What is a Design Pattern?

Design patterns are design level solutions for recurring problems that we software engineers come across often. It's not code - I repeat, ✕**CODE**. It is like a description on how to tackle these problems and design a solution.
Using these patterns is considered good practice, as the design of the solution is quite tried and tested, resulting in higher readability of the final code. Design patterns are quite often created for and used by OOP Languages, like Java, in which most of the examples from here on will be written.

Types of design patterns

There are about 23 Patterns currently discovered

List of the Original 23 Patterns

Purpose	Design Pattern	Aspect(s) that can vary
Creational	Abstract Factory	families of product objects
	Builder	how a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class
Structural	Adapter	interface to an object
	Bridge	implementation of an object

	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Facade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location
Behavioral	Chain of Responsibility	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed, traversed
	Mediator	how and which objects interact with each other
	Memento	what private information is stored outside an object, and when
	Observer	number of objects that depend on another object; how the dependent objects stay up to date
	State	states of an object

	Strategy	an algorithm
	Template Method	steps of an algorithm
	Visitor	operations that can be applied to object(s) without changing their class(es)

These 23 can be classified into 3 types:

1. Creational: These patterns are designed for class instantiation. They can be either class-creation patterns or object-creational patterns.

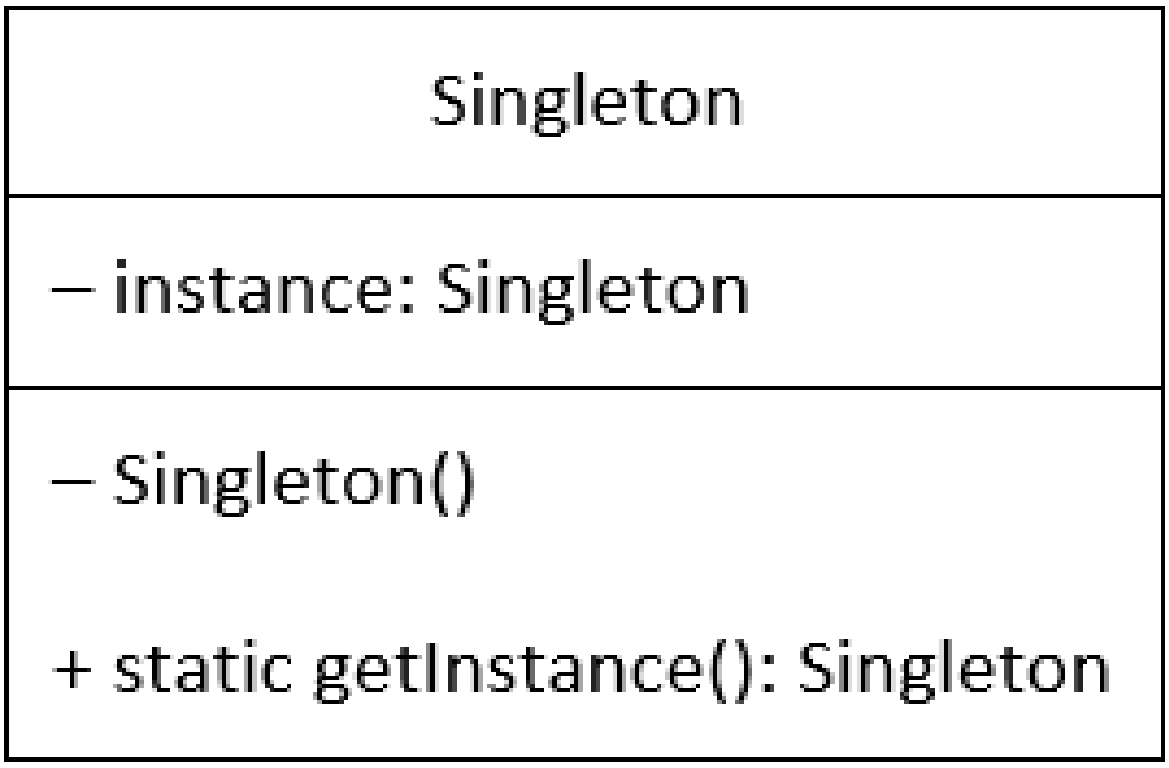
2. Structural: These patterns are designed with regard to a class's structure and composition. The main goal of most of these patterns is to increase the functionality of the class (es) involved, without changing much of its composition.

