# Welcome to Rohith CMS!

# Running Instructions:

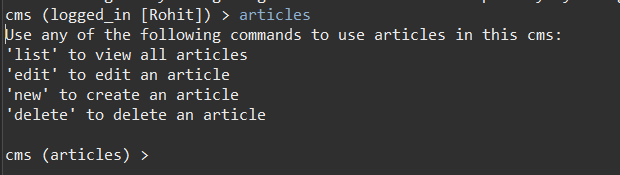
1.) After running the project, you should see the welcome banner as shown below



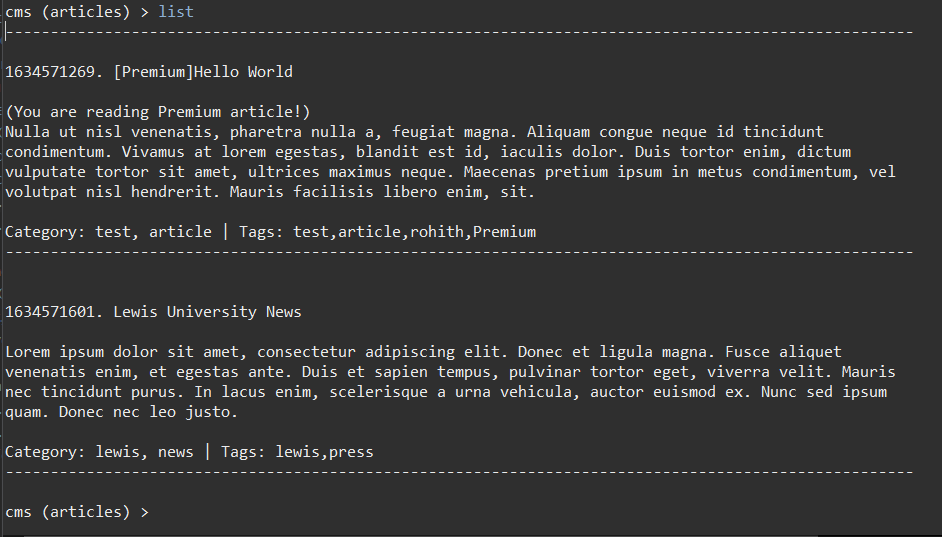
2.) Use the *login* command to enter into the CMS.



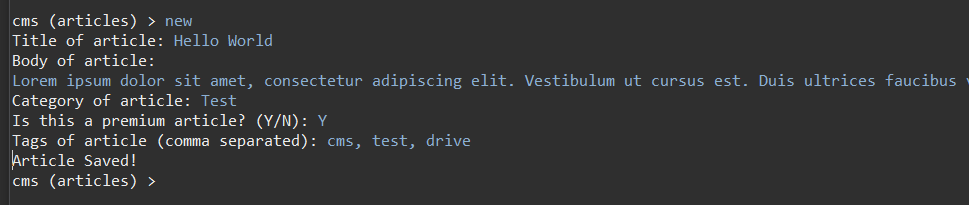
3.) Use the *articles* command to enter into articles location in CMS. This command should show you the commands for 4 crud operations.



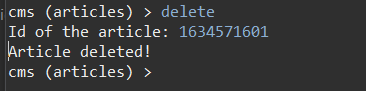
4.) Use the *list* command to show the list of articles available in CMS.



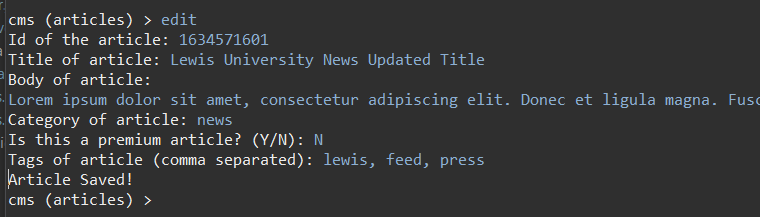
5.) Use the *new* command to create new article



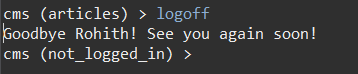
6.) Use the *delete* command to delete existing article



7.) Use the *edit* command to edit/update an existing article



8.) Use the *logoff* command to sign out of the current user “session”.



