COMP40070 Design Patterns

SHRUTI GUPTA 16200254

Lab Journal

Table of Contents

Day 1 – 5th December 2016: Template and Strategy pattern

Theory

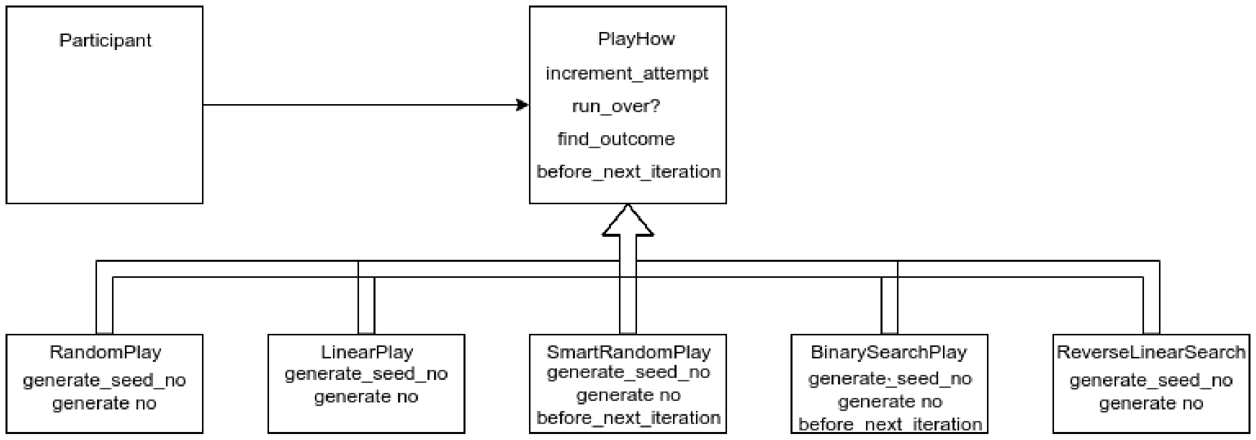
The Strategy pattern is useful in scenarios where the actor or the doer doesn't change; the verb/action doesn't change but the how of the action changes. If one were to model a dancer as a class object, then dance would be one of the methods and dancing in jazz, hip-hop, ballet etc. would be different strategies by which the dancer could dance. In software terminology, this dance format can be changed at runtime i.e. once the dancer is on stage, it can dance in jazz style and if the audience wants hip-hop, it can start dancing in hip-hop style. The Template pattern is useful in scenarios where different actions share a common algorithm. The detail of the algorithm implementation is however shielded from the superclass.

Practical explanation – Participant class

The Participant class has 4 play methods corresponding to 4 different strategies. This is a very obvious indication for using Strategy pattern. So a class called PlayHow is created. All the four strategy classes inherit from this. On observation it is also noted, that all the strategies follow a common procedure and only the implementation details of the steps in the procedure vary. This commonality is very clean and the differences clearly distinct. This is the appropriate place to use Template pattern as well. So the class PlayHow will now contain the template method for execution of any strategy method and the subclasses will define the concrete details required. Initially I defined the template method to follow this procedure: 1) Generate number 2) Increase number of attempts by participant 3) Check if run is over? (Either when number is correct or max number of attempts reached) 4) Print outcome of run when run is over. I know that the first two lines in every strategy method generate a seed number and increases number of attempts. I was trying to think of a way to avoid same method call twice. As I wasn't able to I revised the procedure to now be :i) Generate seed number ii) Increase attempt iii) As long as run is not over do step 1 and 2 iv) Print outcome of run. I defined the functions increase_attempt, run_over? and find_outcome in the PlayHow class. All these are intended to be final methods. I defined the generate_seed_number() and generate_number() for RandomPlay and LinearPlay class. However, I realized for smart random play and binary search, an additional step of manipulating values of upper and lower is required, but this step was same for smart random and binary. So I created another class SmartPlay extending from PlayHow. I added an additional function before_next_iteration to the PlayHow class and implemented it in the SmartPlay class. The strategies SmartRandomPlay and BinarySearchPlay extended from SmartPlay. However, later on retrospect I realized as there is just one point of differentiation between PlayHow and SmartPlay, using inheritance is probably overkill and if the before_next_iteration changed for one it would affect both. So I scrapped it, made SmartRandomPlay and BinarySearchPlay extend PlayHow and added the identical method of 'before_next_iteration' to both these classes. I then added a play_method attribute to the Participant class. In each of the functions play_randomly, play_linear etc I assigned play_method to the instance of the respective class and passed it the parameters self, upper and lower. Since the play method needed to change the value of no_of_attempts attribute, I had to remove it from attr_reader and add it to attr_accessor. I also added another strategy for playing ReverseLinearSearch that starts from the upper bound and decrements one by one until it finds success. This class extends PlayHow too and has only two methods implemented generate_seed_number and generate_number.The final template method and relationship between classes can be seen in the following figure.

