DnD Character Creator

By : Jason Spencer, Dheeraj Singavarapu, JJ Javier

Introduction

Project Background

Dungeons and Dragons, often abbreviated as D&D or DnD, is a tabletop roleplaying game that takes place within a fantasy world/environment. This world is filled with magic, monsters, and various types of adventures that can be either bought as a preset, or can be entirely made from scratch. The game and story of dungeons and dragons is guided by, what is called, a dungeon master or game master. Players will have assumed roles to play as they go in their game which is where the dungeons and dragons character creator takes place in its importance.The Dungeons and Dragon character creator is a program that brings the user through the character creation process in a beginner friendly way. The user is guided through the process by answering various questions depending on how they want their character to be. The system supports the choosing of a character's class, race, job, background, their inventory, their spells, and their skills.

Key Features To Develop

As far as key features to develop goes most if not all of GUI is ready to be used however the features we intend to develop deal with the backend of the system. As part of the project outlines we were assigned to pick 4 design patterns learned in class to implement into our system. The design patterns we chose to implement include: Builder, Mediator, Singleton, and Factory. In order to develop these features into the system we needed to work on modifying the Character.java class within the system. This java class is used to bring together the created objects from the use of the builder and factories, as the character class is the mediator from all the singletons. Our plan of action to modify those features will consist of logic for how the character is created, as the system previously, all the logic that was made into creating the character was all squished together into one giant character class, whereas the updated and improved system follows the SOLID principles as it now has multiple purposeful classes each used for different parts of the character in order to properly create the character the user desires.

The interface is user friendly allowing the creation to be done with ease. This includes text boxes, dropdown menus, and buttons that will create the different customization options used in order to create a character properly.

The user will have customization with a range of options with a character's typically called class, but in this case we're calling it their job to not confuse it with a java class. And as well as a character's race, background, spells if applicable based on their job and race, weapons, inventory and skills which will also be based on their race, job, and background.

This application also has comprehensive rule integration as the application is set with the latest rules of dungeons and dragons to make sure that each character is compliant with the dungeons and dragons guidelines.

Potential Users

Potential users include people who may play Dungeons and Dragons, clients interested in a character builder as a whole, or anyone that wants to use this system as a base for developing a similar system or one that has similar features. In addition to dungeons and dragons players, dungeon masters or game masters may need to use this application to create non-player characters for their campaign they make. Any sort of game designers, authors, writers, people who are passionate about role playing games, artists, illustrators, and even potential cosplayers are able to use this sort of system to figure out characters and what fully goes into making a complete one.

System Design

Design Pattern 1 - Builder Design Pattern:

The first pattern we will be delving into is the builder design pattern. The builder object is used to help create product objects. According to the lecture the product object has many parts that aren't always available at the same time. It is also useful when you prefer not to write constructors with a long list of parameters as the client can make easy mistakes. Another feature of builder that proved useful is its ability to encapsulate. It encapsulates the construction logic of complex objects which makes it easier to change or extend the construction process without having any effect on the client code that uses the object. The builder pattern also allows for consistency and allows for the building of objects to occur in specific steps which reduces the risk of creating invalid objects.Within our system the point of the builder design pattern is to take the user input and then create an instance of a character. This input is passed as a string to the factory design pattern. The factory design pattern then creates a part of a character being either a job, race, or background and then it passes that back to the builder as an instance of a specific character trait. Once the character variable is done being created that variable is passed to the character class which instantiates it as a completed version of the character being made. The system before had methods of creating characters spread throughout the entire system and was somewhat of a mess. However after implementing the builder pattern we were able to move the logic for creating a character into a single area. The builder design pattern also helps satisfy some of the solid principles we have learned. The builder helps satisfy the single responsibility principle as it solely has different classes for different purposes rather than the previous system where all the building and putting together was done all in one class. The builder also was one of the better choices for our project as we are creating a single character, the builder helps focus on bringing together the different parts of the character together and putting that built segment back into the character class.

