HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

BBM 486 – DESIGN PATTERNS

1. INTRODUCTION TO DESIGN PATTERNS

In this class, we will learn why (and how) you can exploit the wisdom and lessons learned by other developers who have been confronted with the same design problem and developed a satisfactory solution. We will look at some key OO design principles, and walk-through examples of how patterns work. The best way to use patterns is to load your brain with them and then recognize places in your designs and existing applications where you can apply them. Instead of code reuse, with patterns you get experience reuse. *I will try to make this class as practical as possible.*

A Pattern is a solution to a problem in a context.

- The **context** is the situation in which the pattern applies. This should be a recurring situation.
- The **problem** refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context.
- The **solution** is what you are after: a general design that anyone can apply which resolves the goal and set of constraints.

An example context is that you have a collection of objects. The problem is that you need to step through the objects without exposing the collection's implementation. And the solution is to encapsulate the iteration into a separate class.

Patterns are general solutions to recurring problems. They also have the benefit of being well tested by lots of developers. So, when you see a need for one, you can sleep well knowing many developers have been there before and solved the problem using similar techniques.

Design patterns had originally been categorized into the following 3 sub-classifications based on kind of problem they solve.

1.1. Creational Patterns

Creational patterns provide the capability to create objects based on a required criterion and in a controlled way. They deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or in added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation. Creational design patterns are composed of two dominant ideas. One is encapsulating knowledge about which concrete classes the system uses. Another is hiding how instances of these concrete classes are created and combined.

Creational design patterns are further categorized into object-creational patterns and Class-creational patterns. In greater details, Object-creational patterns defer part of its object creation to another object, while Class-creational patterns defer its object creation to subclasses. Well-known design patterns that are parts of creational patterns are:

- Factory Method: Define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Singleton Pattern: Ensure that a class has only one instance, and provide a global point of access to it.
- Abstract factory pattern, which provides an interface for creating related or dependent objects without specifying the objects' concrete classes.

1.2. Structural Patterns

Structural patterns are about organizing different classes and objects to form larger structures and provide new functionality. They ease the design by identifying a simple way to realize relationships among entities.

- Decorator Pattern: Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
- Adapter Pattern: Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces.
- Façade Patterns: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Composite Pattern: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Proxy Pattern: Provide a surrogate or placeholder for another object to control access to it.

1.3. Behavioral Patterns

Behavioral patterns are about identifying common communication patterns between objects and realize these patterns. They identify common communication patterns among objects and distribute responsibility. By doing so, these patterns increase flexibility in carrying out communication.

- Strategy Pattern: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Observer Pattern: Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
- Command Pattern: Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.
- Template Method Pattern: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Iterator Pattern: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- State Pattern: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

1.4. Class Patterns versus Object Patterns

Patterns are often classified by a second attribute of whether or not the pattern deals with classes or objects.

Class patterns describe how relationships between classes are defined via inheritance. Relationships at class patterns are established at compile time. Template Method, Adapter, and Factory Patterns fall into this category.

Object patterns describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible. Other patterns that we review are part of this category.

HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

BBM 486 – DESIGN PATTERNS

2. THE STRATEGY PATTERN

The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

We will illustrate the strategy pattern with an example. You are building a duck pond simulation game, *SimUDuck.* The game can show a large variety of duck species swimming and making quacking sounds.

1. The initial design of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit: All ducks quack and swim, the superclass takes care of the implementation code. The display() method is abstract, since all duck subtypes look different. Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.

2. We received a request to add a new feature where ducks can fly. We consider adding the fly() method in the Duck class and then all the ducks will inherit it. This is a good use of inheritance for the purpose of reuse.

3. We however noticed that not all subclasses of duck can fly. Rubber duckies for example cannot fly. By putting fly() in the superclass, we gave flying ability to all ducks, including those that should not. Rubber ducks pose another problem that they don't quack. So we overrode them to squeak.

4. We thought about overriding the fly() method (to do nothing) in rubber duck, the way we did with the quack() method. But then what happens when we add wooden decoy ducks to the program? They are not supposed to fly or quack. Can that be another class in the hierarchy where we override both quack() and fly() to do nothing?

5. We realize that inheritance probably is not the answer, because we received a requirement that the product features will be updated every six moths. Since the spec will keep changing, we will be forced to look and possibly override fly() and quack() for every new Duck subclass that is ever added to the program.

6. How about an interface? We can take the fly() out of the Duck superclass, and make a Flyable() interface with a fly() method. That way, only the ducks that are supposed to fly will implement that interface and have a fly() method, and we might as well make a Quackable() too, since not all ducks can quack.

7. However, while having the subclasses implement Flyable and/or Quackable solves part of the problem (no inappropriately flying rubber ducks), it completely destroys code reuse for those

behaviors, so it just creates a different maintenance nightmare. Moreover, there might be more than one kind of flying behavior even among the ducks that do fly.

8. The one constant in software development. No matter where you work, what you are building, or what language you are programming in, what's the one true constant that will be with you always? CHANGE.

9. Lots of things can drive change:
   Customers or users may decide they want something else or they want new functionality.
   Company may decide to go with another database vendor and purchase its data from another supplier that uses a different data format.
   Technology may change and we have got to update our code to make use of protocols.
   We may have learned enough building our system that we'd like to go back and do things a little better.

10. We know using inheritance has not worked out well, since the duck behavior keeps changing across subclasses, and it's not appropriate for all subclasses to have those behaviors. The Flyable and Quackable interface sounded promising at first, except **Java interfaces have no implementation code, so no code reuse**. That means whenever you need to modify a behavior, you are forced to track down and change it in all the different subclasses where that behavior is defined, probably introducing new bugs along the way.

11. **Design Principle**: Identify the aspects of your application that vary and separate them from what stays the same.

12. Take the parts that vary and "encapsulate" them, so that later you can alter or extend the parts that vary without affecting those that don't. We know that fly() and quack() are the parts of the Duck class that vary across ducks. To separate these behaviors from the Duck class, we will pull both methods out of the Duck class and create a new set of classes to represent each behavior.

13. **Design Principle**: Program to an interface, not an implementation.

14. We'll use an interface to represent each behavior (FlyBehavior and QuackBehavior), and each implementation of a behavior will implement one of those interfaces. So, this time it won't be the Duck classes that will implement the flying and quacking interfaces. Instead, we will make a set of classes whose entire reason for living is to represent a behavior, and it is the behavior class, rather than the Duck class, that will implement the behavior interface.

15. This is in contrast to the way we were doing before, where a behavior either came from a concrete implementation in the superclass Duck, or by providing a specialized implementation in the subclass itself. In both cases, we were locked into using that specific implementation and there was no room to for changing out the behavior other than writing more code.

16. The word interface is overloaded here. There is the concept of interface, but there is also the Java construct `interface`. You can program to an interface, without having to actually use a Java `interface`. The point is to exploit polymorphism by programming to a supertype so that the actual runtime object isn't locked into the code.

17. Here is a simple example of using a polymorphic type – imagine an abstract class Animal, with two concrete implementations, Dog and Cat. **Programming to an implementation** would be:

    ```
    Dog d = new Dog();
    d.bark();
    ```

    But **programming to an interface/supertype** would be:

    ```
    Animal animal = new Dog();
    Animal.makeSound();
    ```

    Even better, rather than hard-coding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime**:

    ```
    a = getAnimal();
    a.makeSound();
    ```

18. With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes. And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that use flying behaviors.

19. Integrating the Duck Behavior: The key is that a Duck will now delegate its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).
    - **First, we will add two instance variables** to the Duck class called flyBehavior and QuackBehavior, that are declared as the interface type. Each duck object will set these variables polymorphically to reference the specific behavior type it would like at runtime (FlyWithWings, Squeak, etc.).
      We also remove the fly() and quack() methods from the Duck class, because we have moved this behavior out into the FlyBehavior and QuackBehavior classes.
      We will replace fly() and quack() in the Duck class with two similar methods, called performFly() and performQuack().

    - **Now we implement performQuack()**:
      ```
      public class Duck {
          QuackBehavior quackBehavior;
          // more
          Public void performQuack() {
                  quackBehavior.quack();
          }
      }
      ```

- Time to worry about **how the flyBehavior and quackBehavior instance variables are set**:

```
public class MallardDuck extends Duck {

    public MallardDuck() {
            quackBehavior = new Quack();
            flyBehavior = new FlyWithWings();
    }

    public void display() {
            System.out.println("I'm a real Mallard duck");
    }
}
```

When a MallardDuck is instantiated, its constructor initializes the MallardDuck's inherited quackBehavior instance variable to a new instance of type Quack. And the same is true for the duck's flying behavior.

20. Testing the Duck code
    - **Type and compile the Duck class below (Duck.java), and the MalardDuck class (MallardDuck.java).**

```
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {
    }

    public abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

- **Type and compile the FlyBehavior interface (FlyBehavior.java), and the two behavior implementation classes (FlyWithWings.java and FlyNoWay.java).**

```
public interface FlyBehavior {
	public void fly();
}

public class FlyWithWings implements FlyBehavior {
	public void fly() {
		System.out.println("I'm flying!!");
	}
}

public class FlyNoWay implements FlyBehavior {
	public void fly() {
		System.out.println("I can't fly");
	}
}
```

- **Type and compile the QuackBehavior interface (QuackBehavior.java), and the three behavior implementation classes (Quack.java, MuteQuack.java and Squeak.java).**

```
public interface QuackBehavior {
	public void quack();
}

public class Quack implements QuackBehavior {
	public void quack() {
		System.out.println("Quack");
	}
}

public class MuteQuack implements QuackBehavior {
	public void quack() {
		System.out.println("<< Silence >>");
	}
}

public class Squeak implements QuackBehavior {
	public void quack() {
		System.out.println("Squeak");
	}
}
```

- **Type and compile the test class (MiniDuckSimulator.java).**

```
public class MiniDuckSimulator1 {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
    }
}
```

- **Run the code!**

21. Setting Behavior Dynamically: Imagine you want to set the Duck's behavior type through a setter method on the duck subclass, rather than by instantiating it in the duck's constructor:

    - **Add two new methods to the Duck class:**

```
public void setFlyBehavior (FlyBehavior fb) {
     flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```

- **Make a new Duck type ModelDuck:**

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("I'm a model duck");
    }
}
```

- **Make a new FlyBehavior type FlyRocketPowered:**

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying with a rocket");
    }
}
```

- **Change the test class, add the ModelDuck and make the ModelDuck rocket-enabled:**

```
public class MiniDuckSimulator1 {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();

        Duck model = new ModelDuck();
        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered());
        model.performFly();
    }
}
```

- **Run it!**

22. The Big Picture on encapsulated behaviors

    Diagram in the class

23. HAS-A can be better than IS-A. Each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking. When you put two classes together like this, you are using **composition**. Instead of inheriting their behavior; the ducks get their behavior by being composed with the right behavior object.

