Dream Catch

Student: Vlad Ursache

**Group: 30238-2**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 3

3. System Architectural Design 4

4. UML Sequence Diagrams 9

5. Class Design 9

6. Data Model 12

7. System Testing 12

8. Bibliography 13

1. Requirements Analysis

# Assignment Specification

Dream Catch Sleep Tracker is a web application designed to help users monitor their sleep patterns and gain insights into their sleep quality. The target audience for the app is individuals who are interested in improving their sleep habits or managing sleep-related conditions. The app must be able to handle a high volume of user data and provide accurate and timely feedback to users.

# Functional Requirements

* Register for an account or log in if account already present
* Submit sleep data, including description, duration, energy, stress levels and custom tags
* Plot different aspects of sleep in the form of area charts.
* Filter sleep data based on specific criteria, such as duration, energy and stress levels

# Non-functional Requirements

* Performance: The app must be able to handle a high volume of user data and provide accurate and timely feedback to users.
* Scalability: The app must be able to scale to accommodate increasing user loads without significant degradation in performance.
* Security: The app must ensure the confidentiality, integrity, and availability of user data and comply with relevant regulations and standards such as GDPR and HIPAA.
* Usability: The app must be user-friendly and intuitive to facilitate ease of use and engagement.
* Reliability: The app must be highly available and reliable, with minimal downtime or service interruptions.
* Maintainability: The app must be easy to maintain and support over time, with clear documentation and well-organized code.
* Compatibility: The app must be compatible with a wide range of web browsers, operating systems, and devices.
* Portability: The app must be designed to run on a variety of platforms, such as desktop or mobile devices.

2. Use-Case Model

***Use case: Register***

***Level: User-goal level***

***Primary actor: User***

***Main success scenario:***

* ***User navigates to the registration page.***
* ***User enters their personal information and creates a username and password.***
* ***User clicks on the "Register" button.***
* ***System validates the user's information and creates a new account for the user.***
* ***System redirects the user to the login page.***

***Extensions:***

* ***If the user enters invalid information, the system will not redirect the user to the login page.***

***Use case: Login***

***Level: User-goal level***

***Primary actor: User***

***Main success scenario:***

* ***User navigates to the login page.***
* ***User enters their username and password.***
* ***User clicks on the "Login" button.***
* ***System validates the user's credentials and grants access to the system.***
* ***System redirects the user to the home page.***

***Extensions:***

* ***If the user enters an incorrect username or password, the system will not redirect the user to the main page.***

***Use case: Submit dream***

***Level: User-goal level***

***Primary actor: User***

***Main success scenario:***

* ***User enters the required data.***
* ***User clicks on the "Submit" button.***
* ***System validates the data and saves it to the database.***
* ***Data is updated within the UI***

***Use case: Visualize data***

***Level: User-goal level***

***Primary actor: User***

***Main success scenario:***

* ***User selects the desired parameters to visualize the data.***
* ***System retrieves the data from the database and generates the visualizations.***
* ***System displays the visualizations to the user.***

3. System Architectural Design

**3.1 Architectural Pattern Description**

The Client-Server architecture stands on the layered one. The Layered Architectural Pattern is a common design pattern used in software development that separates an application into distinct layers. Each layer is responsible for a specific set of tasks and communicates only with the layers directly above and below it. The pattern provides a clear separation of concerns and makes the application more modular and easier to maintain.

In this pattern, the application is divided into four layers: the Presentation layer, the Business layer, the Data Access layer, and the Data Storage layer. The layers are arranged in a hierarchical order, with the Presentation layer being the topmost layer and the Data Storage layer being the bottommost layer.