3. Behavioral: These patterns are designed depending on how one class communicates with others.

In this post, we will go through one basic design pattern for each classified type.

Type 1: Creational - The Singleton Design Pattern

The Singleton Design Pattern is a Creational pattern, whose objective is to create only one instance of a class and to provide only one global access point to that object. One commonly used example of such a class in Java is Calendar, where you cannot make an instance of that class. It also uses its own getInstance()method to get the object to be used.
A class using the singleton design pattern will include,



Singleton Class Diagram

1. A private static variable, holding the only instance of the class.
2. A private constructor, so it cannot be instantiated anywhere else.
3. A public static method, to return the single instance of the class.

There are many different implementations of singleton design. Today, I'll be going through the implementations of;

1. Eager Instantiation

2. Lazy Instantiation

3. Thread-safe Instantiation

Eager Beaver

```
public class EagerSingleton {
        // create an instance of the class.
        private static EagerSingleton instance = new EagerSingleton();

        // private constructor, so it cannot be instantiated outside this class.
```

```
	private EagerSingleton() {  }

	// get the only instance of the object created.
	public static EagerSingleton getInstance() {
		return instance;
	}
}
```

This type of instantiation happens during class loading, as the instantiation of the variable instance happens outside any method. This poses a hefty drawback if this class is not being used at all by the client application. The contingency plan, if this class is not being used, is the Lazy Instantiation.

Lazy Days

There isn't much difference from the above implementation. The main differences are that the static variable is initially declared null, and is only instantiated within the getInstance() method if - and only if - the instance variable remains null at the time of the check.

```
public class LazySingleton {
	// initialize the instance as null.
	private static LazySingleton instance = null;

	// private constructor, so it cannot be instantiated outside this class.
	private LazySingleton() {  }

	// check if the instance is null, and if so, create the object.
	public static LazySingleton getInstance() {
		if (instance == null) {
			instance = new LazySingleton();
		}
		return instance;
	}
}
```

This fixes one problem, but another one still exists. What if two different clients access the Singleton class at the same time, right to the millisecond? Well, they will check if the instance is null at the same time, and will find it true, and so will create two instances of the class for each request by the two clients. To fix this, Thread Safe instantiation is to be implemented.

(Thread) Safety is Key

In Java, the keyword synchronized is used on methods or objects to implement thread safety, so that only one thread will access a particular resource at one time. The class instantiation is put within a synchronized block so that the method can only be accessed by one client at a given time.

```
public class ThreadSafeSingleton {
	// initialize the instance as null.
	private static ThreadSafeSingleton instance = null;

	// private constructor, so it cannot be instantiated outside this class.
	private ThreadSafeSingleton() {  }

	// check if the instance is null, within a synchronized block. If so, create the
object
	public static ThreadSafeSingleton getInstance() {
		synchronized (ThreadSafeSingleton.class) {
			if (instance == null) {
				instance = new ThreadSafeSingleton();
			}
		}
		return instance;
	}
}
```

The overhead for the synchronized method is high, and reduces the performance of the whole operation.

For example, if the instance variable has already been instantiated, then each time any client accesses the getInstance() method, the synchronized method is run and the performance drops. This just happens in order to check if the instance variables' value is null. If it finds that it is, it leaves the method.
To reduce this overhead, double locking is used. The check is used before the synchronized method as well, and if the value is null alone, does the synchronized method run.

```
// double locking is used to reduce the overhead of the synchronized method
public static ThreadSafeSingleton getInstanceDoubleLocking() {
	if (instance == null) {
		synchronized (ThreadSafeSingleton.class) {
			if (instance == null) {
				instance = new ThreadSafeSingleton();
```

```
                    }
                }
            }
            return instance;
}
```

Now onto the next classification.

Type 2: Structural - The Decorator Design Pattern

I'm gonna give you a small scenario to give a better context to why and where you should use the Decorator Pattern.

Say you own a coffee shop, and like any newbie, you start out with just two types of plain coffee, the house blend and dark roast. In your billing system, there was one class for the different coffee blends, which inherits the beverage abstract class. People actually start to come by and have your wonderful (albeit bitter?) coffee. Then there are the coffee newbs that, God forbid, want sugar or milk. Such a travesty for coffee!! ??