24. **Design Principle**: Favor composition over inheritance.

25. Composition gives you a lot more flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime as long as the object you are composing with implements the correct behavior interface.

    Composition is used in many design patterns and you will see a lot more about its advantages and disadvantages throughout the class.

HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

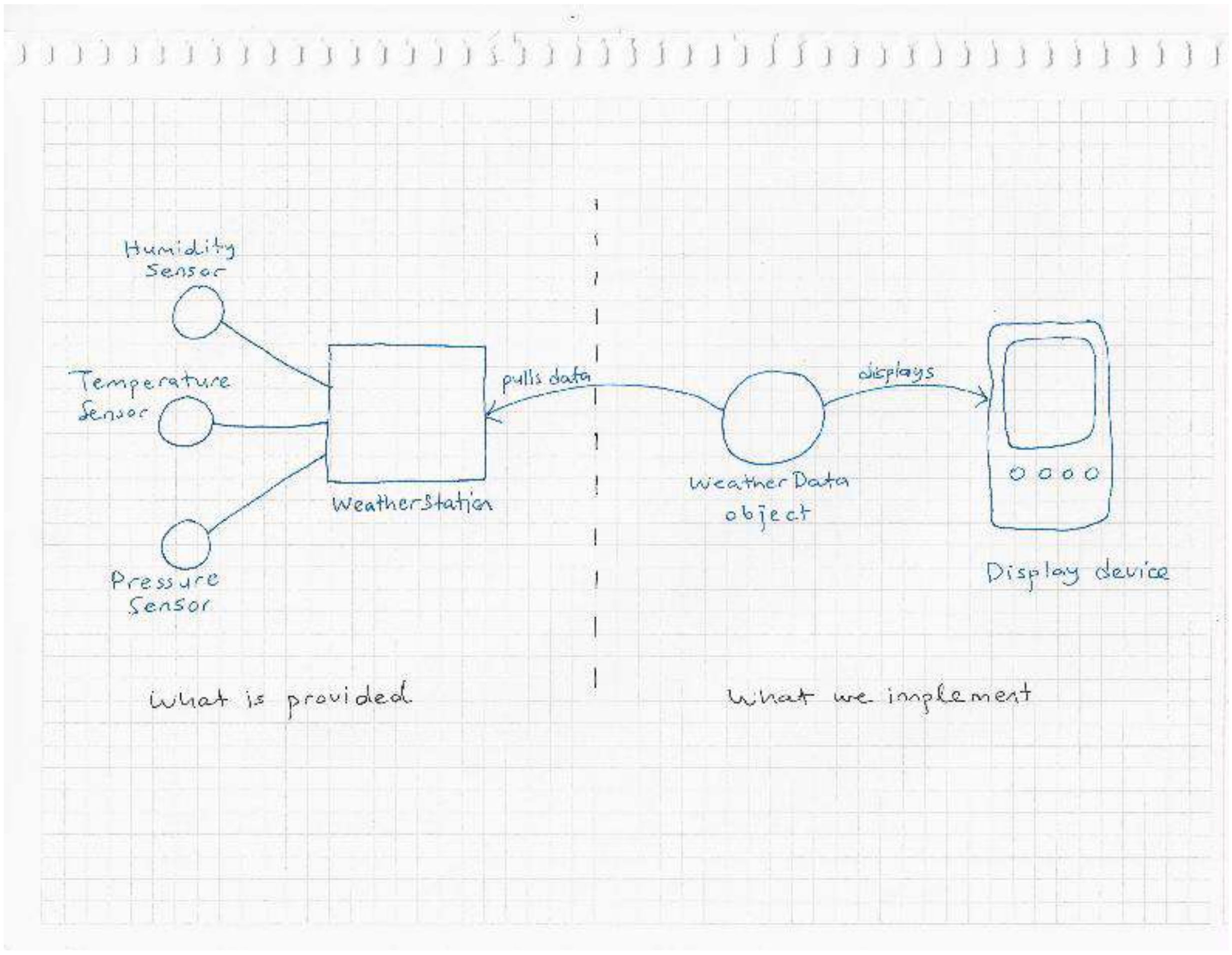
BBM 486 – DESIGN PATTERNS

3. THE OBSERVER PATTERN

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

The observer pattern is one of the most heavily used patterns in the JDK, and it's incredibly useful. We will illustrate the observer pattern with an example. We are building a next generation, Internet-based Weather Monitoring Station, which tracks weather conditions. We would like the weather station to be expandable through an API so that other developers can build their own weather displays.

1. The initial design of the system has three players: the weather station, the WeatherData object that tracks the data coming from the Weather Station and updates the displays, and the display that shows users the current weather conditions. WeatherData class initially had three methods getTemperature(), getHumidity() and getPressure().



2. We are also given a measurementsChanged() method, which gets called whenever the weather measurements have been updated. Our job is to implement this method so that it updates the three displays for current conditions, weather stats, and forecast. We can add the calls to update the three displays into this method, but by doing so we have no way to add or remove other display elements without making changes to the program.

3. What do we know so far?
   - The WeatherData class has getter methods for three measurement values: temperature, humidity and barometric pressure:
     getTemperature()
     getHumidity
     getPressure()
   - The measurementsChanged() method is called anytime new weather measurement data is available. We do not know or care how this method is called; we just know that it is.
   - We need to implement three display elements that use the weather data: a current conditions display, a statistics display, and a forecast display. These displays must be updated each time WeatherData has new measurements.
   - The system must be expandable – other developers can create new custom display elements and users can add or remove as many display elements as they want to the application.