9.) Use the *exit* command to quit CMS.

enter image description here

# Project Packages:

The project contains 5 packages as listed below:

* *com.rohit.cms.articles* - contains articles related logic
* *com.rohit.cms.main* - contains entry point and classes that allow cms to run
* *com.rohit.cms.models* - contains pojo classes and interfaces
* *com.rohit.cms.tests* - contains unit tests for the project
* *com.rohit.cms.user* - contains user related logic like login/logoff commands and UsersService

# Design Patterns & Corresponding Files:

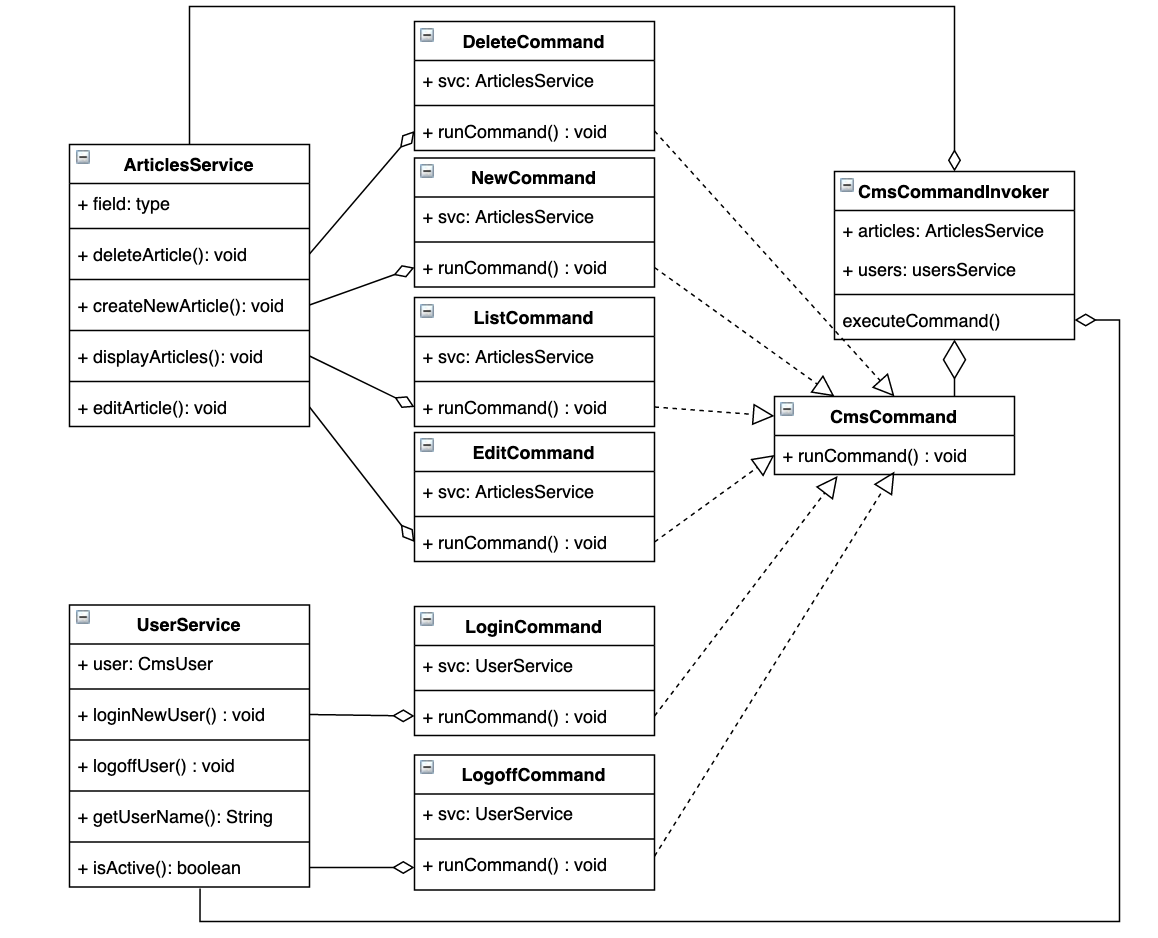
There are 4 design patterns in the code base. The following section describes the patterns, related files and UML diagram of the pattern.

## Command pattern

### Files & Packages:

| File | Package |
| --- | --- |
| CmsCommand.java | com.rohit.cms.models |
| CmsCommandInvoker.java | com.rohit.cms.main |
| EditArticleCommand.java | com.rohit.cms.articles |
| DeleteArticleCommand.java | com.rohit.cms.articles |
| ListArticlesCommand.java | com.rohit.cms.articles |
| NewArticleCommand.java | com.rohit.cms.articles |
| ArticlesService.java | com.rohit.cms.articles |
| LoginCommand.java | com.rohit.cms.user |
| LogoffCommand.java | com.rohit.cms.user |
| UserService.java | com.rohit.cms.user |