Design Pattern 2- Factory Design Pattern:

The second pattern we will be discussing is the Factory design pattern. The factory design pattern creates objects of the same general type according to the professor's lecture. Through personal research and overall understanding of the method we have come to understand that this design pattern is used to create objects without exposing the creation logic to the user/client. It also allows for creating objects and allows the subclasses to decide which class to instantiate. These features proved useful to us as our system needed a way to create objects without having the user see it on their end. We needed the factory to create certain traits of the entire character before it was fully built. In the previous system design the character class hosted all of the creation for a character however after implementing the factory design pattern we moved some specific character traits into their own factories. These factories create parts of the character in a separate area compared to how before it was all done in one area. This made things spread out more and more understandable. The factory design pattern is an overall creational design pattern used to by the use of an interface intermediate layer class creating objects. This overall enables extensibility, flexibility, and makes it easier to maintain as a whole. The factory was the best choice for our character creation because our goal was to not only have the functions properly being used and with purpose, but to also encapsulate the logic of the character creation. We also needed to be able to support multiple created products, so with that we had used multiple factories in order to create the multiple objects in order to use those objects for a later purpose in the creation of the character as whole. Not only does this help with creating multiple objects, but this also allows any customization we want as part of the classes and objects we create. With the use of a character creation, the client would typically want customizable options so that they really get the feel of creating that character from scratch. The factory overall achieves these goals and does so in a way that this project and application is nowhere near closed, but open for extension for any additional customization a user would like to see implemented or that we may think would be a great addition to add to the project. However though there are many advantages like decoupling, extensibility, and encapsulation, there are also some disadvantages like the potential increase in complexity in what we decide to add and use, overuse of the same coding structure and process within methods to create objects for the character, and possibly too difficult to test. But with the creation of the product in mind, we now have to deeply consider the SOLID principles as learned during the lectures. With the factory, the single responsibility principle is taken care of as now we separate concerns by the use of different classes each with their specific use and purpose towards character creation, the open-closed principle is taken code of as the logic itself is closed for modification, but open to extension, the liskov substitution principle is taken care of as the use of the factory design pattern has it to where any object returned by the factory method can be substituted with any of its subclasses, and the client code will continue to work correctly, the interface segregation principle is being taken care of as the client/user will have no direct use of the source code, but with the use of the interface will allow the client to utilize them properly, and finally the dependency inversion principle that allows for loose coupling, flexibility, and testability.

Design Pattern 3- Mediator Design Pattern:

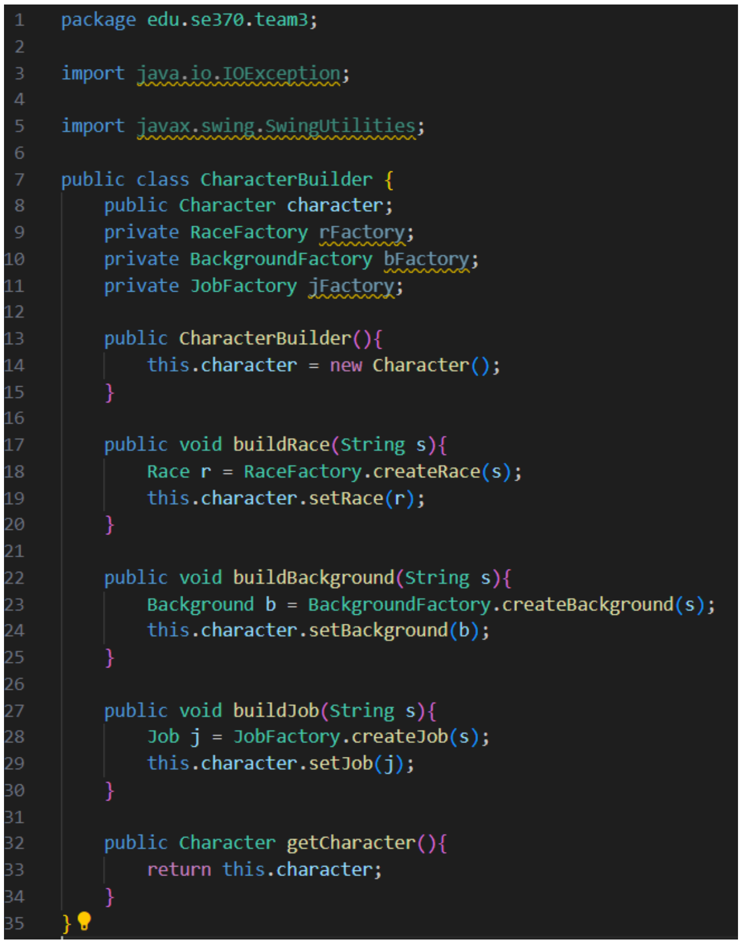
For our third design pattern we have the mediator design pattern. The mediator design pattern helps encapsulate the communication between other objects. In this case with our project the mediator would be the character class that is used between six other key classes that create objects that the character class will interact with and use to create a proper character. The mediator class, which is the character class, will be the mediator between the race class, job class, skills class, stats class, background class, and the spellbook class.The design pattern is an important behavioral pattern as it enables the decoupling of objects by introducing that layer so all the interactions happen within that layer. If the objects created by the different classes were to communicate with each other directly, then the system components would be too tightly together making it hard to maintain and unable to allow extension to the application as a whole. The mediator class in this case receives the messages from other classes and determines how to coordinate everything from what it receives. The mediator class also then invokes the appropriate methods created in order to carry out the functions that are required for the character creation process. In the previous system before updating it with the design patterns, there was no use of a mediator class, but rather everything was already just made and funneled through one giant class filled with many setters and getters in order to fully create a character. This limited the ability to be extensive in what was made and could potentially be done. This also didn’t encapsulate the code from the user, but showed them where things were being done while also violating the SOLID principles, particularly the single responsibility principle (SRP) as everything was put into one class rather than multiple classes having a single responsibility and utilizing those to properly create the character. In this current system the mediator, character class, does exactly what was mentioned about the overall functionality of the mediator design pattern as it will receive the different messages from the different classes like the background, race, stats, and others, and invokes the necessary methods in order to continue the character creation process. The mediator was one of the better choices for our project because it allowed us to continue to use the code that was already made for the previous system, but now we were easily able to separate everything to their own rightful class so that the separation of concern was proper and not a single class had more than one purpose of the character creation.