Template Method: play

1. Generate seed number *#Abstract*
2. Increment participant attempts *#Final*
3. While step 4 returns false do step 5, 6, and 7. When step 4 returns true move to step 8
4. Check if run is over *#Final*
5. Make any adjustments before next iteration *#Hook*
6. Generate number *#Abstract*
7. Increment attempts *#Final*
8. Find outcome *#Final*

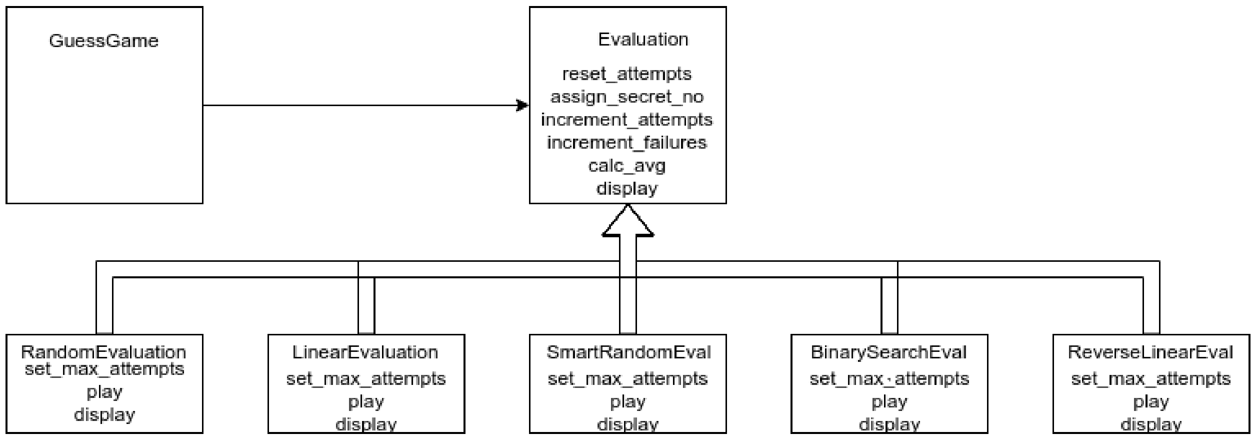


Practical Explanation: Evaluation class and main.rb

After completing this part, I started making changes to the main.rb file. There is a lot of code that has been repeated and even without using patterns; the code could have been made shorter and cleaner by using case statement. I thought of defining a method that takes strategy type as argument and calls the relevant function. This method could then be used in conjunction with the case statement. The code was still very tedious. As I thought of ways around it, I realized that there was potential for applying a combination of Strategy + Template pattern there as well. If you consider a main class representing our Game for e.g. GuessGame then there are different evaluation strategies. Also there is a clear procedure followed by each strategy. I want to point out that in this case since the actual implementation is very trivial this design pattern here might seem like overkill, however, it holds for the concept for evaluating the performance of different playing strategies. So I created a GuessGame class which has a method score_performance and an attribute strategy_method. The attribute strategy_method is changed at runtime to instances of subclasses of Evaluation and the method score_performance calls the evaluate method on it to see the performance of the different playing strategies. Also the older approach was creating multiple instances of Participant class to test performance of different strategies which isn't right. This strategy creates only one instance of Participant class. The superclass Evaluation has an instance of participant class as an attribute which is created when any instance of Evaluation or its subclass is created. All the variables like 'NO_OF_RUNS' and total_no_attempts etc. are moved to the Evaluation class, it's natural home. So there is no need to pass any data or context from GuessGame class to instances of Evaluation or its subclasses. The commonality is captured in the template method 'evaluate'