The following is a description of each layer and its associated packet:

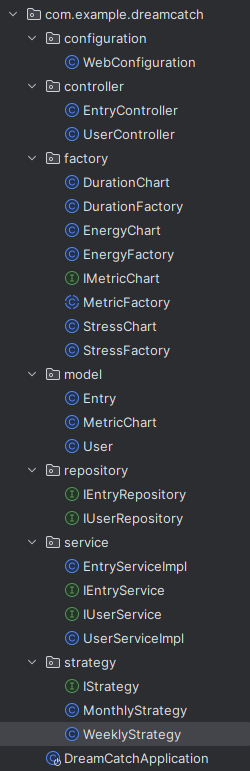
* Presentation layer (Controller): This layer is responsible for handling user input and displaying output to the user. The Controller receives user input from the client and sends it to the appropriate Service layer for processing. It also receives data from the Service layer and formats it for display to the user.
* Business layer (Service): This layer contains the application logic and is responsible for processing the data. It receives data from the Presentation layer, performs business operations on the data, and sends the data to the Data Access layer for persistence.
* Data Access layer (Repository): This layer is responsible for retrieving and persisting data to a data store. It receives data from the Service layer and interacts with the Data Storage layer to store or retrieve the data.
* Data Storage layer (Model): This layer represents the data store where the data is stored. It interacts with the Data Access layer to store and retrieve data from the data store.

In addition to these four layers, a Factory packet can be used to create objects and inject dependencies between the layers. The Factory packet is responsible for creating and managing the objects used by the application and can be used to implement the Dependency Injection design pattern.

Overall, the Layered Architectural Pattern provides a modular and scalable approach to designing software applications. By separating the application into distinct layers, it becomes easier to modify and maintain the code, and different teams can work on different layers independently without affecting other layers of the application.

Another layer, the Strategy layer is responsible for customizing behavior. The general idea of the factory method design pattern is preserved, except for the fact that no object is being created; instead behavior is defined.

**3.2 Diagrams**

****

The diagram above shows the main layers and components of the system's architecture. The system is organized into five main layers: presentation, service, data access, domain, and factory.

The presentation layer is responsible for handling user interactions and displaying data to the user. It is implemented using React and communicates with the service layer to retrieve and modify data.

The service layer contains the application's business logic and coordinates the flow of data between the presentation and data access layers. It is implemented using SpringBoot and includes the EntryServiceImpl, IEntryService, and IUserService classes.

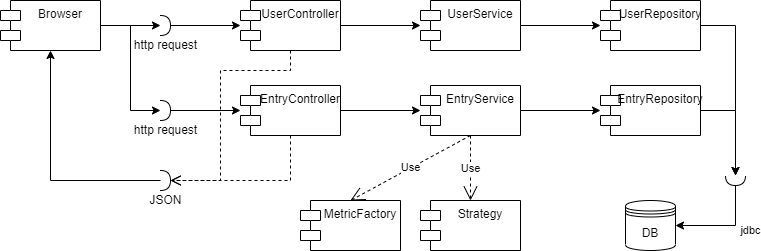
The data access layer is responsible for accessing the database and persisting data. It is implemented using Spring Data JPA and includes the IEntryRepository and IUserRepository classes.

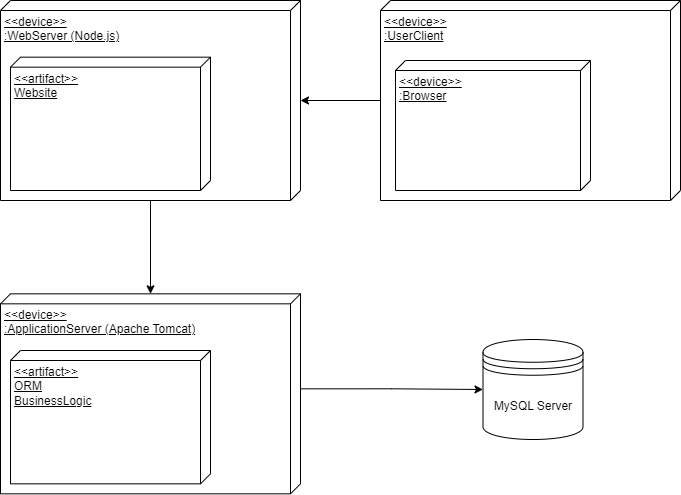
The domain layer contains the application's domain objects and business logic. It includes the Entry, User, MetricChart, and Tag classes.