### UML Diagram:

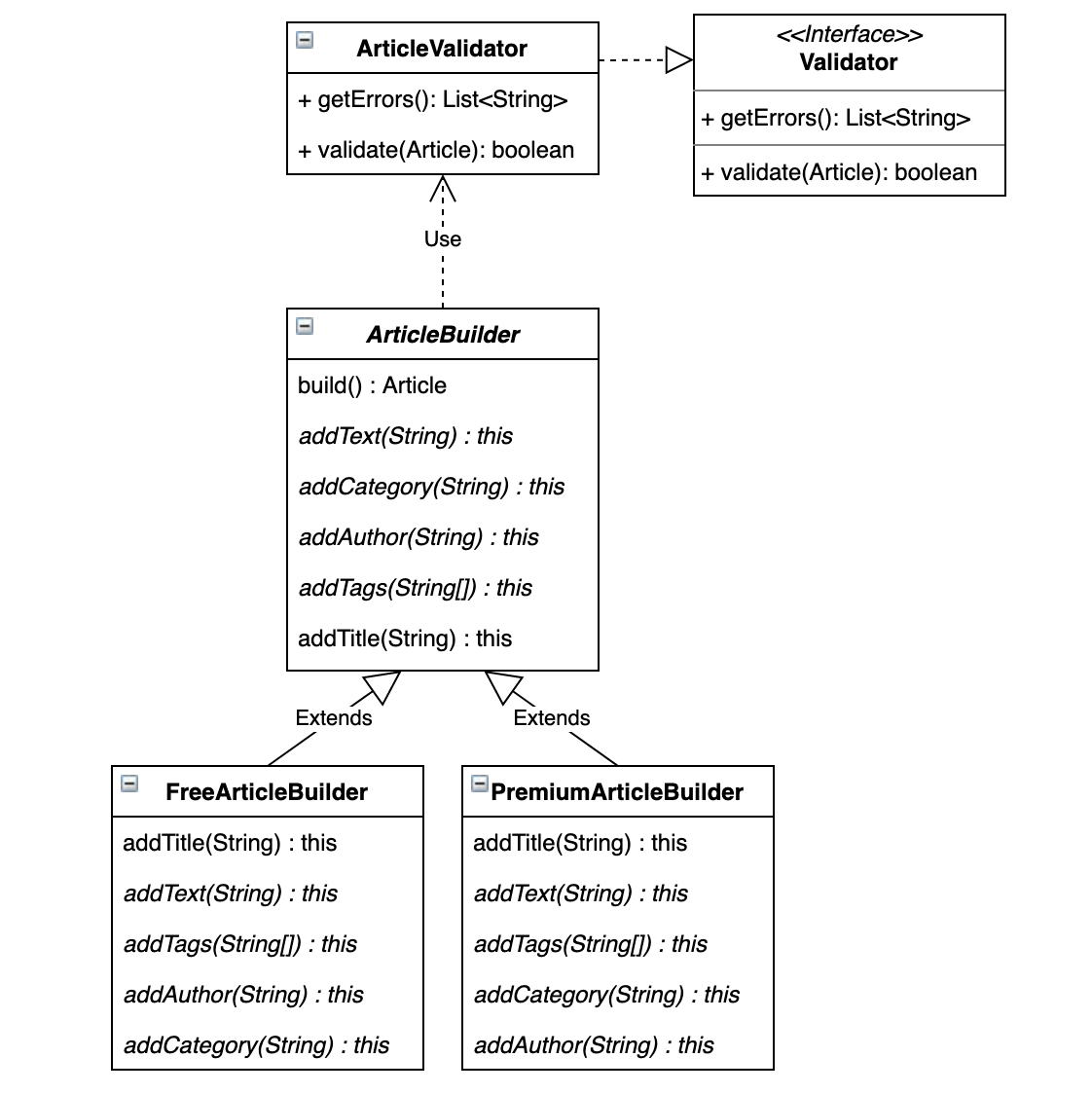


## Builder Pattern

### Files & Packages:

| File | Package |
| --- | --- |
| ArticleBuilder.java | com.rohit.cms.articles |
| FreeArticleBuilder.java | com.rohit.cms.articles |
| PremiumArticleBuilder.java | com.rohit.cms.articles |