Template method: Evaluate

1. Reset no of attempts for participant *#Final. As we are using same participant object for all strategies*
2. Set Max attempts for participant *#Abstract. I thought of making this a hook method, as max attempts for three of the strategies is NUM_OF_RUNS * 2 and different for the other two. But I decided against it, because I couldn't think of a logic for which this should be the default logic.*
3. Repeat step 4, 5, 6 for NUM_OF_RUNS times
4. Assign a secret number *#Final*
5. Reset no of attempts for participant *#Final*
6. Participant plays *#Abstract*
7. If outcome of 6 is success, do step 8, else do step 9
8. Increment attempts *#Final.* Move to step 10
9. Increment failures *#Final.*
10. Calculate average no of attempts *#Final*
11. Display *#Hook. I initially made this abstract as the output displays the name of the method. But then I thought of the display method to be analogous to the toString() method in JAVA. It allows you to override but there is a default implementation. So I made it a hook method.*



In both cases, it makes sense to apply the Template pattern first and then the Strategy pattern as that removes a lot of duplicated code.

Only the developer of the template pattern knows the most crucial hook methods to other developers. These should be listed in the documentation for the API.

Day 2 – 6th December 2016: Observer pattern

Theory

Observer pattern is used where one class encapsulates a state. When the state changes you want to notify all the classes that are dependent on this state

Practical Explanation

I decided to address the problem in the following particular order and managed to get it working.

1. Get it working for one participant and one observer(auditor) with push model
2. Get it working for one participant and one observer(auditor) with pull model
3. Get it working for multiple participants and one observer(auditor) with pull model
4. Get it working for multiple participants and multiple observers(police , auditor) with pull model
5. Add both built-in and custom observable module to the participant class
6. Attach one of the observers using the built-in module and one using the custom observable module and see how it affects the system
7. Get the custom observable module working with a proc
8. Create a simulation with a few participants and run the process a 100 times.
9. Post the simulation, do some analyses on the data stored by the three observers (auditor , police and proc)

After adding name, gender, age and strategy to the Participant class, I got rid of the various play_xxx methods. The single play method now based on the value of the strategy method, calls the appropriate play method. I thought to use symbols as values of available strategies and declared them as a constant array in Strategies module. I created the Auditor class with the pull model, added an update function that would take outcome of the run and no of attempts as inputs. If it was success and no of attempts were less than 3 it would print "CHEAT". To test this, I added another playing strategy cheat that knows the answer and tells it in 1-3 steps. After this, to get the pull model, I passed the self-object to the update method. Then I thought that the update method should support both pull and push. This is because there will be a thousand observers but a single observable and it's the observable's discretion whether to use pull and push. So using a check I modified my update method to support both pull and push. To now test this with multiple participants I needed to create dummy data. I created 6 participants with the help of integer.times and arrays, each one of which plays using one of the 6 strategies. To add multiple auditors I initially implemented a RegularAuditor and a SeniorAuditor but I couldn't find much value for them, so I implemented a RegularAuditor and PoliceOfficer class, two auditors with totally different functions. However, both of them needed to check if a particular run by a participant was to be considered cheating or not. This seemed like a right use of module to me so I created a module Supervisory with the method to check cheating and included it in both RegularAuditor and PoliceOfficer class. The RegularAuditor's update method checks for cheating and then stores the strategy used for cheating. This seemed like a good metric to evaluate on. The PoliceOfficer's class checks for cheating and stores the participant object in a hash map of offenders with their offence count. Again this record can be used for analyzing multiple runs and multiple participants' performance. Right about now, it looked to me that an entire simulation process was coming together. So I created a Simulation class with the methods prepare_data, simulate, analyse_by_strategy and analyse_by_age which would make all the participants play multiple times and generate statistics on the information collected by both the observers. At this point, I didn't think it was a good idea to simply delete all of my old code and replace it with a new custom observable module. So I thought to explore what would happen if I were to use both the original and custom observable module. I did that and just changed the method names slightly in the custom observable module. I let the RegularAuditor attach with the built-in observable and attached PoliceOfficer with the custom observable and it worked. This means that if multiple observers for a given object can be grouped into different sets, then the given object can include observable modules for each of these sets, which will have different implementations of notify. I created a Proc object and passed it to my custom observable module's attach_method. That worked as long as I had a check for @observer.class == Proc in my notify method. However, I couldn't pass a simple block to the same method, as it required a different argument type. However, it's never a good idea for an interface to support only entities of one type. So I added separate methods for adding and removing block code. I was also curious to check the true closure property of Ruby that was one of the points of discussions in the class today. So I created a block that would simply update the gender count for each run. Then in the final step of simulation, I used these gender counts to display stats of gender breakdown and it worked.