Design Pattern 4- Singleton Design Pattern:

For our fourth design pattern, we have the singleton design pattern. The singleton design pattern ensures that a class only has one instance created and provides use to it back into the mediator character class. The key components of the singleton design pattern is its static member as the singleton employs it as the static member ensures that the instance is created only once, and the private constructor makes sure that the class itself has control of the instance process and blocks out anything else that tries to do so. Advantages of the singleton design pattern includes flexibility based on the use case, thread safety as the instance will only be made through the singleton class, and the reduction of memory usage. The singleton design pattern was one of the better choices for our project because it ensures that objects/instances created from classes for the character will be made once and only once. This way there won’t be any interruptions, exceptions, or any memory leaks in the code, but that each object is sure to make only one of its kind that will be later on used in the mediator to continue the overall character creation operation. With the singleton design pattern added this also helps with some of the solid principles as it’s a single responsibility as singleton only allows itself to call and create what's needed once and the singleton design allows extension by use of that created object, but doesn’t allow modification to what it already does.

System Implementation

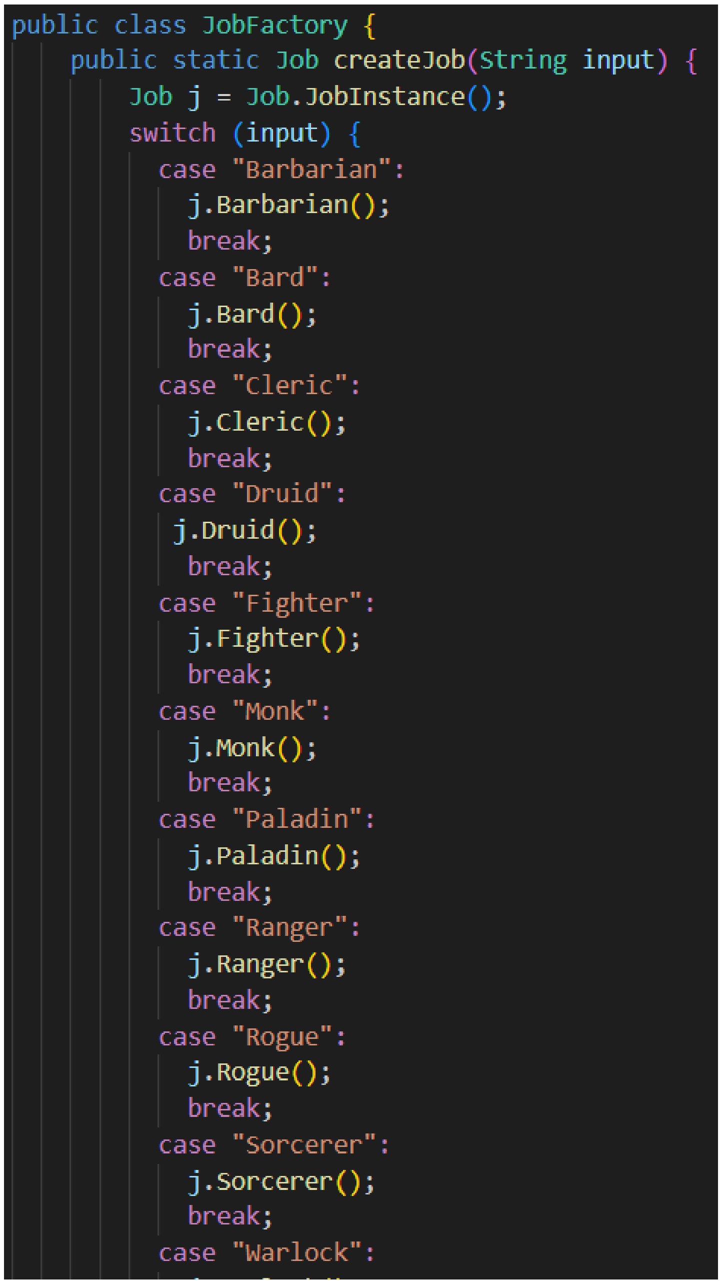
Design Pattern 1- Builder Design Pattern:



```java
package edu.se370.team3;

import java.io.IOException;

import javax.swing.SwingUtilities;

public class CharacterBuilder {
    public Character character;
    private RaceFactory rFactory;
    private BackgroundFactory bFactory;
    private JobFactory jFactory;

    public CharacterBuilder(){
        this.character = new Character();
    }

    public void buildRace(String s){
        Race r = RaceFactory.createRace(s);
        this.character.setRace(r);
    }

    public void buildBackground(String s){
        Background b = BackgroundFactory.createBackground(s);
        this.character.setBackground(b);
    }

    public void buildJob(String s){
        Job j = JobFactory.createJob(s);
        this.character.setJob(j);
    }

    public Character getCharacter(){
        return this.character;
    }
}
```