4. Here is a first implementation possibility:

```
Public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

            float temp = getTemperature();
            float humidity = getHumidity();
            float pressure = getPressure();

            currentConditionsDisplay.update(temp, humidity, pressure);
            statisticsDisplay.update(temp, humidity, pressure);
            forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

5. Based on our first implementation, we of the following apply?

A. We are coding to concrete implementations, not interfaces.	X
B. For every new display element we need to alter code.	X
C. We have no way to add or remove display elements at run time.	X
D. The display elements don't implement a common interface.	
E. We have not encapsulated the part that changes.	X
F. We are violating encapsulation of the WeatherData class.	

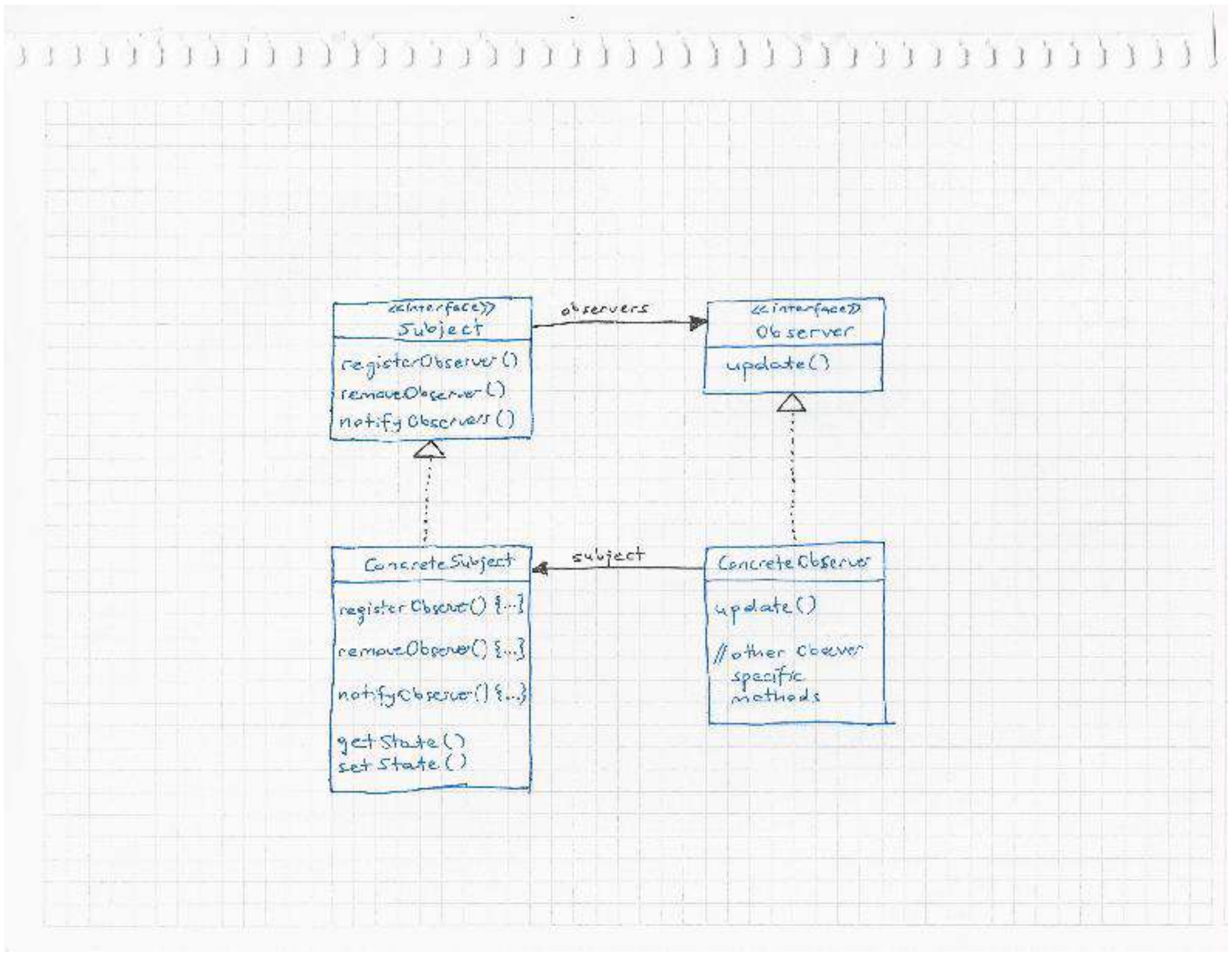
6. What's wrong with our implementation?
   By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.
   At least we seem to be using a common interface to talk to the display elements. They all have an update method that takes the temp, humidity and pressure values.

7. Observer pattern works like newspaper subscriptions. An object subscribes to a subject that it wants to be an observer. When the subject gets a new data value, it sends a notification to all the registered objects. When an object asks to be removed as an observer, it no longer receives updates from the subject.

8. The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

   Let's relate this definition to what we have been talking about the pattern:
   The subject and observers define the one-to-many relationships. The observers are dependent on the subject such that when the subject's state changes, the observers get notified.

9. There are a few different ways to implement the Observer Pattern but most revolve around a class design that includes Subject and Observer interfaces.



10. Subject interface provides registerObserver(), removeObserver() and notifyObservers() methods. Objects use this interface to register as observers and also to remove themselves from being observers. All potential observers need to implement the Observer interface, which has just one method, update(), that gets called when the subject's state changes.

11. A concreteSubject implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method to update all the current observers whenever state changes. Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

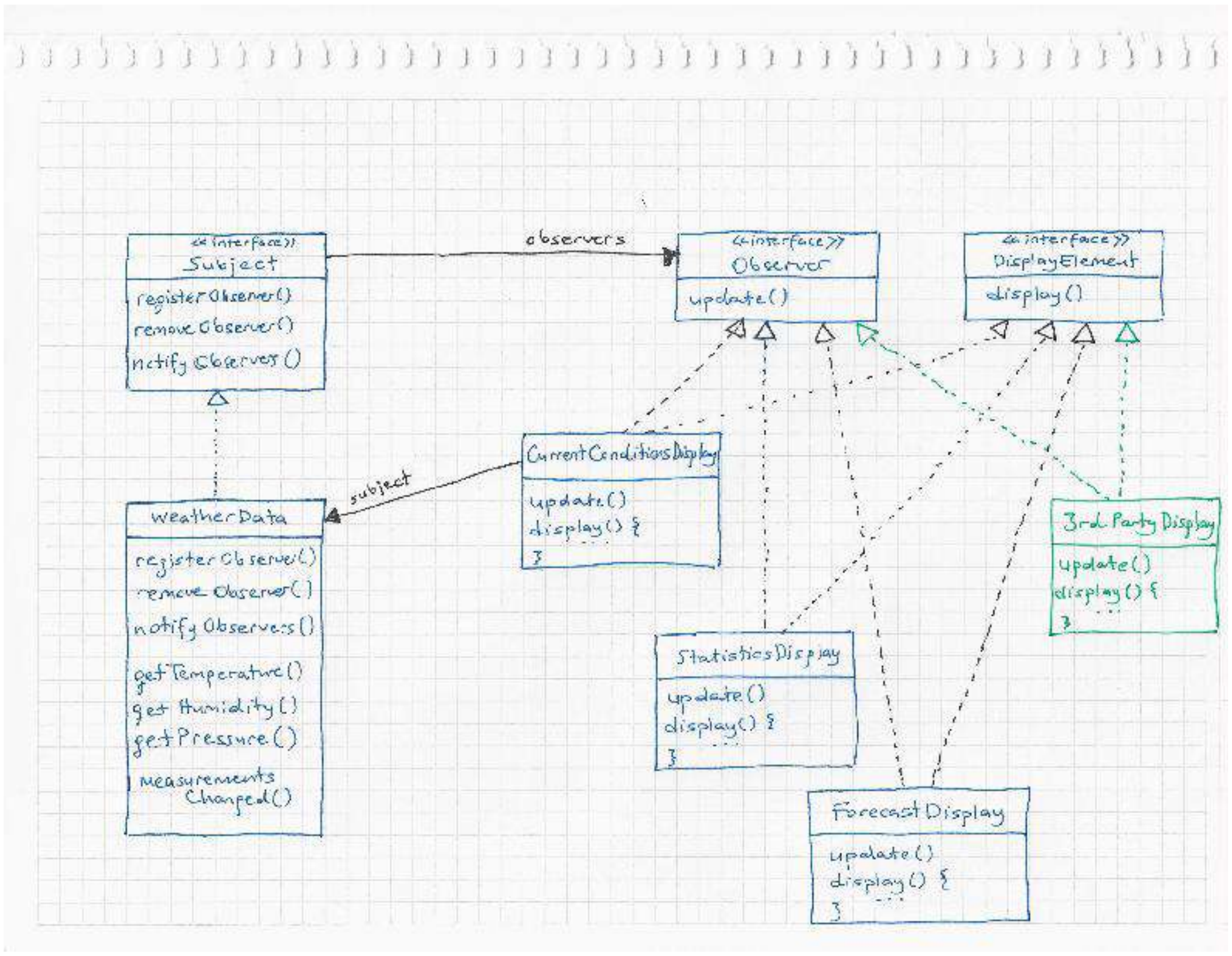
12. The Power of Loose Coupling: When two objects are loosely coupled, they can interact, but have very little knowledge of each other. The Observer Pattern provides an object design where subjects and observers are loosely coupled. Why?
    - The only thing the subject knows about an observer is that it implements a certain interface (Observer interface). It does not need to know the concrete class of the observer, what it does, or anything else about it.

- We can add new observers any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. Likewise, we can remove observers whenever we want.
- We never need to modify the subject to add new types of observers. Let's say we have a new concrete class come along that needs to be an observer. We do not need to make any changes to the subject to accommodate the new class type. All we have to do is to implement the Observer interface in the new class and register as an observer.
- We can reuse subjects or observers independently of each other. If we have another use for a subject or an observer, we can easily reuse them because the two are not tightly coupled.
- Changes to either the subject or an observer will not affect the other. Because the two are loosely coupled, we are free to make changes to either, as long as the objects meet their obligations to implement the subject or observer interfaces.

13. **Design Principle**: Strive for loosely coupled designs between objects that interact.

14. Loosely coupled designs allow us to build flexible object-oriented systems that can handle change because they minimize the interdependency between objects. Because:
    a. The only thing the subject knows about an observer is that it implements a certain interface.
    b. We can add new observers at any time.
    c. We never need to modify the subject to add new types of observers.
    d. We can reuse subjects or observers independently of each other.
    e. Changes to either the subject or an observer will not affect the other.

15. Designing the Weather Station



16. Implementing the Weather Station

```
public interface Subject {
        public void registerObserver(Observer o);
        public void removeObserver(Observer o);
        public void notifyObservers();
}

public interface Observer {
        public void update(float temp, float humidity, float pressure);
}

public interface DisplayElement {
        public void display();
}
```

17. Implementing the Subject Interface in WeatherData

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
```

18. Building the display elements

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
```

```
        private Subject weatherData;

        public CurrentConditionsDisplay(Subject weatherData) {
            this.weatherData = weatherData;
            weatherData.registerObserver(this);
        }

        public void update(float temperature, float humidity, float pressure) {
            this.temperature = temperature;
            this.humidity = humidity;
            display();
        }

        public void display() {
            System.out.println("Current conditions: " + temperature
                + "F degrees and " + humidity + "% humidity");
        }
    }
```

19. Create a test harness and run the code

```
    public class WeatherStation {

        public static void main(String[] args) {
            WeatherData weatherData = new WeatherData();

            CurrentConditionsDisplay currentDisplay =
                new CurrentConditionsDisplay(weatherData);
            StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
            ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

            weatherData.setMeasurements(80, 65, 30.4f);
            weatherData.setMeasurements(82, 70, 29.2f);
            weatherData.setMeasurements(78, 90, 29.2f);
        }
    }
```

20. So far, we have rolled our own code for Observer Pattern, but Java has built-in support in several of its APIs. The most general is the Observer interface and the Observable class in the java.util.package. To get a high-level feel for java.util.Observer and java.util.Observable, we can rework the design for the WeatherStation as follows:

21. The built in Observer Pattern works a bit differently than the implementation that we used on the Weather Station. The most obvious difference is that WeatherData (our subject) now extends the

Observable class and inherits the add, delete and notify Observer methods. Here is how we use Java's version:

22. For an object to become an observer: As usual, implement the Observer interface (this time the java.util.Observer interface) and call addObserver() on any Observable object. Likewise, to remove yourself as an observer just call deleteObserver().

    For the Observable to send notifications: First of all you need to be Observable by extending the java.util.Observable superclass. From there it is a two step process: (1) You first must call the setChanged() method to signify that the state has changed in your object. (2) Then, call one of two notifyObservers() methods:

    Either notifyObservers() or notifyObservers(Object arg)

    For an Observer to receive notifications: It implements the update method, as before, but the signature of the method is a bit different:

    update(Observable o, Object arg);

    If you want to "push" data to the observers you can pass the data as a data object to the notifyObserver(arg) method. If not, then the Observer has to "pull" the data it wants from the Observable object passed to it.

23. The setChanged() method is used to signify that the state has changed and that notifyObservers() when it is called, should update its observers. If notifyObserver() is called without first calling setChanged(), the observers will not be notified. The setChanged method is used to signify that the state has changed and that notifyObservers(), when it is called, should update its observers.

```
setChanged() {
    changed = true
}

notifyObservers (Object arg) {
    if (changed) {
            for every observer on the list {
                    call update(this, arg)
            }
    }
}

notifyObservers () {
    notifyObservers(null)
}
```

24. Reworking the Weather Station with the built-in support: First, let's rework WeatherData to use java.util.Observable:

```
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public WeatherData() { }

    public void measurementsChanged() {
            setChanged();
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    public float getTemperature() {
            return temperature;
    }

    public float getHumidity() {
            return humidity;
    }

    public float getPressure() {
            return pressure;
    }
}
```

25. Now let’s rework the CurrentConditionsDisplay:

```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;
    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData) obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