Day 3 – 7th December 2016: Abstract Factory and Singleton Pattern

Exercise 1: Explanation

This part of the practical is enclosed in folder named Part1&2. For this part of the exercise, I thought of the three products A, B, C as subclasses of a superclass Product. The superclass Product would have a function that would define its job and the subclasses would override it. So it didn't make any sense for the different product classes to have different method names. So I made them all uniform. I created a class ProductFactory with methods to return instances of ProductA , ProductB and Product C respectively as asked. I also wanted to avoid multiple functions call, so I thought of adding a function to the ProductFactory class that would take in an array of class names (in this case ProductA, ProductB etc) and return an array of their instances. Also the code in the FactoryExercise.rb (renamed to main_client.rb) looked a little shabby. To make it look more elegant, in the foo method I retrieve the array of product instances in one call by passing the class names as parameters, then iterate over the array and call their do_my_job function. This is the beauty of polymorphism!!!

Exercise 2: Explanation

This part of the practical is enclosed in in folder named Part1&2 as well. As the lecturer had hinted about using module for applying Singleton in Ruby, a quick Google confirmed the correctness of this way. So I included the module Singleton in the ProductFactory class. The Ruby Docs suggested that I couldn't call ProductFactory.new anymore and should use ProductFactory.instance instead. I tested for this by using a begin–rescue block and noticed it does indeed throw NoMethodError if I use ProductFactory.new. I tested for the singleton property by making two calls to ProductFactory.instance and storing them in two different variables and checking for equality.

Exercise 3: Explanation

As required I created a CoolProductFactory class and changed my ProductFactory to UncoolProductFactory. Also I had to now modify my code to have ProductA, ProductB and ProductC as super classes themselves, from which the Cool and Uncool version of products would inherit. I debated a lot about introducing a superclass for the factories at this point as I wanted to add a method that would return an array of all the products for any factory, and the best place for this method would be in the superclass, but eventually I decided against it as the current requirements didn't demand it and I have a tendency to plan for the future while coding and as the lecturer points out it's not good. I completed all the requirements of this part and passed instance of UncoolFactory and CoolFactory to the Client class and everything worked as expected.

Exercise 4: Explanation

A new family of products implied a new ProductFactory class and a new product subclass for each product. Now there were three ProductFactories and the idea of using inheritance at this point definitely had credibility. You can't let three factories roam around in the system; they need to have a common identity which would be defined by the superclass. So I created a new class DeadlyProductFactory and ProductFactory from which all three ProductFactories inherited. I made ProductFactory singleton. And then surprisingly the code started throwing errors about using new with other ProductFactories. This is a classic example of how a small code somewhere can break the system. Thinking about it later and knowing how Ruby works, this was expected as the superclass now included the module Singleton, all the subclasses now were Singleton, so using new operator was giving errors. I fixed that and voila! The code worked as expected.

Exercise 5: Explanation

No explanation required. Didn't do anything fancy here. There were changes to a lot of classes and I tried to think of ways around it. One possible way was for each class to read a list of products from somewhere and instantiate the product. But I realized that didn't make sense for a new product type and the lecturer did mention that if you add a new product type to any system there would be lots of changes required. So I let it be and moved on.