Now you need to have those two add-ons as well, both to the menu and unfortunately on the billing system. Originally, your IT person will make a subclass for both coffees, one including sugar, the other milk. Then, since customers are always right, one says these dreaded words:

"Can I get a milk coffee, with sugar, please?"

???

There goes your billing system laughing in your face again. Well, back to the drawing board....

The IT person then adds milk coffee with sugar as another subclass to each parent coffee class. The rest of the month is smooth sailing, people lining up to have your coffee, you actually making money. ??

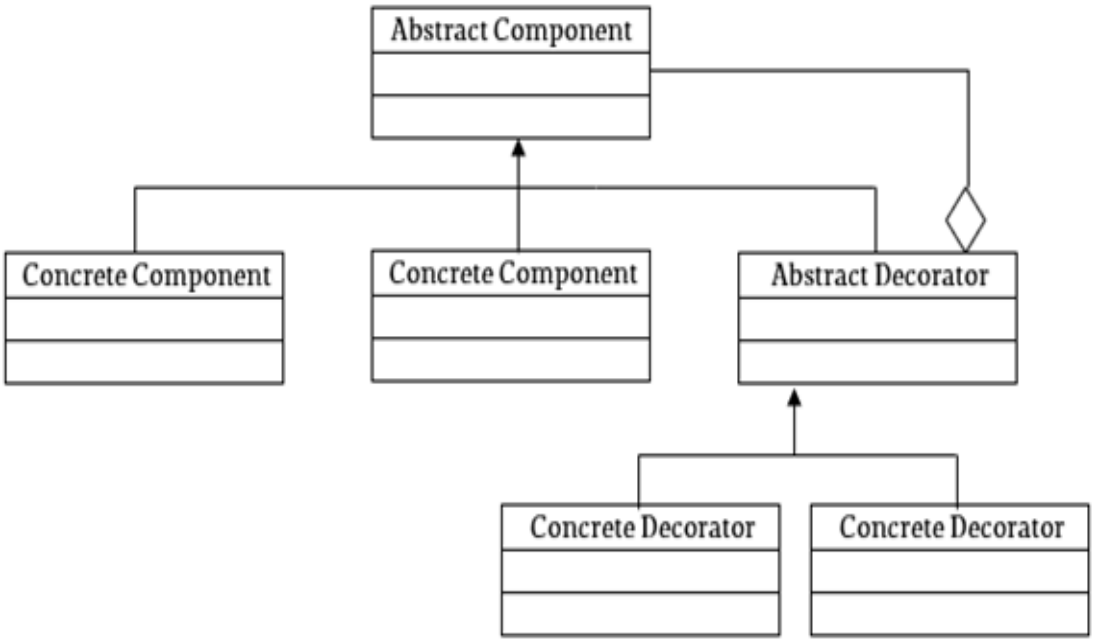
But wait, there's more!

The world is against you once again. A competitor opens up across the street, with not just 4 types of coffee, but more than 10 add-ons as well!