### UML Diagram:

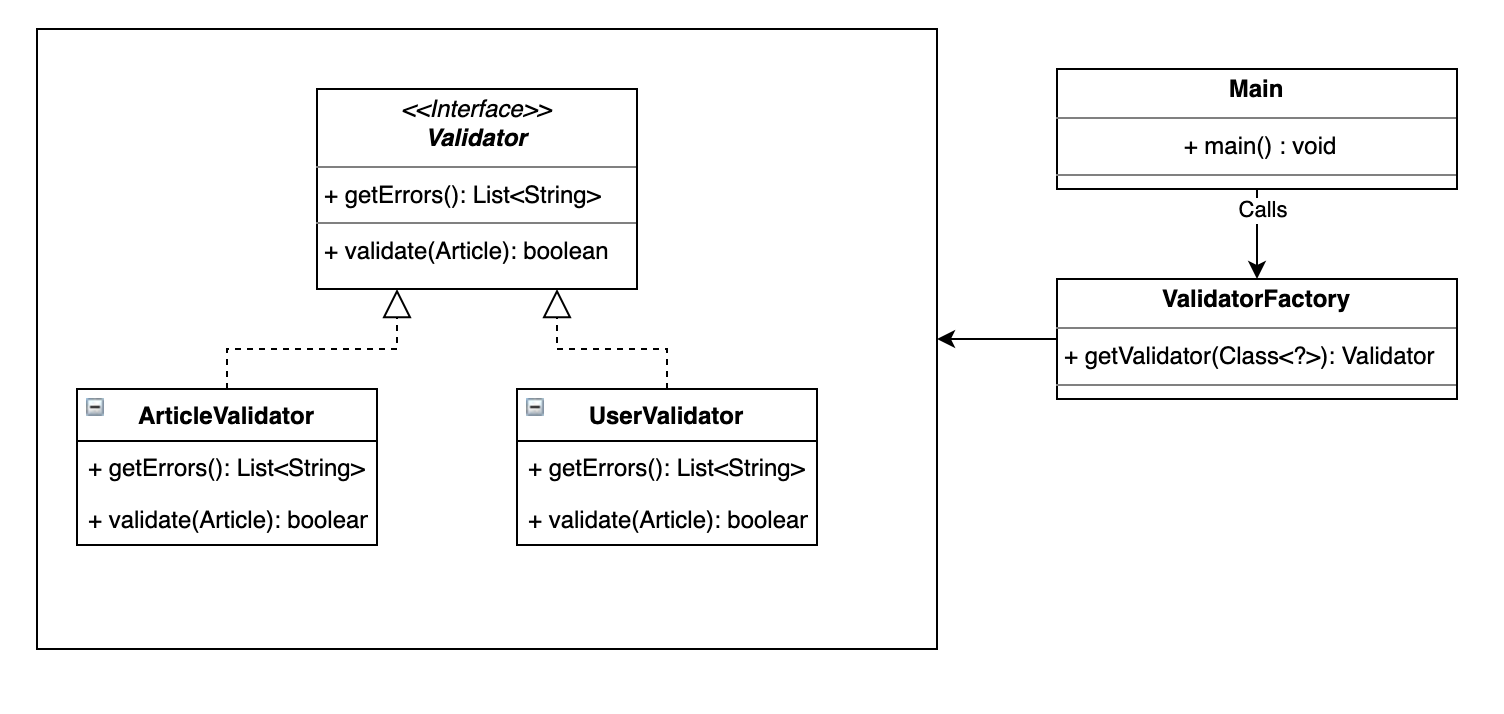


## Factory Pattern

### Files & Packages:

| File | Package |
| --- | --- |
| Validator.java | com.rohit.cms.models |
| ArticleValidator.java | com.rohit.cms.articles |
| UserValidator.java | com.rohit.cms.user |
| ValidatorFactory.java | com.rohit.cms.main |