The above code snippet is the CharacterBuilder class. This class is responsible for collecting the information the user provides in a character class, then transfers it to a finalized character class at the end of the program. The code is neat and conforms to the Single Responsibility Principle of the SOLID principles. It has 4 operations; buildRace, buildBackground, buildJob, and getCharacter. The build operations are the core of the design pattern, they are the functions that are called when the user has selected their race, background, or class. They utilize the respective factory of that type of class to create the object, then it is stored into the builders instance of the character.

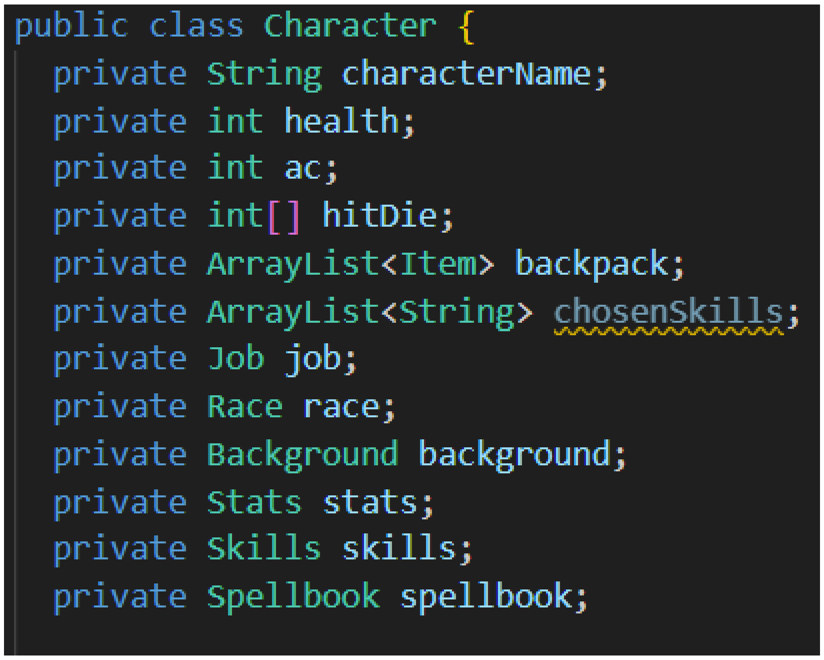
Design Pattern 2- Factory Design Pattern:



```java
public class JobFactory {
    public static Job createJob(String input) {
        Job j = Job.JobInstance();
        switch (input) {
          case "Barbarian":
            j.Barbarian();
            break;
          case "Bard":
            j.Bard();
            break;
          case "Cleric":
            j.Cleric();
            break;
          case "Druid":
           j.Druid();
            break;
          case "Fighter":
            j.Fighter();
            break;
          case "Monk":
            j.Monk();
            break;
          case "Paladin":
            j.Paladin();
            break;
          case "Ranger":
            j.Ranger();
            break;
          case "Rogue":
            j.Rogue();
            break;
          case "Sorcerer":
            j.Sorcerer();
            break;
          case "Warlock":
```