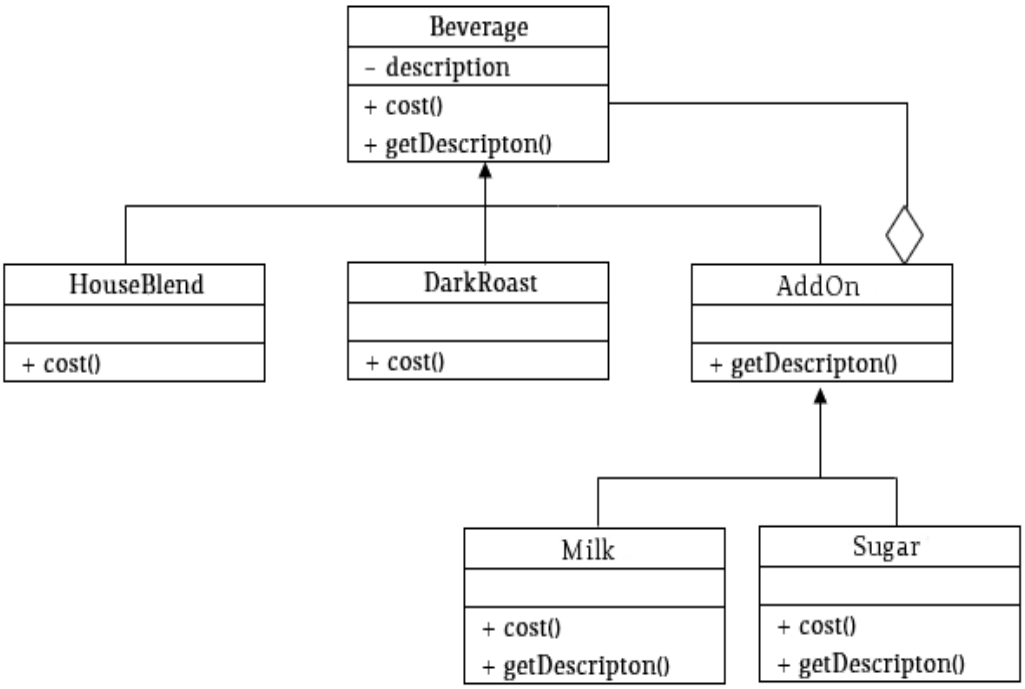
You buy all those and more, to sell better coffee yourself, and just then remember that you forgot to update that dratted billing system. You quite possibly cannot make the infinite number of subclasses for any and all combinations of all the add-ons, with the new coffee blends too. Not to mention, the size of the final system.

Time to actually invest in a proper billing system. You find new IT personnel, who actually knows what they are doing and they say,

"Why, this will be so much easier and smaller if it used the decorator pattern."



Decorator Design Pattern Class diagram



Class diagram according to coffee shop scenario
If we map out our scenario according to the class diagram above, we get 4 classes for the 4 coffee blends, 10 for each add-on and 1 for the abstract component and 1 more for the abstract decorator. See! 16! Now hand over that $100? (jk, but it will not be refused if given… just saying)

As you can see from above, just as the concrete coffee blends are subclasses of the beverage abstract class, the Add-on abstract class also inherits its methods from it. The add-ons, that are its subclasses, in turn inherit any new methods to add functionality to the base object when needed.

Let's get to coding, to see this pattern in use.

First to make the Abstract beverage class that all the different coffee blends will inherit from:

```
public abstract class Beverage {
        private String description;

        public Beverage(String description) {
                super();
                this.description = description;
```

```
	}

	public String getDescription() {
		return description;
	}

	public abstract double cost();
}
```

Then to add both the concrete coffee blend classes.

```
public class HouseBlend extends Beverage {
	public HouseBlend() {
		super("House blend");
	}

	@Override
	public double cost() {
		return 250;
	}
}

public class DarkRoast extends Beverage {
	public DarkRoast() {
		super("Dark roast");
	}

	@Override
	public double cost() {
		return 300;
	}
}
```

The AddOn abstract class also inherits from the Beverage abstract class (more on this below).

```
public abstract class AddOn extends Beverage {
	protected Beverage beverage;
```

```
        public AddOn(String description, Beverage bev) {
                super(description);
                this.beverage = bev;
        }

        public abstract String getDescription();
}
```
And now the concrete implementations of this abstract class:

```
public class Sugar extends AddOn {
        public Sugar(Beverage bev) {
                super("Sugar", bev);
        }

        @Override
        public String getDescription() {
                return beverage.getDescription() + " with Mocha";
        }

        @Override
        public double cost() {
                return beverage.cost() + 50;
        }
}

public class Milk extends AddOn {
        public Milk(Beverage bev) {
                super("Milk", bev);
        }

        @Override
        public String getDescription() {
                return beverage.getDescription() + " with Milk";
        }
```