    public void display() {
      System.out.println("Current conditions: " + temperature
          + "F degrees and " + humidity + "% humidity");
    }
}
```

26. Unfortunately, the java.util.Observable implementation has a number of problems that limit its usefulness and reuse:
    - As you have noticed, Observable is a class, not an interface, and worse, it does not even implement an interface. Since Observable is a class, you have to subclass it. That means you can't add on the Observable behavior to an existing class that already extends another superclass. This limits its reuse potential.
    - Because, there isn't an Observable interface, you cannot even create your own implementation that plays well with Java's built-in Observer API.
    - If you look at the Observable API, the setChanged() method is protected. This means, you cannot call setChanged() unless you have subclassed Observable. You cannot even create an instance of the Observable class and compose it with your own objects, you have to subclass it. The design violates a second design principle here: *favor composition over inheritance.*

HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

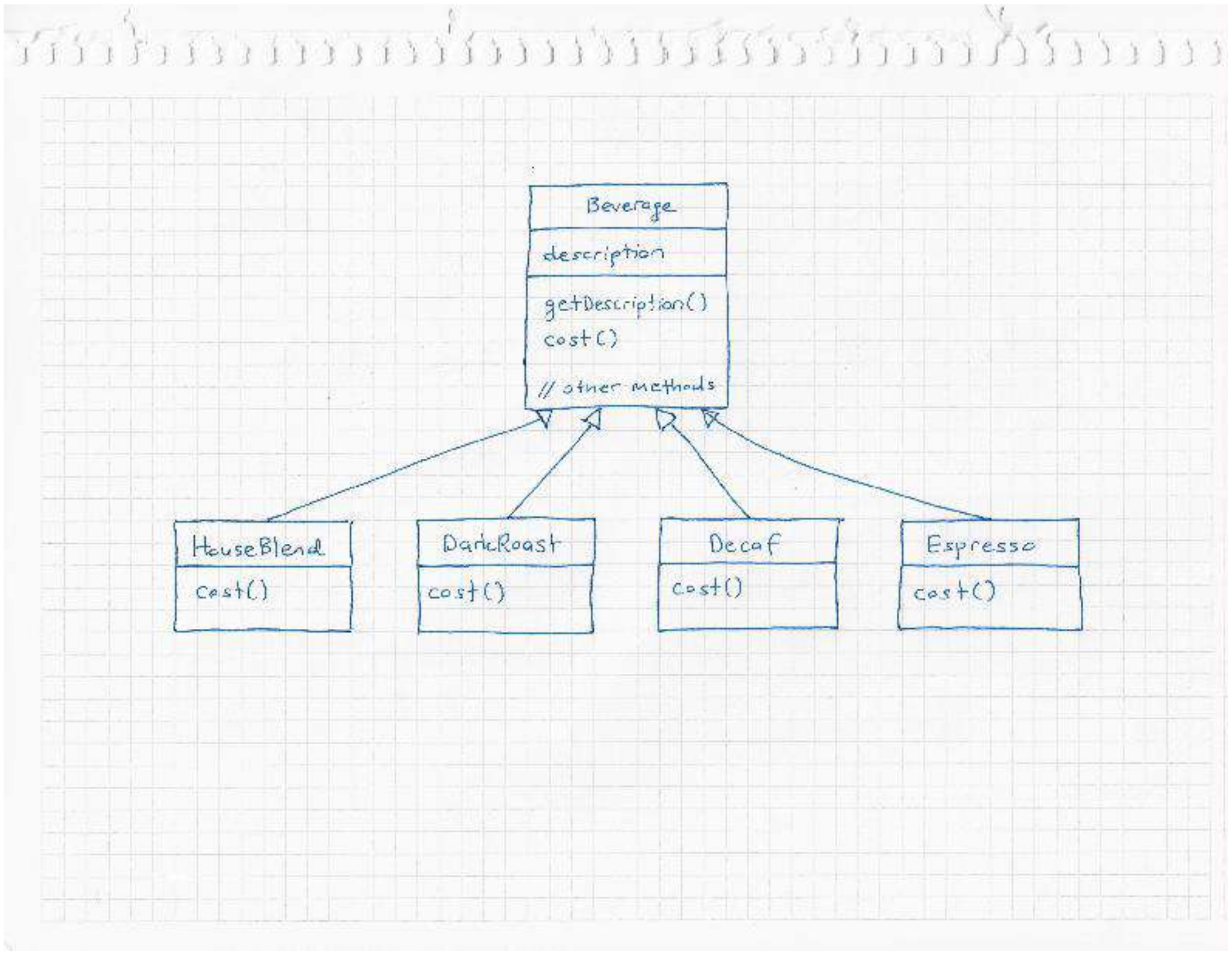
BBM 486 – DESIGN PATTERNS

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.

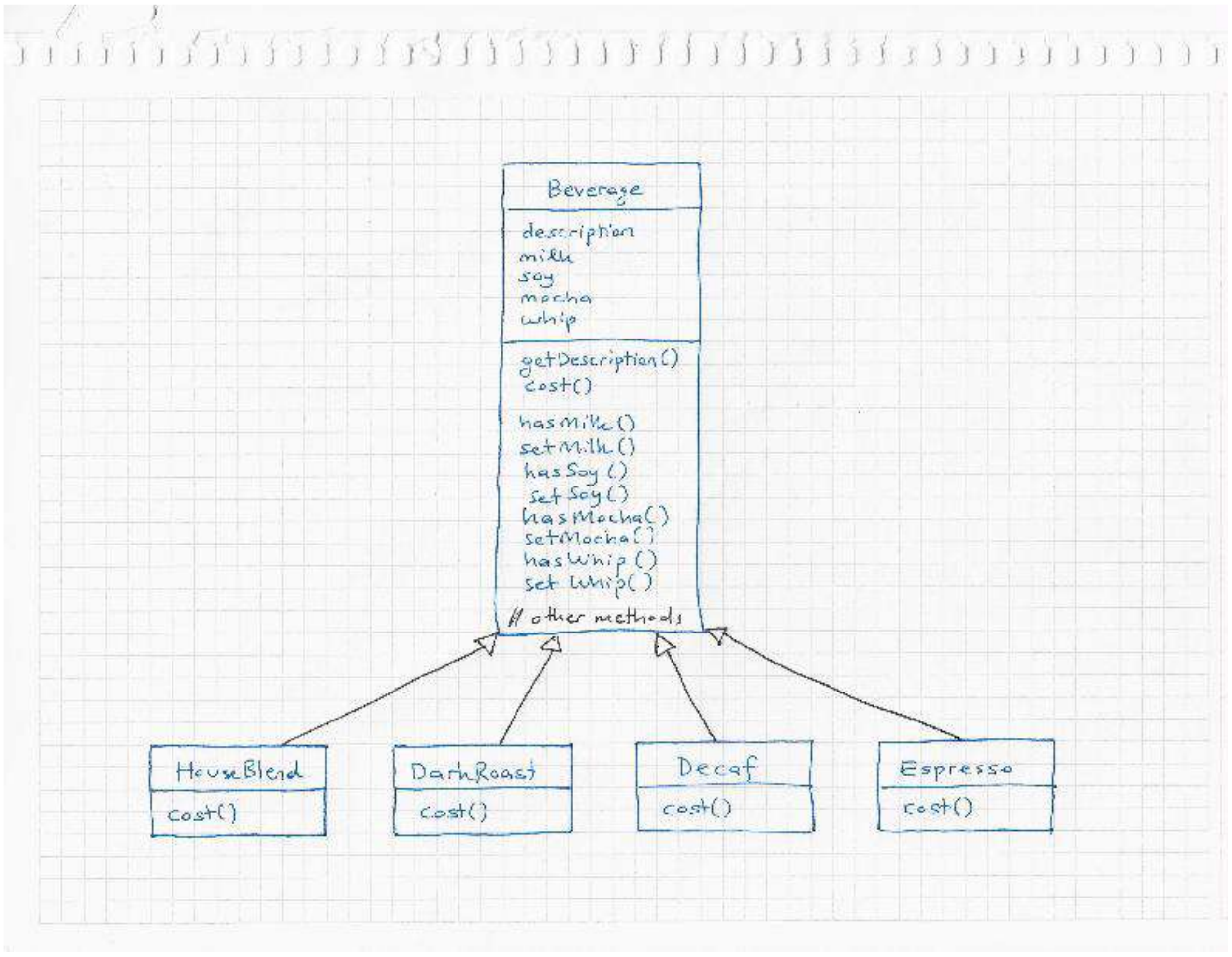
1. In the initial design of their system Beverage was an abstract class, which was subclassed by all beverages offered in the coffee shop: HouseBlend, DarkRoast, Decaf and Espresso. The description instance variable is set in each subclass. The cost() method is abstract; subclasses need to define their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



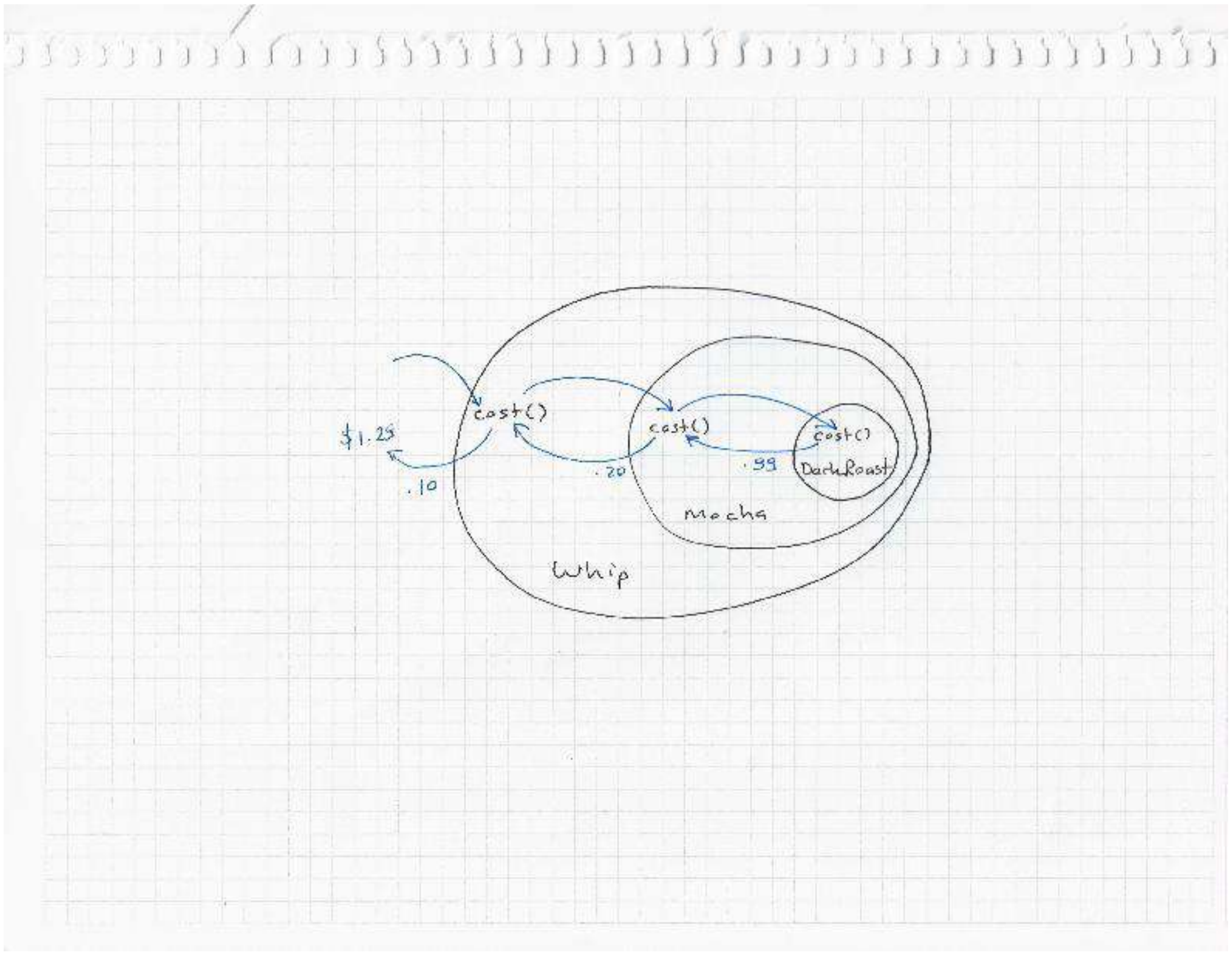
```
public class Beverage {

    // 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.