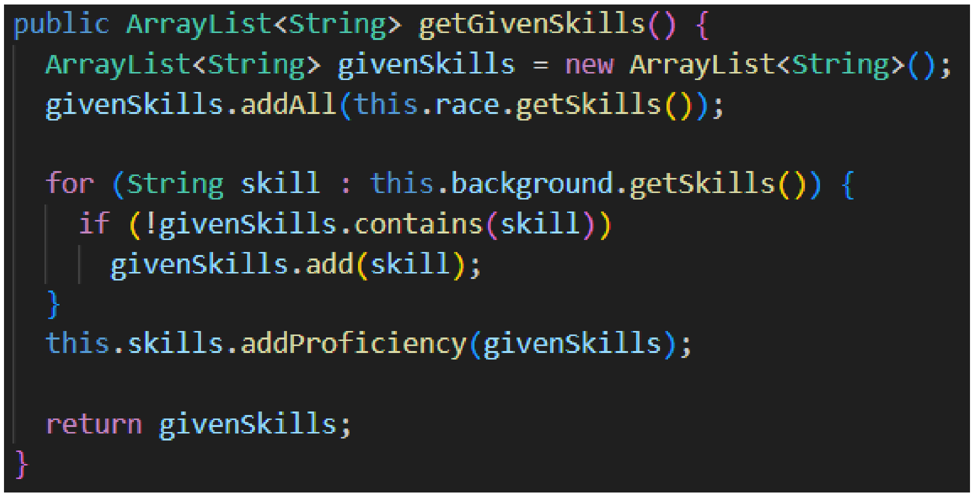
The code above is from the JobFactory class, showing how our program uses the factory method to create the complex components of a character. There are three types of factories in this program, a job factory, a race factory, and a background factory. These specific classes have factories because there are many different types of them that the user can choose, so we needed a way to create only the object the character was going to have, rather than create everything. The factories all work the same way, taking an input as a string, then creating a class based on that string. The factories are integrated within the builder, as each of the build functions of the CharacterBuilder are called, the factory makes the object that is required.

Design Pattern 3- Mediator Design Pattern:



```java
public class Character {
  private String characterName;
  private int health;
  private int ac;
  private int[] hitDie;
  private ArrayList<Item> backpack;
  private ArrayList<String> chosenSkills;
  private Job job;
  private Race race;
  private Background background;
  private Stats stats;
  private Skills skills;
  private Spellbook spellbook;
```

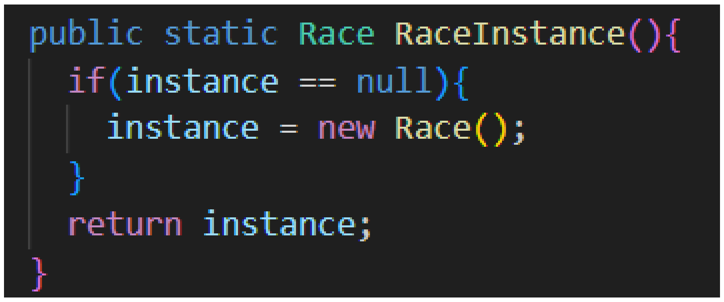
The Character class acts as a mediator between all the different components that make up a character. It has an instance of every piece of data needed to make a character, as well as functions to handle the interactions between each of them. For example, the race and background of a character will influence the skills they have. Without the mediator pattern, these classes would awkwardly communicate with each other with logic for all of them strewn everywhere.



```
public ArrayList<String> getGivenSkills() {
  ArrayList<String> givenSkills = new ArrayList<String>();
  givenSkills.addAll(this.race.getSkills());

  for (String skill : this.background.getSkills()) {
    if (!givenSkills.contains(skill))
      givenSkills.add(skill);
  }
  this.skills.addProficiency(givenSkills);

  return givenSkills;
}
```

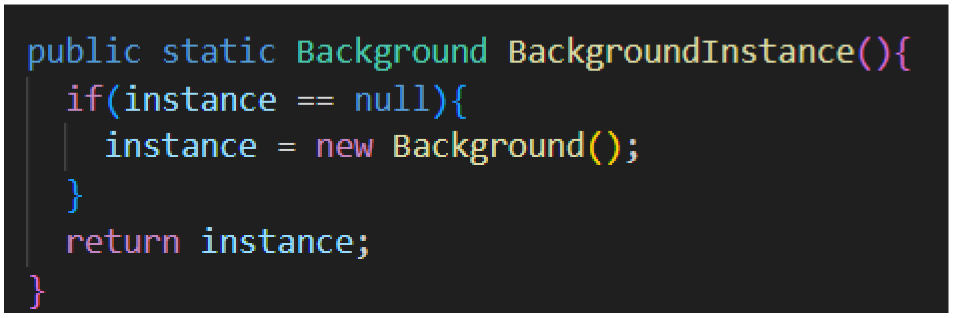
This function in the character class compiles the skills a character will have from their race and background, and adds it into a list. This list can then be added into the chosenSkills list, forming a mediator between Race, Background, and Skills.