Exercise 6: Explanation

Now this was the cincher! I didn't understand the complexity first. I just assumed that since my superclass Factory is Singleton, all sub factories are singleton, so done! Later I got more clarity about the actual requirement and understood that there was to be total one instance of any of the factories at a given point of time. This wasn't achievable using the Singleton module, as it would hold one instance of each of the factories throughout the program. So I removed the Singleton module and tried to do this manually. I tried to think in multiple directions to achieve this, maybe a central class that would observe all factories and control them? I tried to sketch it out and it just made the program complex. I also thought of the lecturer's solution, but I was tempted to try out a different way. Being a JAVA coder, it was hard to deal with absence of constructors in Ruby. Many of my solutions involved constructors. So I finally thought that every class inherits the new method from its ultimate superclass and overrode it in the superclass ProductFactory. The ProductFactory has a class variable @@current_factory which holds an instance to the ProductFactory. In the overridden new method, I make a super call and hold the result in a variable. I then check the type of the result, if it's ProductFactory then I instead return @@current_factory else I return the result. The class also has three functions to return the other factories. Whenever these functions are called @current_factory is set to instance of that class. This ensures that any point of time, there is just one instance amongst all the Factory classes. The solution might seem hacky, it did to me too, but then it's a one-line change and I am not breaking any formal contract of the language. Ruby docs say 'new' returns an object, which it still does, the code is not doing anything unexpected. All the functions are named in a way that tells the client which factory is being returned.

Exercise 7: Explanation

This part of the practical is enclosed in the folder named OneFactory. This one was pretty simple so I am not sure if I did what was expected. I created a singleton factory class with a method create_product that takes class_name as parameter and returns an instance of that class_name. In my main, I initialize different clients with an array of combination of different product class names. In the client's initialization, I use a map block to obtain an array of instances of each of these class names. The output of the file clearly shows results for cool, uncool, deadly and hybrid clients. According to me, this is probably one of the worse ways. First of all, the names of the class need to be known beforehand. Secondly if there's a spelling mistake but a class exists with that name, it would instantiate that and the system would either break or behave unpredictably.

Exercise 8: Explanation

This one is tricky as well. If there is no reference to an object anymore, Ruby GC takes care of it and developers don't need to worry, so I am confused about the purging part. I anyway put a section in my main file that creates a client with the UncoolFactory and change its factory later to DeadlyFactory and it works fine. Only addition I did is to create a setter for the @factory attribute of client. Once it's set to a new value, the products list is created. I guess this is what the requirement meant by purging and replacing existing objects.

Day 4 - 8th December 2016: State and Decorator pattern

Exercise 4: State Pattern

Part 1: Explanation

So as the requirement, I created a class PersonType and let Child, Adult and Pensioneer extend from it. When the Person class is instantiated its state variable is set to an instance of Child. The method increment_age allows age increment by only one, and if age reaches one of the maturity ages = 18 or 65, the state variable is reassigned to an Adult or Pensioneer instance respectively. Also, instead of defining the methods vote, buy_buspass and conscript in the Person class and repeating the method call on the state variable, I thought this scenario to be a better application for using Ruby's Forwardable module with the def_delegators method. So I defined these functions in PersonType and its subclasses, and added them to the def_delegators. Also, the three types of Person sometimes have totally different responses to the three methods, or like in the case of vote and conscript, at least two of them have the same response. So to avoid code duplication, for every method, whichever version had more majorities, I put that version in the super class and let the subclass override it, if it had a different response. For e.g. the vote method that outputs “vote accepted” went into the superclass and the child subclass had to override it. For the test cases, as I didn’t have any idea about how to write test cases and it would have been too time consuming to sit through a tutorial. So I basically tried to emulate the participant_spec.rb file from the Day1 practical and I am not sure if it’s the best way, but all the test cases worked, so far so good. I have added cases to check the instance of the state variable of the person object as his age increases. For each person type, I have added test cases, to see if the three methods give the appropriate response.

Part 2: Explanation

Adding a Teenager class shouldn’t have required major changes. Creating a new subclass of PersonType and adding a line in the increment_age method should have been enough. However, with the features of Teenager class I realized that the majority for different versions of the three methods: vote, conscript and bus pass was now lost. There was pretty much a tie for everything. So then I thought, that by default a general person should be able to do anything, if a particular person type is not able to, it should override the method. So that’s how all the positive versions of the three methods, went in the superclass, and the subclasses overrode them as and when required. I also added similar test cases as before for the Teenager class to my spec file.

Part 3: Explanation

It’s time to add a new method. So as I had decided in part 2, this method went in the PersonType superclass and Adult and Teenager classes had to override it for their inability to buy a medical card. I also added this new method to the def_delegators section of Person class. Using def_delegators is really handy as it avoids adding new methods/ changing existing methods in several places. Just change one place and voila! You are done.