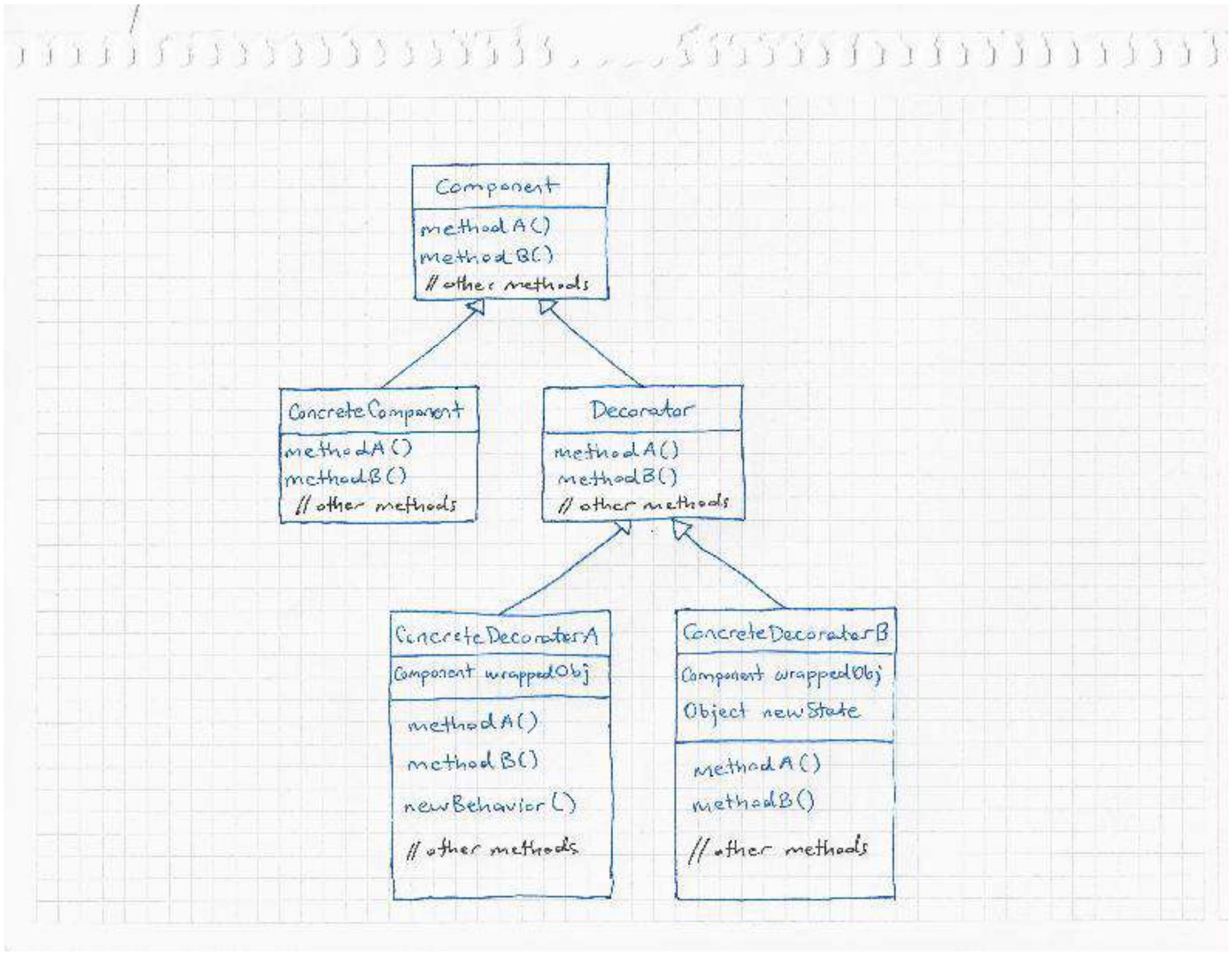
6. Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

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

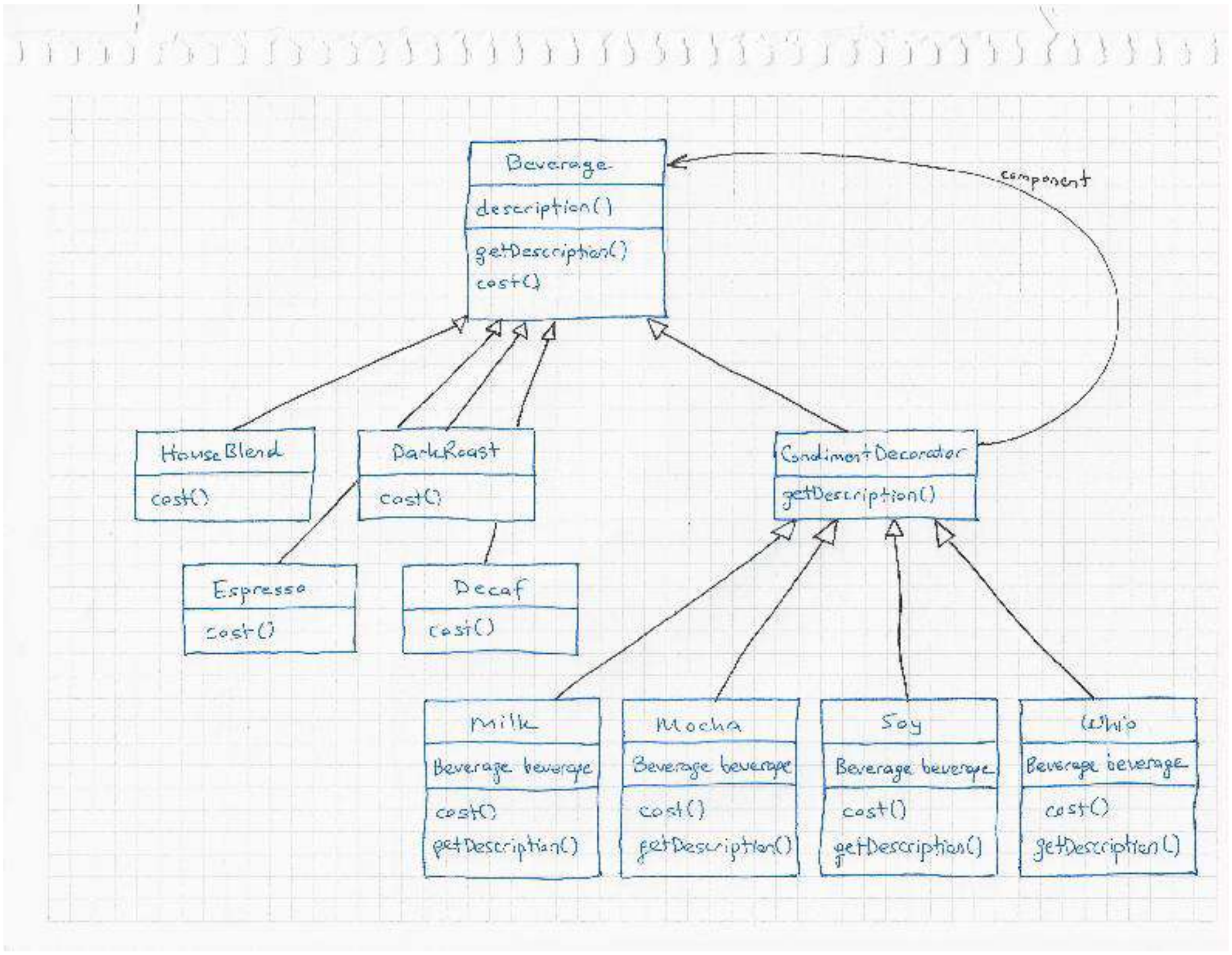


8. Decorators have the same supertype as the objects they decorate. You can use one or more decorators to wrap an object. Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.

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;
	}
}
```

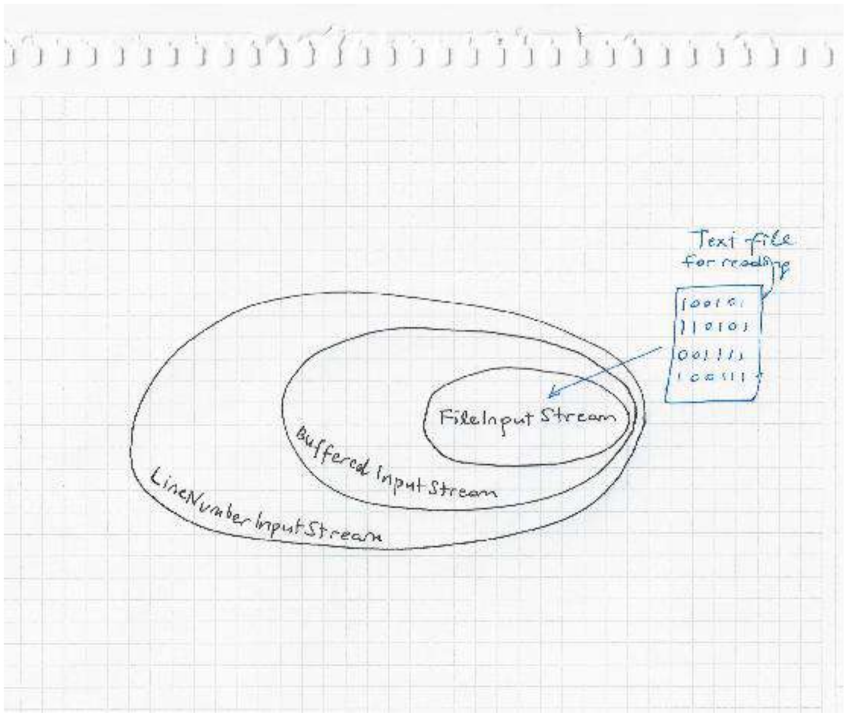
13. Coding condiments:

```
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;
	}
}
```

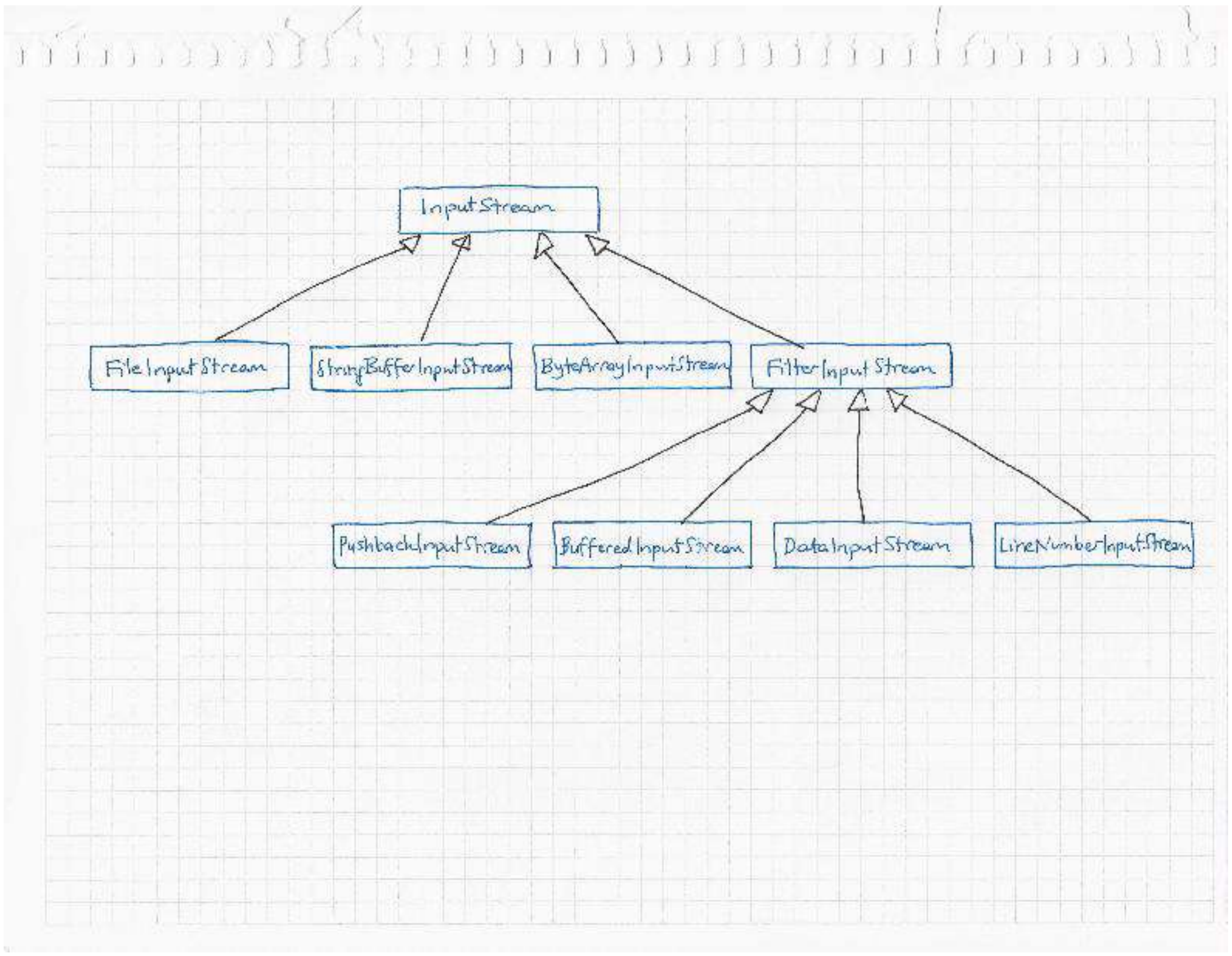
14. Here is a test code to make orders:

```
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();
                }
        }
    }
```

HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

BBM 486 – DESIGN PATTERNS

5. THE FACTORY PATTERN

The factory method pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

Here is more to making objects than just using the new operator. You will learn that instantiation is an activity that should not always be done in public and can often lead to coupling problems. Factory pattern can help save you from embarrasing dependencies. We will illustrate the factory pattern with as example.

1. Let’s say we have a pizza shop, and the pressure is on to add more pizza types. Initially, we might end up writing some code like this:

```
Pizza orderPizza() {
    Pizza pizza = new Pizza();

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

2. But you need more than one type of pizza… So, the you’d add some code that determines the appropriate type of pizza and then goes about making the pizza:

```
Pizza orderPizza(String type) {
    Pizza pizza;

    if (type.equals(“cheese”)) {
            pizza = new CheesePizza();
    } else if (type.equals(“greek”)) {
            pizza = new GreekPizza();
    } else if (type.equals(“pepperoni”)) {
            pizza = new PepperoniPizza();
    }

    pizza.prepare();
    pizza.bake();
```

```
        pizza.cut();
        pizza.box();
        return pizza;
    }
```

3. Assume that competitors have added a couple of trendy pizzas to their menus: the Clam Pizza and the Veggie Pizza. To keep up with the competition, you add these items to your menu. And you have not been selling many Greek Pizzas lately, so you decide to take that off the menu.

```
    Pizza orderPizza(String type) {
        Pizza pizza;

        if (type.equals(“cheese”)) {
                pizza = new CheesePizza();
        } else if (type.equals(“greek”)) {
                pizza = new GreekPizza();
        } else if (type.equals(“pepperoni”)) {
                pizza = new PepperoniPizza();
        } else if (type.equals(“clam”)) {
                pizza = new ClamPizza();
        } else if (type.equals(“veggie”)) {
                pizza = new VeggiePizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
```

This requires more modifications on orderPizza(). Clearly, dealing with which concrete class is initiated is really messing up our orderPizza() method, and preventing it from being closed for modification. But now that we know what is varying and what is not, it is probably time to encapsulate it. We have got a name for this new object: we call it a Factoty.

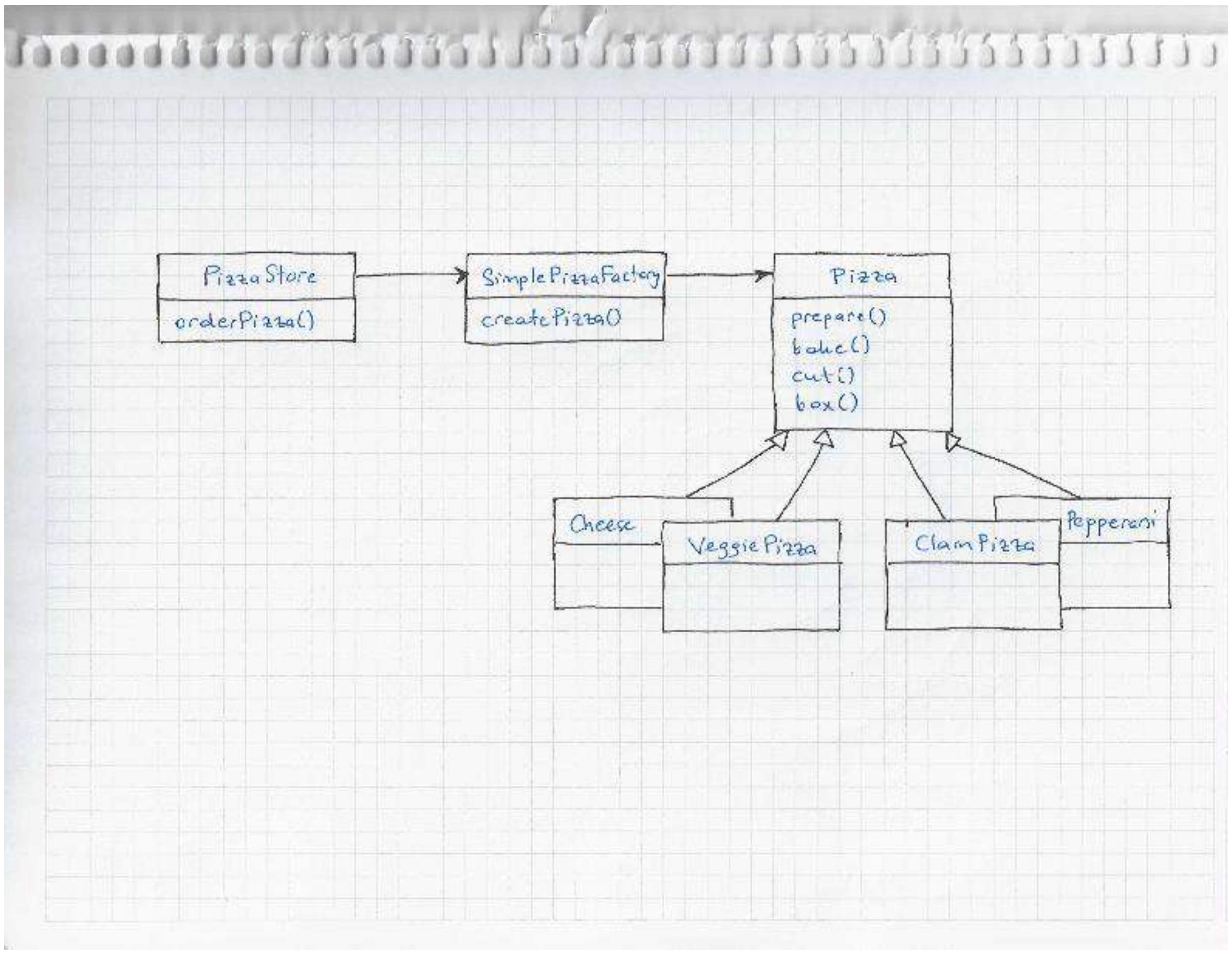
4. Encapsulating object creation: We know we would be better off moving the object creation out of the orderPizza() method into another object that is only going to be concerned with creating pizzas. Define a class that encapsulates the object creation for all pizzas:

```
    public class SimplePizzaFactory {
        public Pizza createPizza(String type) {
                Pizza pizza = null;
                if (type.equals("cheese")) {
                        pizza = new CheesePizza();
                } else if (type.equals("pepperoni")) {
                        pizza = new PepperoniPizza();
                } else if (type.equals("clam")) {
                        pizza = new ClamPizza();
                } else if (type.equals("veggie")) {
                        pizza = new VeggiePizza();
                }
                return pizza;
        }
}
```

5. Reworking the PizzaStore class:

```
public class PizzaStore {
        SimplePizzaFactory factory;

        public PizzaStore(SimplePizzaFactory factory) {
                this.factory = factory;
        }

        public Pizza orderPizza(String type) {
                Pizza pizza;

                pizza = factory.createPizza(type);

                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
                return pizza;
        }

        // other methods here

}
```

6. **The Simple Factory defined**
   The Simple Factory is not actually a Design Pattern; it is more of a programming idiom.