Design Pattern 4-Singleton Design Pattern:



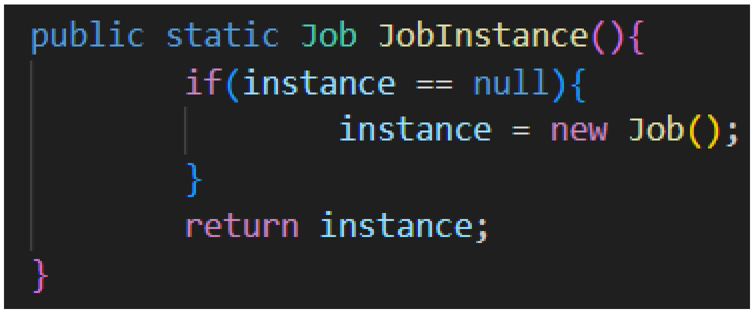
```
public static Race RaceInstance(){
  if(instance == null){
    instance = new Race();
  }
  return instance;
}
```

Here is an example of the implementation of the singleton design pattern in our project. The classes Race, Job, and Background all use the singleton design pattern in a similar fashion. Each class has an instance of itself as a member variable, set to null. The instance function is static, because it needs to be called as a constructor. To create an object of that type, Race, the RaceInstance function must be called. This checks if the instance variable is null. If it is, a new object is made and is returned. If the instance has already been created, that object is returned.



```java
public static Background BackgroundInstance(){
  if(instance == null){
    instance = new Background();
  }
  return instance;
}
```

This exact same pattern is also followed by the Background class.

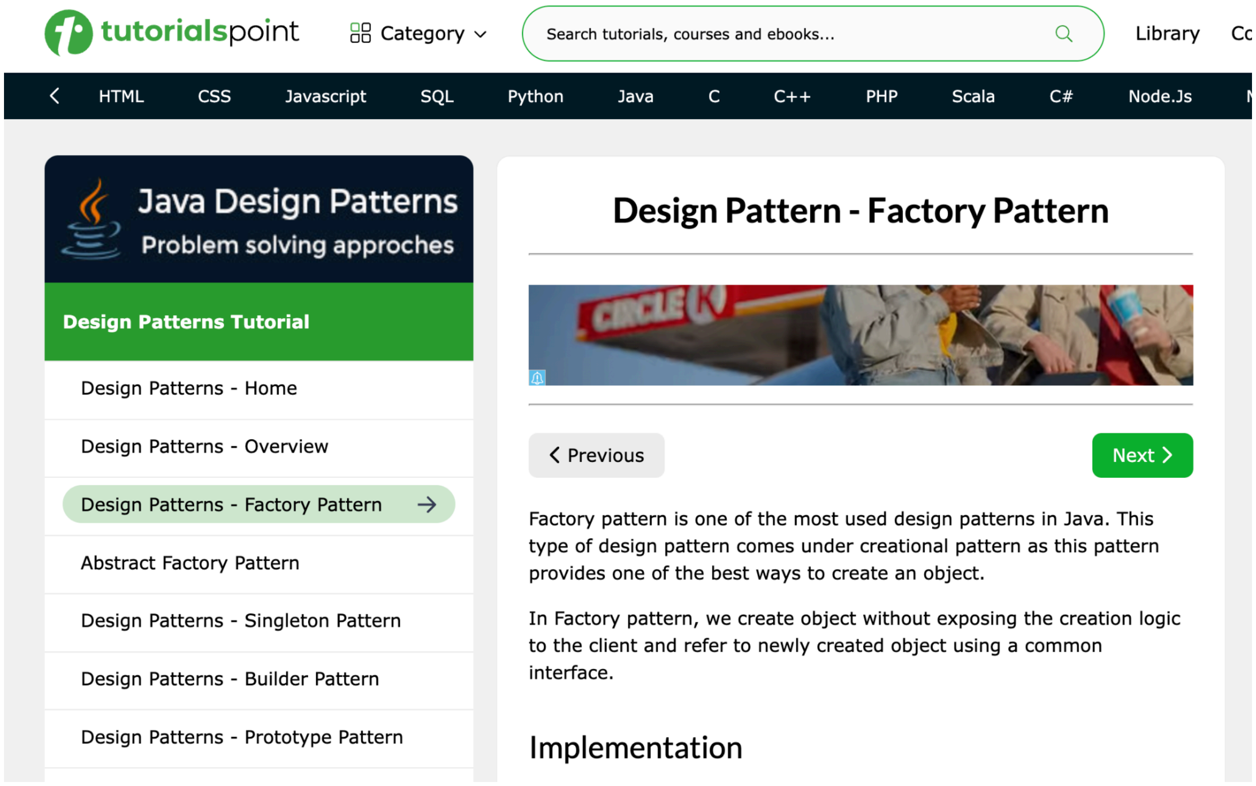


```java
public static Job JobInstance(){
        if(instance == null){
                instance = new Job();
        }
        return instance;
}
```

The Job class is the third and final class that also uses the singleton design pattern.