### UML Diagram:

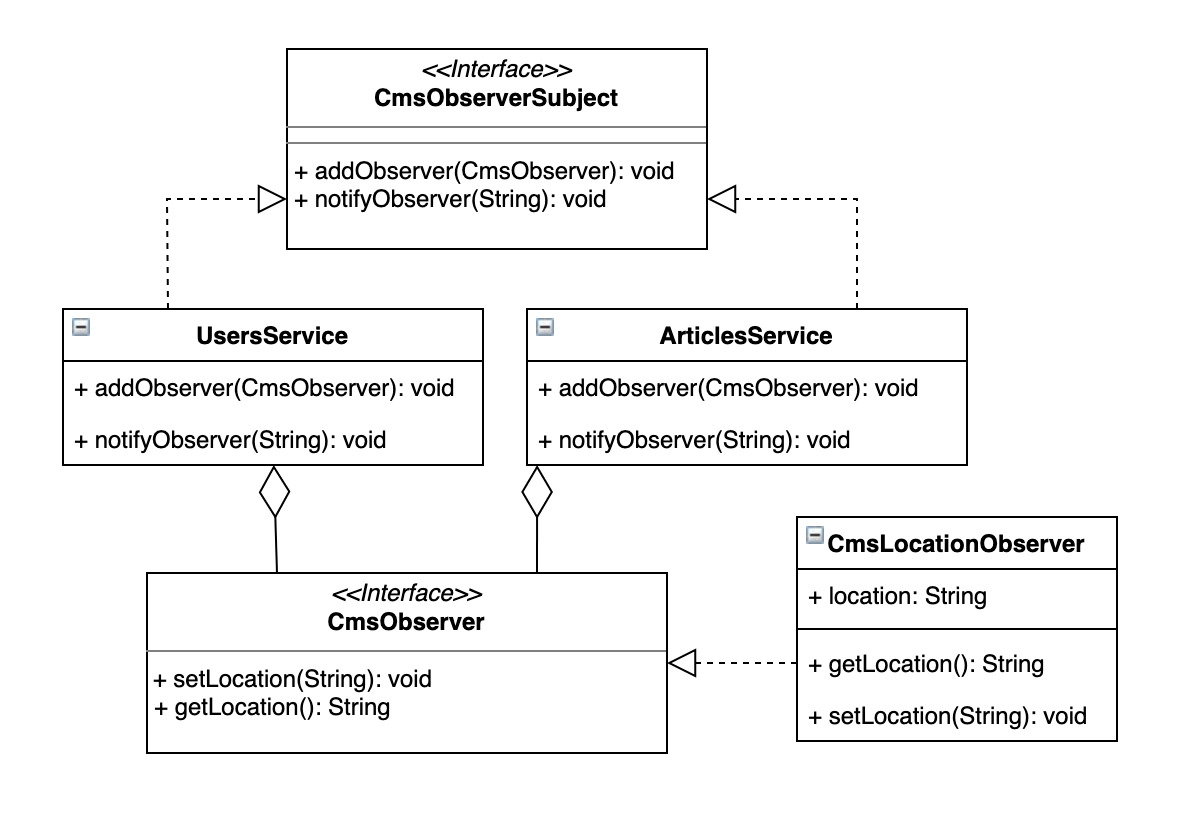


## Observer Pattern

### Files & Packages:

| File | Package |
| --- | --- |
| CmsObserverSubject.java | com.rohit.cms.models |
| CmsObserver.java | com.rohit.cms.models |
| CmsLocationObserver.java | com.rohit.cms.models |
| UserService.java | com.rohit.cms.user |
| ArticlesService.java | com.rohit.cms.articles |

### UML Diagram:



# Write Up:

I wanted to build a CMS cli that can

1. (Simulate) Login/logoff of a user
2. Do CRUD operations on articles

Since it’s a CLI, I needed a way to take commands from the user and trigger corresponding functionality.

I chose the command pattern because it allows me to trigger functionality on services while decoupling execution between the cli input and classes that implement them.

I felt the commands were a thin layer with almost no functionality in them and wondered if they were useful at all. But it came very handy in the CmsCommandInvoker because the execution has consistent API for all the cli commands (runCommand), and I can swap out the implementation of the command while keeping the invoker unchanged.

I wanted to implement validation functionality since there is some amount of user input in the login and articles section. I needed a consistent API for both user and articles validator. Factory pattern was a good fit because it allowed me to return the instance of the validator I needed if I passed object class information to factory class. If I need to extend it with another validator, I can just implement another validator and update the factory class to return the correct validator for the respective object type.

Being a CLI, there needed to be a way of tracking where the user is during the use and only allow some commands when the user is in a certain location. I also had to retrieve the current location to display it in the CLI. To accomplish this, I used Observer pattern where the services “notify” the registered observer on the current location. I had difficulty implementing this pattern as unlike the examples where there is a one-to-many relationship between subject and observer, I had 2 subject classes which can mutate the location and one class to ‘observe’ the state. There is also some tight coupling between the observer and the subject class which is also happens to be the receiver service class implemented in command pattern.

When the user wants to create an article, they can select premium or non-premium articles. Since the article object has custom logic if it's premium or not, I needed a way to separate the two ways to build the respective classes. Builder suited this well since I can create two different types of objects of the same class based on a condition like isPremium.

The challenge I faced is to figure out which pattern would suit me best for the problem since I saw some similarities between Factory and Command pattern when I was trying to implement validation for user and article objects (using UserValidationCommand and ArticleValidationCommand which invoke UserValidationService and ArticleValidationService respectively). Then I realised, it's best to see which API makes sense from a consumer class and which pattern would best fit for the potential changes in the future.

I chose Factory pattern for the validation since there will be a single validate method and that API will almost never change in contrast to the articles service which can have more features in the future and still needs to have consistent API in the invoker. Command pattern suited this better. I think with more practice I can get a hang of which pattern to use when.