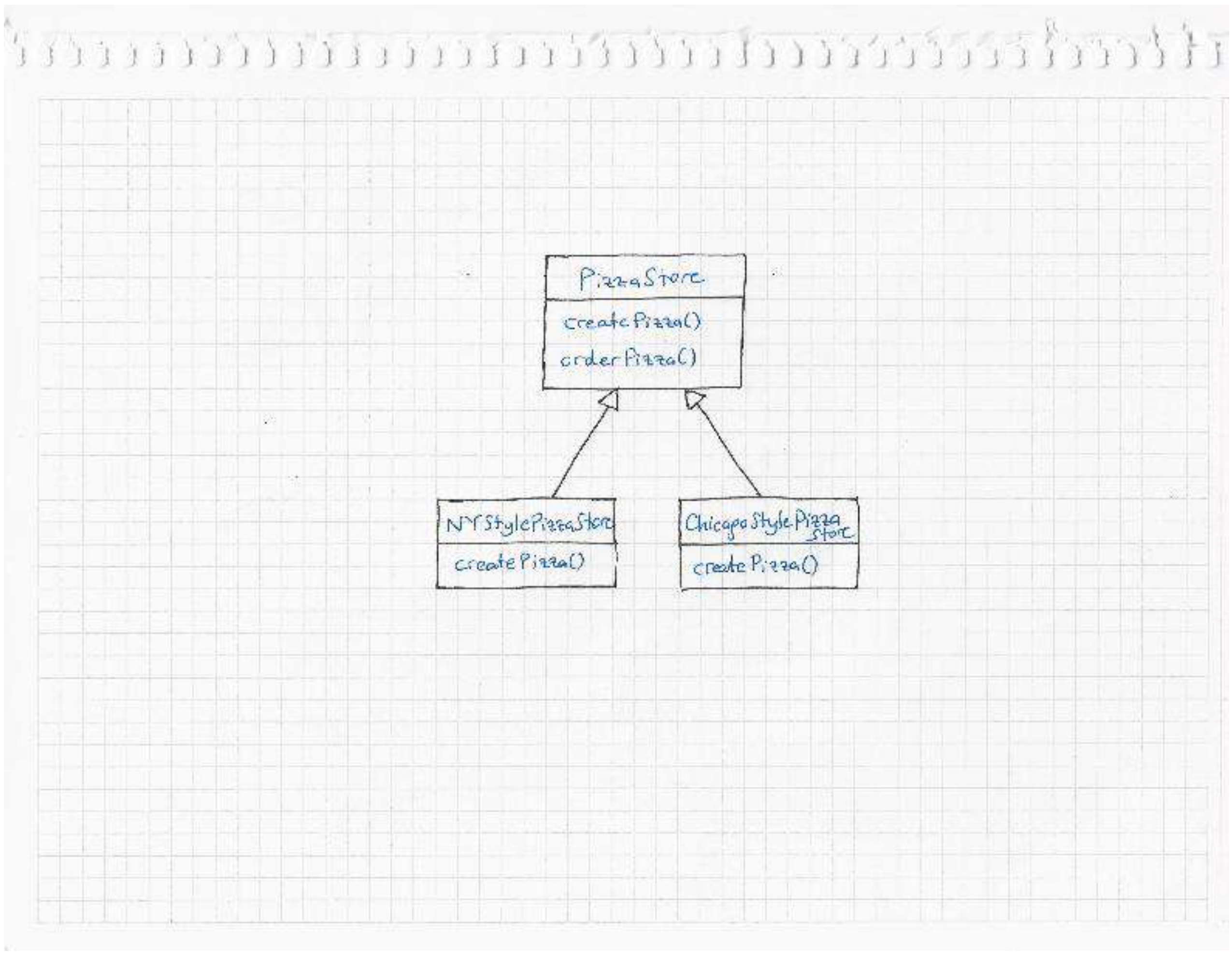
7. Franchising the pizza store: Your PizzaStore has done so well that everybody wants a PizzaStore in their own neighborhood. As the franchiser, you want to ensure the quality of the franchise operations and so you want them to use your time-tested code. But what about regional differences? As one approach, we can take SimplePizzaFactory and create different factories for each region:

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("Veggie");

ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("Veggie");
```

8. The franchises were using the factory to create pizzas, but starting to employ their own home grown procedures for the rest of the process. Rethinking the problem, you decided to create a framework for the pizza store:

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
            Pizza pizza;

            pizza = createPizza(type);

            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.box();

            return pizza;
    }

    abstract Pizza createPizza(String type);

}
```

9. Allowing the subclasses to decide: What varies among the regional PizzaStores is the style of pizza they make. We are going to push all these variations into the createPizza() method, and make it responsible for creating the right kind of pizza.



10. Let's make a PizzaStore: All the regional stores need to do is subclass PizzaStore and supply a createPizza() method that implements their style of pizza.

```
public class NYPizzaStore extends PizzaStore {
    Pizza createPizza(String type) {
            if (item.equals("cheese")) {
                    return new NYStyleCheesePizza();
            } else if (item.equals("veggie")) {
                    return new NYStyleVeggiePizza();
            } else if (item.equals("clam")) {
                    return new NYStyleClamPizza();
            } else if (item.equals("pepperoni")) {
                    return new NYStylePepperoniPizza();
            } else return null;
    }
}
```

11. Declaring a factory method: We have gone from having an object handle the instantiation of our concrete classes to a set of subclasses that are now taking on that responsibility.

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
            Pizza pizza;

            pizza = createPizza(type);

            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.box();

            return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here

}
```

A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

abstract Product factoryMethod (String type)

12. Let's check out how these (NY syle and Chicago style) pizzas are really made to order…

First, we need a NY PizzaStore:

```
    PizzaStore nyPizzaStore = new NYPizzaStore();
```

Then, we can take an order:

```
    nyPizzaStore.orderPizza("cheese");
```

The orderPizza() method then calls the createPizza() method:

```
    Pizza pizza = createPizza("cheese");
```

Finally, we have the unprepared pizza in hand and the orderPizza() method finishes preparing it:

```
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
```

Let’s implement the Pizza abstract class:

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList toppings = new ArrayList();

    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (int i = 0; i < toppings.size(); i++) {
            System.out.println("   " + toppings.get(i));
        }
    }

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    public String getName() {
        return name;
    }
}
```

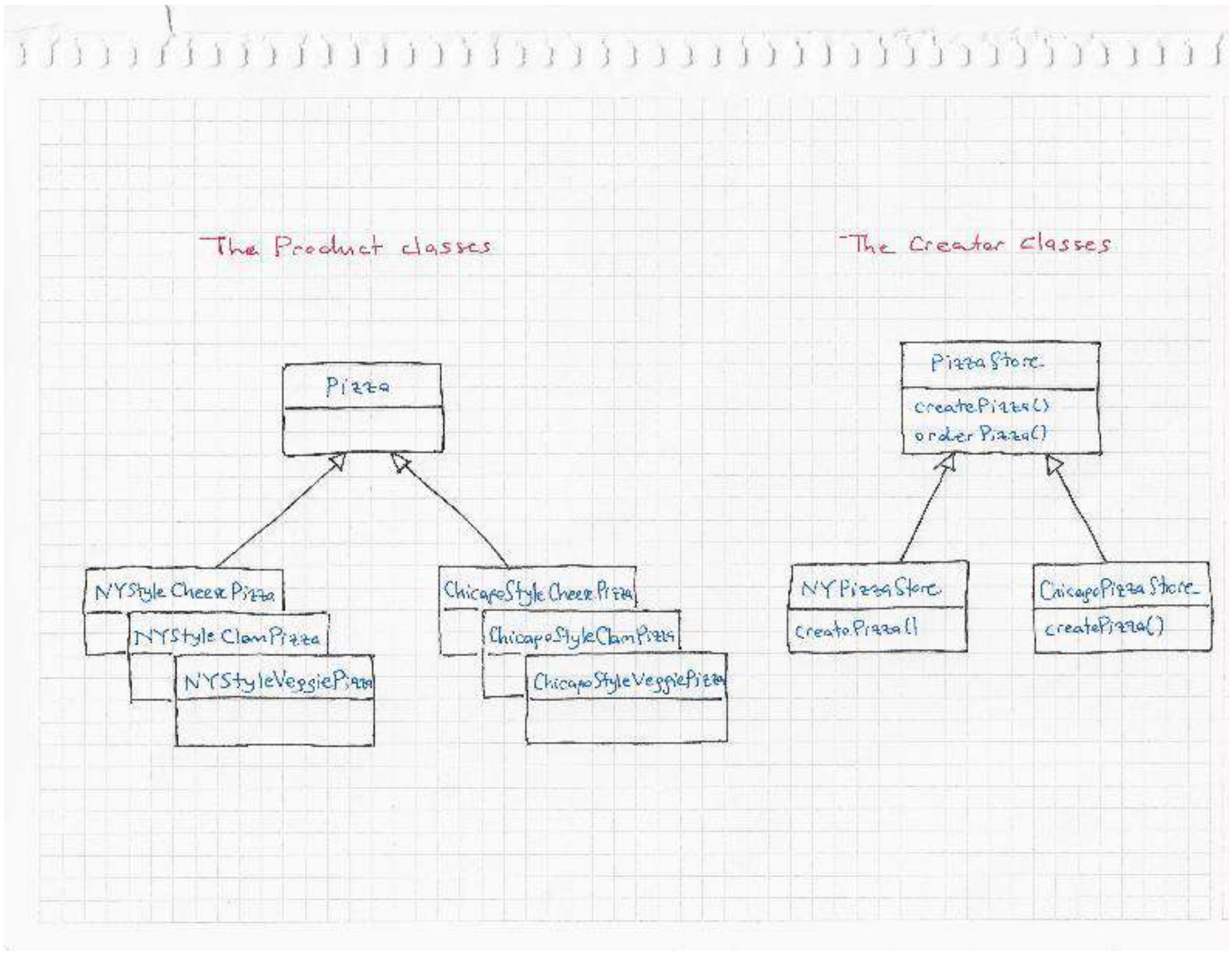
Now, we need some concrete subclasses:

```
public class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
        }
}

public class ChicagoStyleCheesePizza extends Pizza {
        public ChicagoStyleCheesePizza() {
                name = "Chicago Style Deep Dish Cheese Pizza";
                dough = "Extra Thick Crust Dough";
                sauce = "Plum Tomato Sauce";

                toppings.add("Shredded Mozzarella Cheese");
        }

        void cut() {
                System.out.println("Cutting the pizza into square slices");
        }
}
```

And here is how we can test this:

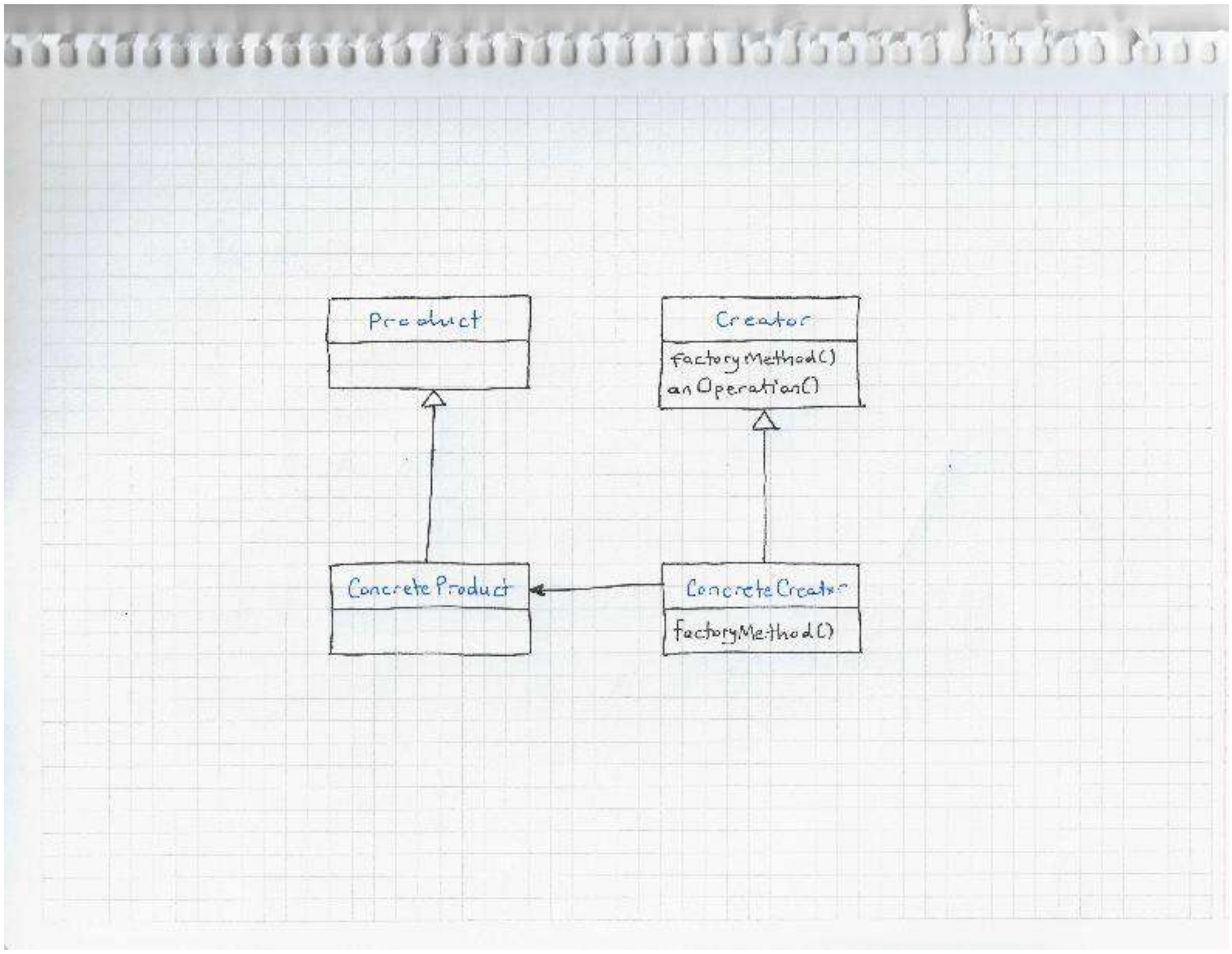
```
public class PizzaTestDrive {

        public static void main(String[] args) {
                PizzaStore nyStore = new NYPizzaStore();
                PizzaStore chicagoStore = new ChicagoPizzaStore();

                Pizza pizza = nyStore.orderPizza("cheese");
                System.out.println("Ethan ordered a " + pizza.getName() + "\n");

                pizza = chicagoStore.orderPizza("cheese");
                System.out.println("Joel ordered a " + pizza.getName() + "\n");
        }
}
```

13. It’s finally time to meet the Factory Method Pattern:



14. Factory Method Pattern defines an interface for creating an object, but lets the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

    As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types.



15. A very dependent PizzaStore

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

16. When you directly instantiate an object, you are depending on its concrete class. It should be pretty clear that reducing dependencies to concrete classes in our code is a good thing. We have got an OO design principle that formalizes this notion; *Dependency Inversion Principle*.

17. **Design Principle**: Depend upon abstractions. Do not depend upon concrete classes.

18. This is also called **dependency inversion principle**. This sound a lot like “Program to an interface, not an implementation,” but it makes an even stronger statement about abstraction. It suggests that out high-level components should not depend on our low-level components, rather, they should depend on abstractions.

19. The design for the PizzaStore is really shaping up: it has got a flexible framework and it does a good job of adhering to design principles. Now what you have discovered is that your franchises have been following your procedures, but a few franchises have been substituting inferior ingredients in their pies to lower costs. So how are you going to ensure each franchise is using quality ingredients?

20. Building the ingredient factories:

```
public interface PizzaIngredientsFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();
}
```

21. Building the New York and Chicago ingredient factories:

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();
    }

    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    public Cheese createCheese() {
        return new ReggianoCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FreshClams();
    }
}



public class ChicagoPizzaIngredientFactory implements PizzaIngredientFactory
{

    public Dough createDough() {
        return new ThickCrustDough();
    }

    public Sauce createSauce() {
```

```
		return new PlumTomatoSauce();
	}

	public Cheese createCheese() {
		return new MozzarellaCheese();
	}

	public Veggies[] createVeggies() {
		Veggies veggies[] = { new BlackOlives(), new Spinach(), new Eggplant() };
		return veggies;
	}

	public Pepperoni createPepperoni() {
		return new SlicedPepperoni();
	}

	public Clams createClam() {
		return new FrozenClams();
	}
}
```

22. Reworking the pizzas…

```
public abstract class Pizza {
	String name;

	Dough dough;
	Sauce sauce;
	Veggies veggies[];
	Cheese cheese;
	Pepperoni pepperoni;
	Clams clam;

	abstract void prepare();

	void bake() {
		System.out.println("Bake for 25 minutes at 350");
	}

	void cut() {
		System.out.println("Cutting the pizza into diagonal slices");
	}
```

```
    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
            // code to print pizza here
    }
```

23. When we wrote the Factory Method code, we had a NYCheesePizza and a ChicagoCheesePizza class. If you look at the two classes, the only thing that differs is the use of regional differences. The pizzas are made just the same (dough + sauce + cheese). The same goes for the other pizzas: Veggie, Clam, and so on. They all follow the same preparation steps; they just have different ingredients. So, we really need two classes for each pizza; the ingredient factory is going to handle the regional differences for us.

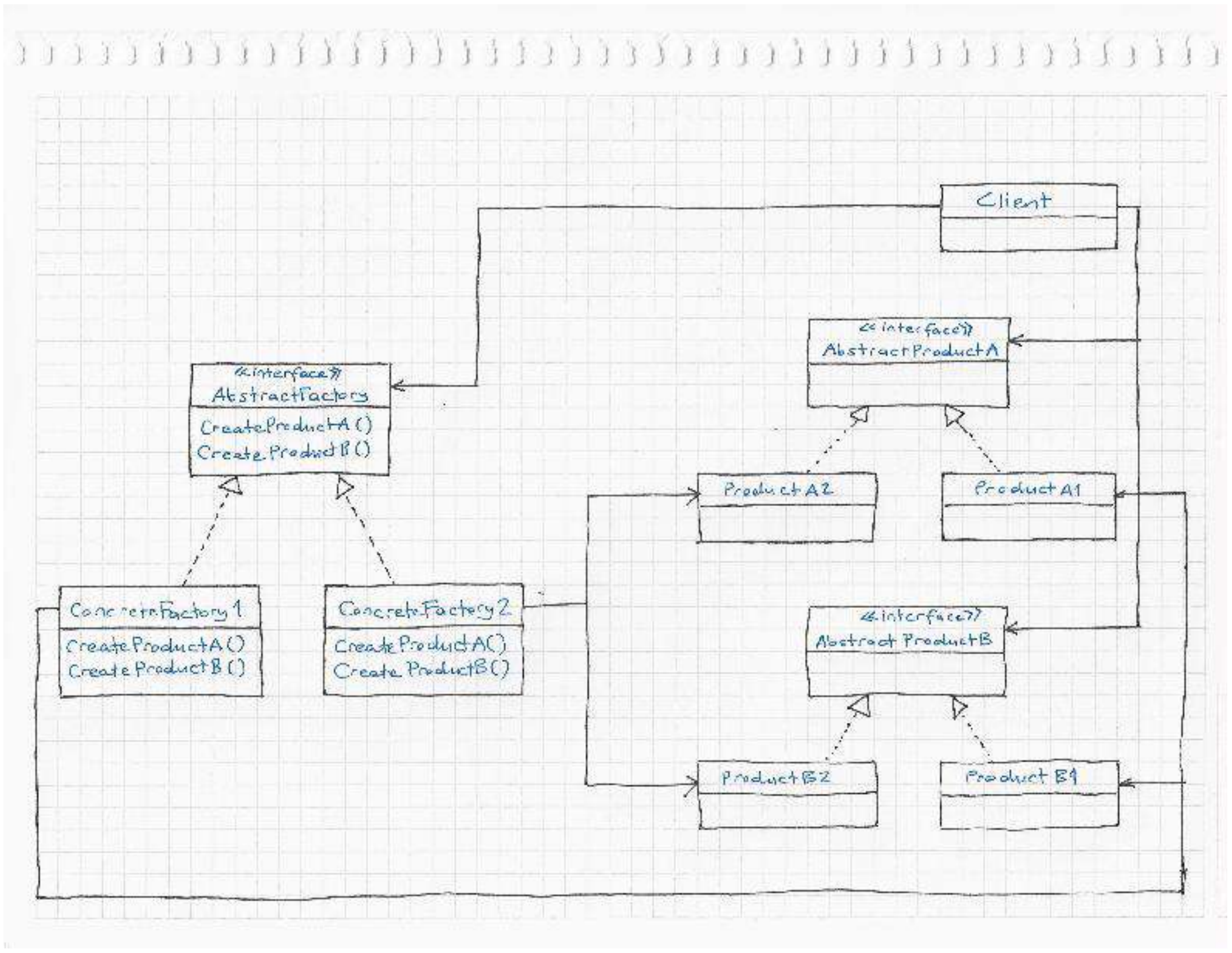
```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}


public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public ClamPizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

24. Revisiting our pizza stores

```
    public class NYPizzaStore extends PizzaStore {

        protected Pizza createPizza(String item) {
            Pizza pizza = null;
            PizzaIngredientFactory ingredientFactory =
                new NYPizzaIngredientFactory();

            if (item.equals("cheese")) {

                pizza = new CheesePizza(ingredientFactory);
                pizza.setName("New York Style Cheese Pizza");

            } else if (item.equals("veggie")) {

                pizza = new VeggiePizza(ingredientFactory);
                pizza.setName("New York Style Veggie Pizza");

            } else if (item.equals("clam")) {

                pizza = new ClamPizza(ingredientFactory);
                pizza.setName("New York Style Clam Pizza");

            } else if (item.equals("pepperoni")) {

                pizza = new PepperoniPizza(ingredientFactory);
                pizza.setName("New York Style Pepperoni Pizza");

            }
            return pizza;
        }
```

}

25. The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. In this way, the client is decoupled from any of the specifics of the concrete products.



Client
<<interface>>
AbstractProductA
<<interface>>
AbstractFactory
CreateProductA()
CreateProductB()
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
<<interface>>
AbstractProductB
ProductB2
ProductB1