Utilization of external learning resources to complete the project:

TutorialsPoint-


tutorialspoint
Category
Search tutorials, courses and ebooks...
Library
Cc
HTML
CSS
Javascript
SQL
Python
Java
C
C++
PHP
Scala
C#
Node.Js
Java Design Patterns
Problem solving approches
Design Patterns Tutorial
Design Patterns - Home
Design Patterns - Overview
Design Patterns - Factory Pattern
Abstract Factory Pattern
Design Patterns - Singleton Pattern
Design Patterns - Builder Pattern
Design Patterns - Prototype Pattern
Previous
Next

Design Pattern - Factory Pattern

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

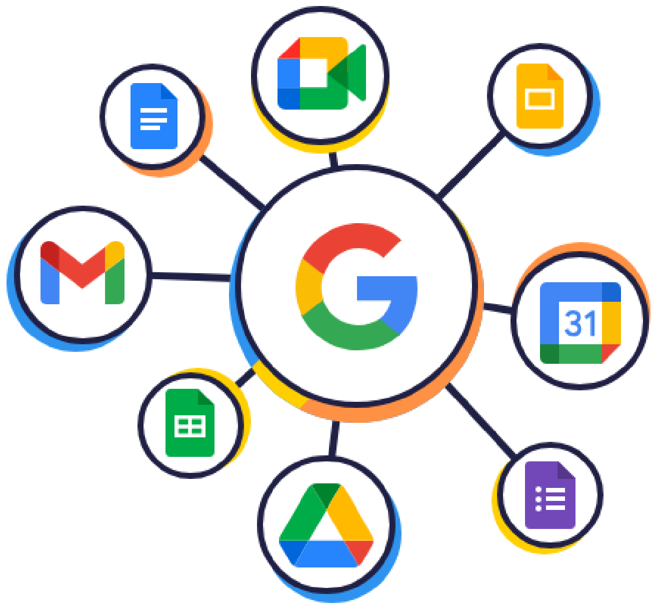
Implementation

TutorialsPoint's website proved to be very useful in our attempt to complete this project. It has clear descriptions of different design patterns as you can see. It provides implementations of the design patterns and step by step tutorials on how to go about actually implementing the design pattern correctly.

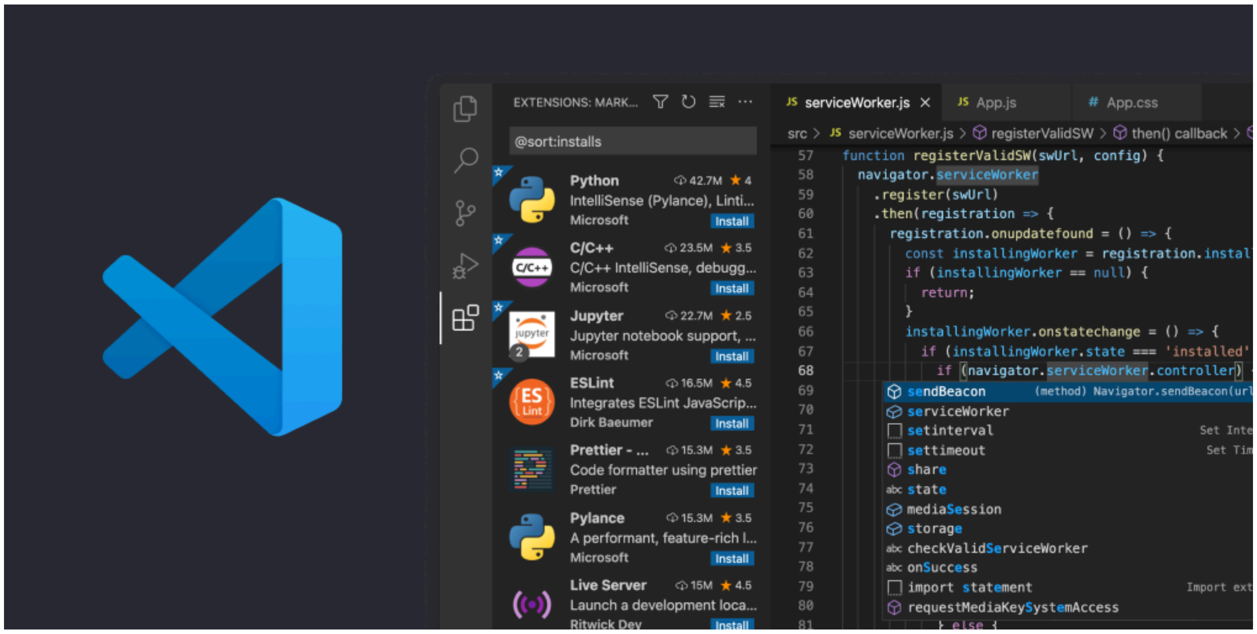
Stack overflow was also very helpful with completing this project as when we ran into many specific error messages that we had a hard time resolving. To fix a lot of these errors we found different forums on stackoverflow that presented us with similar errors that were solved or had possible solutions. Stack was a good resource that gave us help outside the classroom with specific issues we ran into.