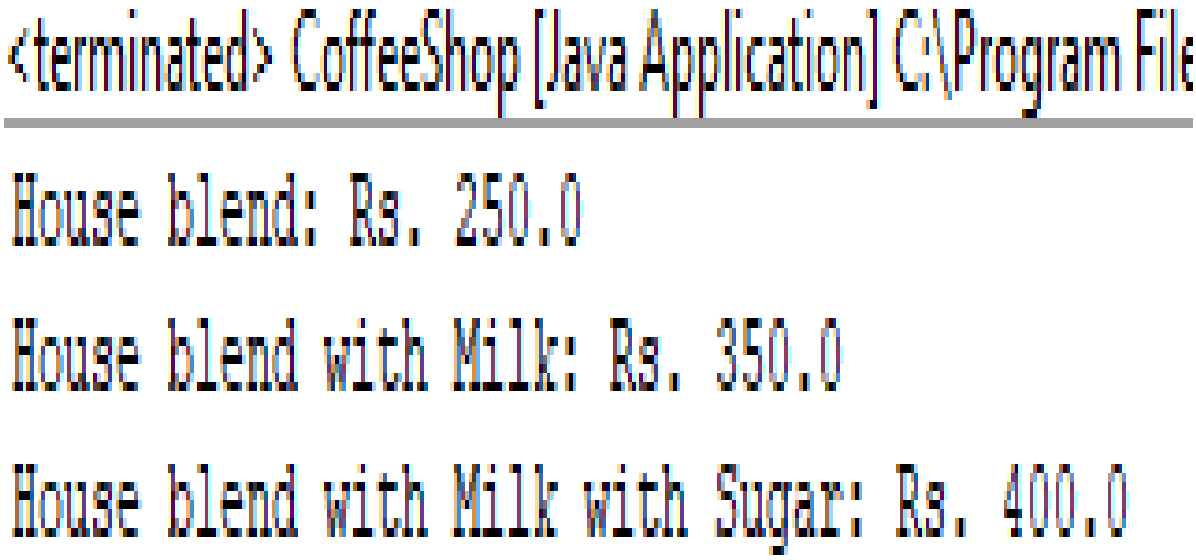
```
        @Override  public double cost() {
                return beverage.cost() + 100;
        }
}
```
As you can see above, we can pass any subclass of Beverage to any subclass of AddOn, and get the added cost as well as the updated description. And, since the AddOn class is essentially of type Beverage, we can pass an Add-on into another Add-on. This way, we can add any number of add-ons to a specific coffee blend.

Now to write some code to test this out.

```
public class CoffeeShop {
        public static void main(String[] args) {
                HouseBlend houseblend = new HouseBlend();
                System.out.println(houseblend.getDescription() + ": " +
houseblend.cost());

                Milk milkAddOn = new Milk(houseblend);
                System.out.println(milkAddOn.getDescription() + ": " +
milkAddOn.cost());

                Sugar sugarAddOn = new Sugar(milkAddOn);
                System.out.println(sugarAddOn.getDescription() + ": " +
sugarAddOn.cost());
        }
}
```
The final result is:



P.S. this is in Sri Lankan Rupees
It works! We were able to add more than one add-on to a coffee blend and successfully update its final cost and description, without the need to make infinite subclasses for each add-on combination for all coffee blends.

Finally, to the last category.

Type 3: Behavioral - The Command Design Pattern
A behavioral design pattern focuses on how classes and objects communicate with each other. The main focus of the command pattern is to inculcate a higher degree of loose coupling between involved parties (read: classes).