The factory layer contains the metric factories responsible for creating metric charts. It includes the DurationFactory, EnergyFactory, MetricFactory, and StressFactory classes.

Overall, the system's architecture follows a layered architecture pattern and uses the factory method pattern for creating metric charts. The Strategy design pattern customizes bahavior for weekly and monthly data extraction.

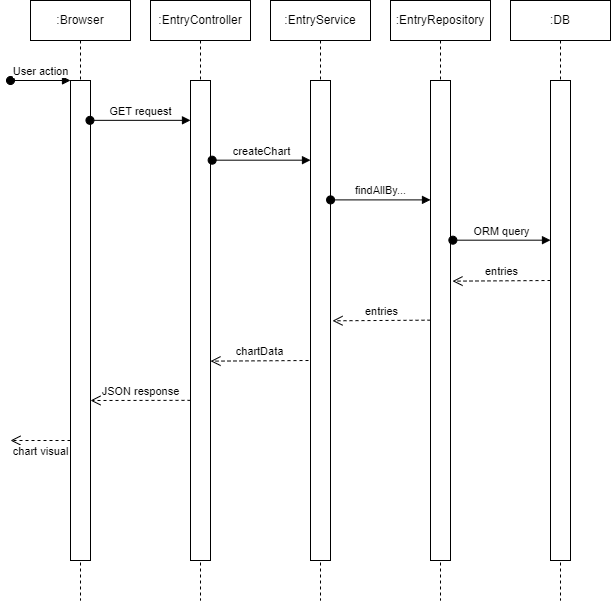
Down below are the component and deployment diagrams (in that order).

****

****

4. UML Sequence Diagrams

Down below is the sequence diagram for the creation of MetricCharts.

**

5. Class Design

**5.1 Design Patterns Description**

The Factory Method Design Pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

In this pattern, a factory method is defined in an abstract superclass or interface, which declares a method that returns an object of a specified class or interface. Subclasses of the superclass or interface implement this factory method to create objects of different concrete classes that implement the same interface.

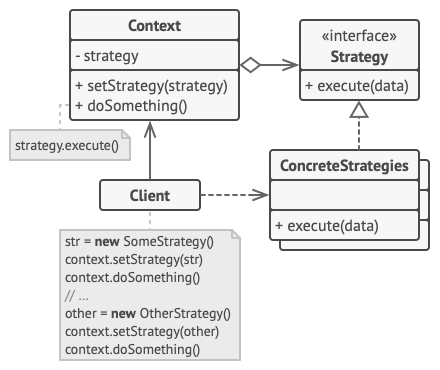
This pattern provides a way to encapsulate object creation in a separate method or class, and it also allows for flexibility in changing the class of objects that are created without affecting the code that uses those objects.



The Factory Method Design Pattern is useful in situations where we need to create different objects based on certain conditions or parameters, and where we want to avoid tight coupling between the code that creates objects and the code that uses them. This pattern is often used in frameworks where the exact classes of objects needed are not known at compile-time, but are instead determined at runtime.

Similar to the Factory DP is the Strategy one. The Strategy design pattern is useful because it allows for the separation of concerns in a program. It enables you to define a family of algorithms, encapsulate each one as an object, and make them interchangeable at runtime.

By doing this, you can choose which algorithm to use at runtime, depending on the situation or conditions. This makes your code more flexible, reusable, and maintainable, as you can easily switch between different algorithms without having to change the code that uses them.