Discord was our main tool used for communication for this project. All of our discussions as well as group meetings where we would call happened through discord. We planned many things here as well as communicated with each other throughout the project here.

We got together on calls and worked on the project a couple of times just so we all were on the same page with things.



Of course we used google and their plethora of resources in order to create our presentation as well as this document you are currently reading from. It proved to be very useful to get our slides together and was helpful especially because we were collaborating on it together.



```
function registerValidSW(swUrl, config) {
  navigator.serviceWorker
    .register(swUrl)
    .then(registration => {
      registration.onupdatefound = () => {
        const installingWorker = registration.instal
        if (installingWorker == null) {
          return;
        }
        installingWorker.onstatechange = () => {
          if (installingWorker.state === 'installed'
            if (navigator.serviceWorker.controller)
              sendBeacon          (method) Navigator.sendBeacon(url
              serviceWorker
              setInterval                           Set Inte
              setTimeout                            Set Tim
              share
              state
              mediaSession
              storage
              checkValidServiceWorker
              onSuccess
              import statement                      Import ext
              requestMediaKeySystemAccess
            } else {
```

Another resource we used and one of the most important we used was visual studio code. This was our IDE that we decided to use for this project and what we programmed on. As this project was from a previous year Jason uploaded the project to github. We cloned the repository he made and all worked on VScode to change the code up in order to implement the new design patterns we intended on.

Lessons Learned:

Dheeraj’s Lessons Learned:

As far as lessons learned goes I think through this project I learned a great deal about software development as a whole. I think this was a small taste of what we will be doing in the capstone project so it gave me a nice learning experience for that. I also learned that UML diagrams are a great help when it comes to creating or adding to an entire system. Especially when implementing new design patterns into an existing system, it really allows you to see how things work together and where new design patterns would fit in. I also learned that communication with teammates is key to getting things done on time and without stress. Luckily this time my team was very communicative and helpful throughout the project making things run very smoothly. Finally one of the last things I learned is that design patterns make things much more readable and understandable for someone that is not familiar with the system itself.

Jason’s Lessons Learned:

Throughout the development of this project, the most important lesson I have learned is that writing quality, maintainable code in the past will save you a lot of effort in the future. I had worked on this program last semester, and while it was just as functional as it is now, it was a mess on the inside. There were shortcuts and messy logic and all sorts of common errors I would cringe at if I did them today. After incorporating the design patterns, however, the code is much more clean and readable. This taught me that if you make your code lean and organized the moment you write it, it becomes much more accessible in the future, and to others who aren't familiar with the system. Learning the design patterns in class is good, but actually implementing them made me understand how useful they can be. I learned that these patterns aren't just theoretical luxuries, but are super useful and can be applied to improve the quality of code.

JJ's Lessons Learned

When it comes to creating an application from scratch or improving an already existing one, it's always good to be able to have a map of what needs to be done, how you're going to do it, why some structures are better than others, and how to even assess what is potentially the best solution to different problems. What I learned from this course is exactly that, but at the beginning level I feel. Being able to go through many hypothetical engineering scenarios and being able to analyze what the problem is and how to potentially solve them was the biggest thing for me. While learning each design pattern, their uses, their advantages, disadvantages, and why some may be better than others depending on the situation at hand will always be necessary when trying to achieve effectiveness and maintaining the SOLID principles. Creating clearer, maintainable, flexible, and testable code while having proper structures to them will always improve the quality of any application if done properly. I learned what it takes to start thinking like a software engineer and be able to apply it to how I approach problems for the betterment of not only myself, but to my peers who I may work with and ask questions, but also for the betterment of whatever application I may work on.