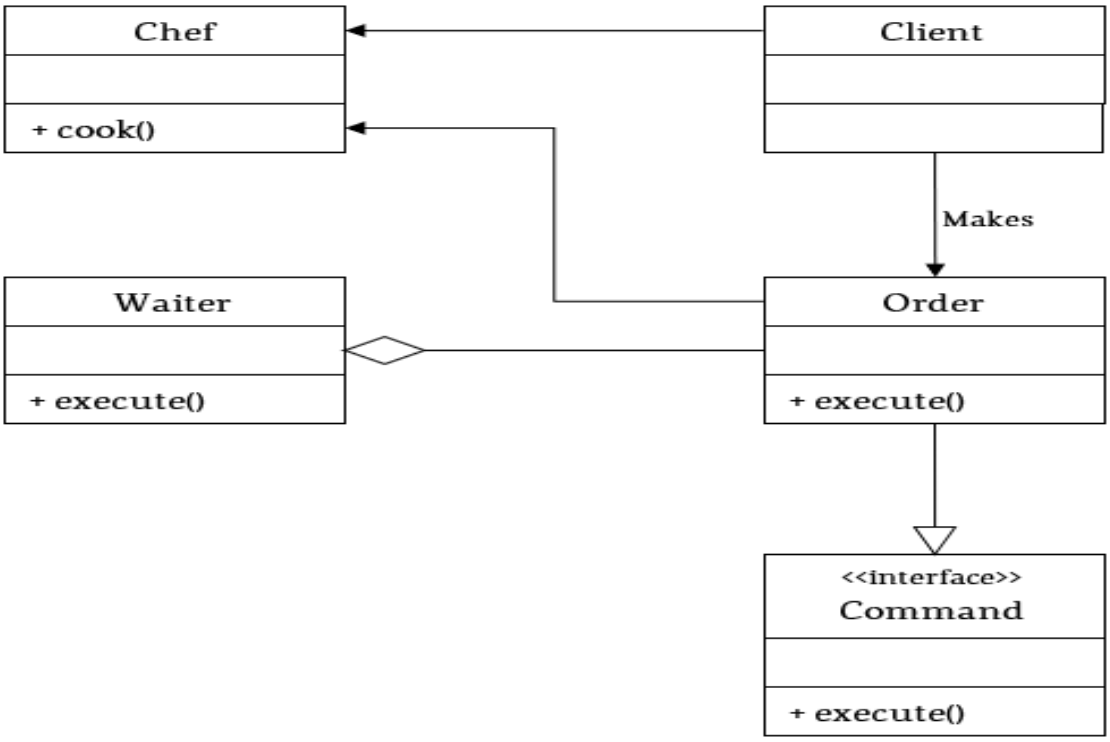
Uhhhh… What's that?
Coupling is the way that two (or more) classes that interact with each other, well, interact. The ideal scenario when these classes interact is that they do not depend heavily on each other. That's loose coupling. So, a better definition for loose coupling would be, classes that are interconnected, making the least use of each other.

The need for this pattern arose when requests needed to be sent without consciously knowing what you are asking for or who the receiver is.

In this pattern, the invoking class is decoupled from the class that actually performs an action. The invoker class only has the callable method execute, which runs the necessary command, when the client requests it.

Let's take a basic real-world example, ordering a meal at a fancy restaurant. As the flow goes, you give your order (command) to the waiter (invoker), who then hands it over to the chef(receiver), so you can get food. Might sound simple... but a bit meh to code.

The idea is pretty simple, but the coding goes around the nose.



Command Design Pattern Class Diagram
The flow of operation on the technical side is, you make a concrete command, which implements the Command interface, asking the receiver to complete an action, and send the command to the invoker. The invoker is the person that knows when to give this command. The chef is the only one who knows what to do when given the specific command/order. So, when the execute method of the invoker is run, it, in turn, causes the command objects' execute method to run on the receiver, thus completing necessary actions.

What we need to implement is;

1. An interface Command

2. A class Order that implements Command interface
3. A class Waiter (invoker)
4. A class Chef (receiver)
   So, the coding goes like this:

Chef, the receiver

```
public class Chef {
        public void cookPasta() {
                System.out.println("Chef is cooking Chicken Alfredo…");
        }

        public void bakeCake() {
                System.out.println("Chef is baking Chocolate Fudge Cake…");
        }
}
```

Command, the interface

```
public interface Command {
        public abstract void execute();
}
```

Order, the concrete command

```
public class Order implements Command {
        private Chef chef;
        private String food;

        public Order(Chef chef, String food) {
                this.chef = chef;
                this.food = food;
        }

        @Override
        public void execute() {
                if (this.food.equals("Pasta")) {
                        this.chef.cookPasta();
                } else {
```

```
                this.chef.bakeCake();
            }
        }
}
```

Waiter, the invoker

```
public class Waiter {
        private Order order;

        public Waiter(Order ord) {
            this.order = ord;
        }

        public void execute() {
            this.order.execute();
        }
}
```

You, the client

```
public class Client {
        public static void main(String[] args) {
            Chef chef = new Chef();

            Order order = new Order(chef, “Pasta”);
            Waiter waiter = new Waiter(order);
            waiter.execute();

            order = new Order(chef, “Cake”);
            waiter = new Waiter(order);
            waiter.execute();
        }
}
```

As you can see above, the Client makes an Order and sets the Receiver as the Chef. The Order is sent to the Waiter, who will know when to execute the Order (i.e. when to give the chef the order to cook). When the invoker is executed, the Orders’ execute method is run on the receiver (i.e. the chef is given the command to either cook pasta ? or bake cake?).