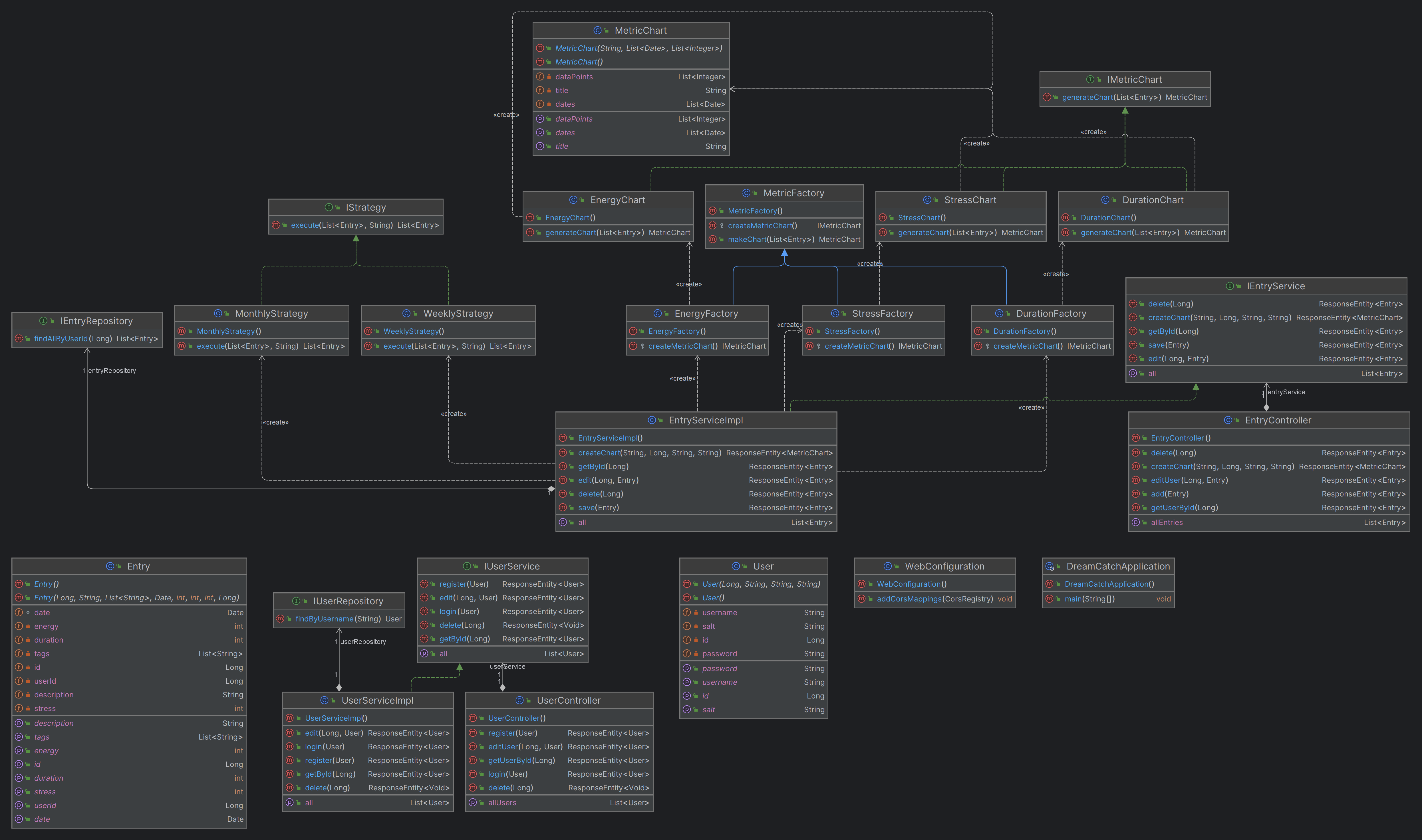


**5.2 UML Class Diagram**

To create different types of charts for different metrics, you can implement concrete Factory classes that implement the IMetricChart interface. For example, I created a DurationFactory class that generates charts for duration metrics, an EnergyFactory class that generates charts for energy metrics, and a StressFactory class that generates charts for stress metrics. Each Factory class would implement the createChart() method specified in the IMetricChart interface, and would create a concrete Chart object that is specific to the corresponding metric.