Part 4: Explanation

This part of the practical is represented by the files in folder named Part4&5. First of all, I am in disagreement that the natural home for state-transition code lies in the State class. The state is an attribute of the Person class, so following encapsulation; all changes should be done from within Person class. If you take the TCPConnection example in class, today, the state of the connection is a property of the Connection object, so the natural place for the Connection to change its state is in the Connection class. The only situation I see this to be a good idea when the states are in sequential order. In that case, each state would know which is the next natural state and will change that. However, what I have tried here is for the superclass PersonType to hold an array of the states in a sequential order. The event for state change is still determined by the Person class, but when a state change is required it calls the change_state function of the PersonType hierarchy and passes its context object to it for the PersonType hierarchy to be able to change the state of the Person class. In the change_state function, I would determine the current class from the context object, get its index from the states array, determine the next state in the array and changes the state object to that instance. All the test cases still pass.

Part 5: Explanation

This part of the practical is represented by the files in folder named Part4&5. Singleton could have been used to allow for only one instance, but that wouldn't give lazy instantiation. So I tried another way. In my PersonType superclass, where the state change is happening, I changed the states array to a hash map with initial values as 0 for each class. Then when a state is being instantiated for the first time, it creates an instance and stores it in the hash map. Next time, when a request for the same state is made, it retrieves the previously created instance from the hash. I have also added test cases to ensure that the states are created only when they are required and state objects from different person objects have same object ids.

Exercise 5: Decoration Pattern

Part 1: Explanation

This one was a bit tricky. First of all, in the JAVA based approach, a Decorator class and the normal class extend the same interface. This is because JAVA is a typesafe language. However, in Ruby you can pass anything to anything (Note: I don't think that's a good idea). So using inheritance wasn't necessary and the more I thought about it a condiment couldn't be a coffee. So I decided to not use Inheritance to define the relationship between Coffee and Condiment class. Decaf and Espresso extended Coffee. Functions for description and cost were defined in the Coffee class. The Condiment class is extended by Milk and Sugar with the relevant attributes and functions. The constructor of the Condiment class takes in an instance of the Coffee class @coffee. This instance is going to let the Condiment class and its subclasses pass request to the Coffee class as well. Again looking at the participant_spec.rb file from Day1 practical I populated a coffee_spec.rb file with test cases and they ran fine. I was a little confused about how the flow of code is. So I did a triple chaining while creating the instance (Decaf with both Milk and Sugar) and realized it's from outside-inside and propagates again from inside-outside.

Part2: Explanation

By my experiment from part1, I was pretty confident about how delegation actually works. However, it was tricky to think of an example in regards to coffee particularly. Finally I came up with packing_cost, as that would be accumulated from each component in the coffee. Initially it would be assigned 0(i.e. on the way in). Then it would delegate and get the packing cost of the coffee. Then it would add to it, the packing_cost of the condiment (on the way out) to return the final results. I have tried to show this with relevant print statements in my code.

Part3: Explanation

Don't know if this was about design patterns. I just added more tests to my coffee_spec.rb file

Part4: Explanation

Now this was something very interesting. Ruby does offer a versatile range of modules. Using the Forwardable module makes life a lot easier for so many developers. It lets you cut away all those one liner methods, just calling another method. However, I couldn't find a great example to showcase here. But the final instance of whatever we create will be of Condiment class, but we might want to perform operations on the Coffee class only. I guess this is a consequence of using patterns. Data abstraction and encapsulation gets violated. Moving on, I added two methods to warm up coffee and get a cup to the Coffee class and added delegators for it in the Condiment class. I found a much bigger use case for def_delegators in the State pattern exercise. I have used it there more elaborately.

Day 5 – 9th December 2016: Final Exercise (5 Patterns)

