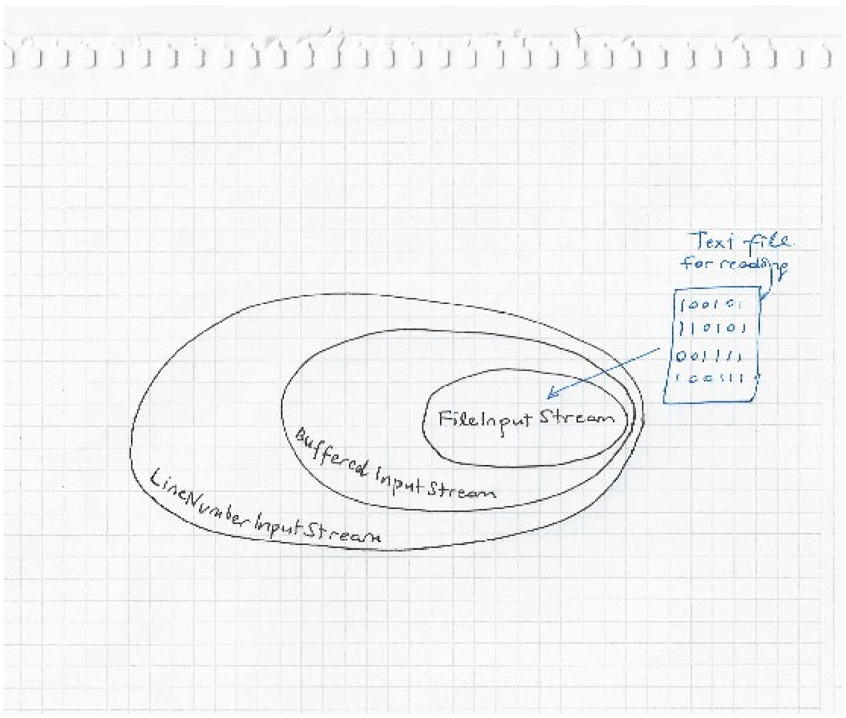
13. Coding condiments:

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

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

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

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

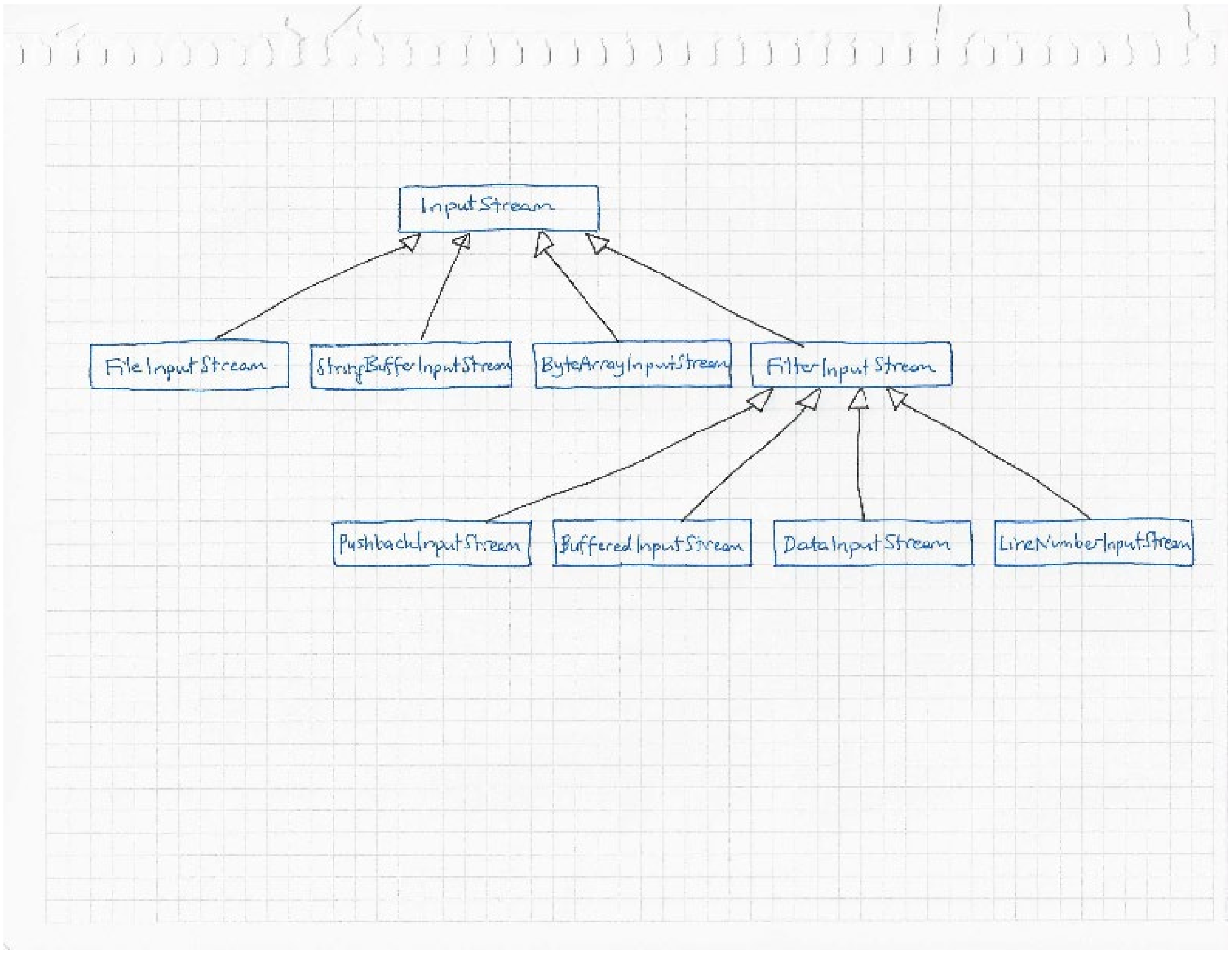
14. Here is a test code to make orders:

```
public class StarbuzzCoffeee {

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

            Beverage beverage2 = new DarkRoast();
            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());
    }
}
```

15. Real World Decorators: Java I/O



16. Decorating the java.io classes



17. Writing your own Java I/O Decorator

```
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;
        }
    }
```

18. Test out the new Java I/O 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();
            }
        }
    }
```