Pattern 1: Composite pattern

This one required more of rough designing on paper before beginning and yet the design changed as I coded and encountered coupling between classes. According to the initial design, I created an interface Furniture with methods name and cost. I also added two default functions addFurniture and RemoveFurniture here (JAVA8 lets us add default functions to interface now). I created three classes Block, Door, Window that implement the interface Furniture and overrode the getName and getCost functions. Then I created the composite class AssembledFurniture with a static method that would take multiple furniture objects and return an instance of AssembledFurniture. As I was doing it, it occurred to me that composite says we should be able to treat a single object or a composite in the same way, and AssembledFurniture couldn't be instantiated the same way as the other leaf nodes. However, clearing this doubt with the TA made me realize that to the client the usage is still the same. The composite pattern here has hidden the details of the separate implementation and that's the crux of the matter. Having understood that, I added the name and cost methods that would iterate over its component furniture objects and return a concatenated string of all their names and a total cost. Post this I started coding for the Invoice class. The Invoice class according to the requirements should hold a list of Furniture objects. But I realized that doing this would create a tight coupling between the Invoice class and the Furniture interface. This invoice class would then be able to work only with Furniture objects. However, the Invoice class doesn't have this dependency because of context. In the real world, an invoice can be generated for any item, as long as it has a name and a price. So I created an interface Buyable and moved up the methods name and cost to this interface and so now my Invoice class held a list of Buyable objects. I made the Furniture interface extend the Buyable interface as a furniture object could be purchased. This is a classic example of how the context forces the design to change. However, now that name and cost methods went in the Buyable interface, my Furniture interface was empty. A Furniture object can have many attributes. However to keep things simple at the start, I added only width and height and propagated the change in its implementing classes. When I wanted to test this, I realized I should have written the test cases first which is a better practice. Anyway, I wrote 5 test cases, 3 to test the instantiation of the leaf nodes, 1 to test the instantiation of the composite class and 1 to test the creation of invoice.

Pattern 2: Builder pattern

I didn't have to think about which pattern to use next. While coding for the Invoice class itself in part1, it had occurred to me that it's a perfect use case for Builder pattern. The Invoice is a generic class that can generate invoice for any list of buyable objects. A typical invoice also has other attributes such as shop name, client name, discount, payment mode and total cost. However, making all this mandatory would make this class too complex for a client. Using multiple parameterized constructors or setters would make the code unreadable. So using a builder pattern here would take care of both the problems by giving a fluent interface as well as flexibility. Before I begun however, remembering my lesson from the last part, I created the test cases first. Point to note though I had to go back and change them as I proceeded because of changing code. Ideally this shouldn't happen as everything should be thought of in the design phase, but that's idealistic not realistic. Anyway then, I used a static inner class InvoiceBuilder to achieve this. Every method in this inner class would set some attribute and return an object of InvoiceBuilder back. Using which the client could set some other attribute. Finally when the client is done, the generateInvoice method can be invoked on the InvoiceBuilder which will return a new instance of the Invoice class. Also it's necessary to ensure that only InvoiceBuilder is used to create an invoice, otherwise the program might behave in unexpected ways. Also giving many options to instantiate will confuse the client more than it will help. So I made the constructor of Invoice class private. However, a client might want to get or set the values after he's constructed the invoice as well. So I created setters and getters for these variables. At this point I realized, that if the client invokes a get on some attribute, he's not set, it will return null. I wasn't sure of the best way to handle this. I could let it return null and let the client worry about it, I could check if its null and throw a RuntimeException or I could check if its null and consume the error silently. After having a discussion with the lecturer on this, I decided to go with empty strings for all the uninitialized values. This makes more sense if you consider a Person class that can have multiple attributes like Home Tel, Work Tel, and Mobile. But a person might not have all three of this. And if the class started giving null values when asked, it would lead to trouble. So an empty string is the best idea. I changed the toString method to represent this new information and finally ran all the test cases.

Pattern 3: Bridge pattern

I suppose the program to be a reconstruction of a Furniture Shop because of classes like Furniture, Window, and Door etc. It acts as a client of the Invoice API. So now the Furniture Shop has classes for different furniture like Window, Door etc. As the shop grew, the owners observed that some of the elite customers also wanted different architecture styles like Victorian, Modern etc. Then the owners and the developers of the Furniture Shop got into a discussion. The owners wanted the developers to have one class for each combination for e.g. a Victorian Architecture Door , a Modern architecture door , a Simple Architecture Block , a French Architecture Window and so on. The developers however realized by now, what a huge mess of unmaintainable codebase it would lead to. So the developers committed to the owners for fulfilling the requirement all the while internally thinking of a better way to get themselves out of this mess. Fortunately for them, one of the developers on the team had just attended a Design Patterns class and was excited to put to use the patterns taught in the class. Now , one of the main design principle is to always code to an abstraction/interface and not to a concrete implementation, however , in this case the abstraction itself was always varying. Only one pattern out there allows putting abstraction and implementation in two separate class hierarchies and thereby varying both of them. So instead of having one class for each combination of concrete furniture and an architecture style, the Furniture abstraction is composed with the Architecture abstraction. In the codebase, the interface Style.java represents the architecture abstraction with the function applyStyle that every concrete architecture style class would have to implement. The abstraction Furniture now contained an object of this interface and a function that would call the applyStyle function on the Style object. Initially the Furniture.java file was an interface, I however had to change it to an abstract class to allow for this, and all the child classes had to make a call to the super constructor first to instantiate the Style object. This worked fine for all leaf nodes , but I was stumped at what to do for the AssembledFurniture class – the composite which could be instantiated only by passing in one or more Furniture leaf nodes each of which will already have an style object associated with it and didn't make sense for the composite to have a style object. To get around this I inserted a call to the super constructors in the private constructor of the AssembledFurniture class with a Default Style object. This line of code isn't meant to do anything. It's not a clean solution, but I suppose when you are trying to fit multiple design patterns together, such situations would often arise. Also, there was the context that not every customer would want to choose a style for the furniture, making another use case for the Default Style class which would use SimpleStyle as the default style unless changed otherwise.

Pattern 4: Façade pattern

The owners and the developers were happy with the solution with the Bridge pattern as now they could add more and more furniture objects without having to create multiple combinations for each architecture style and add they did. As the shop became more popular, they added more furniture objects liked Bed, Closet, Sofa, Lamp, and Table. The Furniture Shop was becoming a one-stop destination for all furniture objects a new unfurnished house would need. The owners saw a big business opportunity here and a brilliant idea of starting a small Interior Decoration unit right in their shop wherein the client need not bother with nitty-gritty details of furniture selection. All the client had to do was provide dimensions of the room they were furnishing and the type of Architecture Style they wanted and leave the rest to the Interior Decorator. When this requirement reached the developers team, initial knee-jerk solution was to have different classes for each combination of furniture objects that could exist. However, as the number of furniture objects had become very large and would continue to grow in the future, it led to a very complex system. However, the clients needed to be shielded from this. They ought to be provided with a simple interface to a complex sub-system i.e. the Façade Pattern. The InteriorDecorator.java class acts as the Façade in this codebase. It exposes methods like buildLivingRoom and buildBedRoom, both of which take in dimensions of the room and optionally the architecture style. It then instantiates furniture objects that would go in the particular room keeping in mind the dimensions proportions. In future, when new furniture objects are added to the shop , more functions can be added to the Façade class to buildKitchen , buildOffice etc.

Pattern 5: Chain of Responsibility pattern

Good news! The Interior Decorator unit is a huge hit with the customers and more and more orders are pouring in. However, as the business is expanding, some customers are having certain grievances with the furniture delivered to them. The owners are very distressed about this as bad mouth-of-word could harm their image. So they want to ensure every customer grievance is addressed with proper care and the customer is happy at the end of the day. So once again they turn to the team of developers to set this up. The developers were more than happy to set up a queue of customer service executives to handle this. However, the owners weren't very pleased with the idea. They understood human dynamics and knew that the more severe a customer's issue was, it should be handled by a person of higher authority to ensure all is well at the end of the day. The developers were excited about taking this approach because it meant they could use another pattern – The Chain of Responsibility pattern. In this pattern a request passes through a chain of objects until it is handled by one of them. Each object in the chain implements a common interface with each concrete implementation specifying how to handle the request. If a concrete implementation can't handle the request, it passes it to its next object. In this codebase a chain of customer service executives were created CallExecutive -> Manager -> Supervisor -> Owner. All of these extend the abstract class CustomerService. I had to choose between making the CustomerService file an abstract class or interface. I went with abstract class so that the code snippet for setting the next object in the chain did not have to be repeated in every concrete implementation. The InteriorDecorator is the client of the CustomerService API and is composed with a CustomerService object. At the start, it is set to the CallExecutive and whenever an issue is raised with the InteriorDecorator shop, it delegates it to the CustomerService object. This issue passes along the chain according to the severity of the issue. Low: CallExecutive , Moderate : Manager , High : Supervisor and Severe to the owner finally.