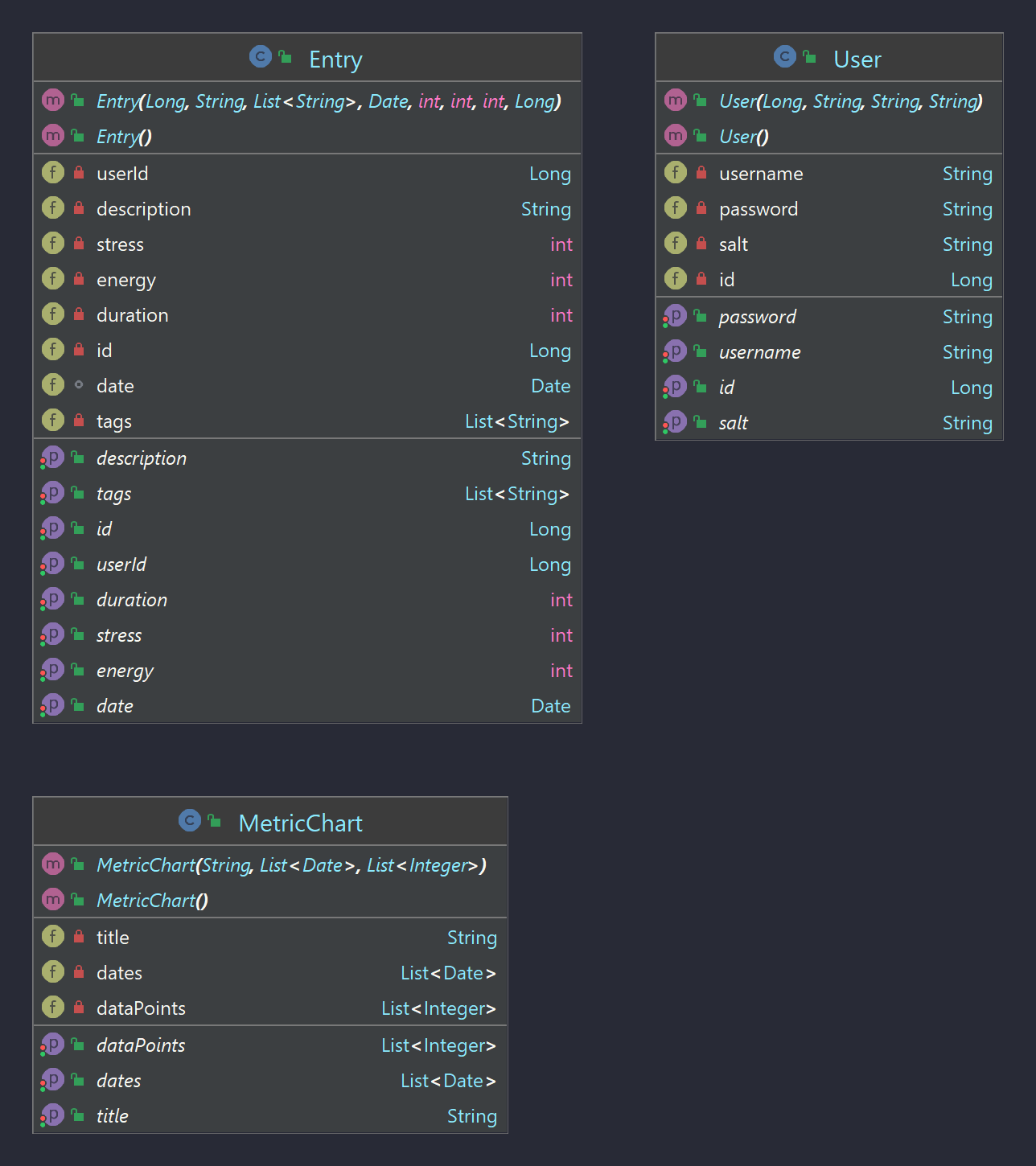
Using this approach, I can easily add new Factory classes to generate charts for new metrics, without having to modify the existing code. You can also create different types of charts for the same metric by creating additional Factory classes that implement the IMetricChart interface in different ways.

Overall, this approach allows for greater flexibility and maintainability, as it separates the chart creation process from the chart rendering process, and allows for easy customization and extension. The detailed UML class diagram is below:



6. Data Model

A UML diagram speaks more than a thousand words, therefore here is my model (domain layer):



7. System Testing

Unit testing was done with JUnit and dependencies were mocked with Mockito. The entire User and Entry API have been successfully tested.  
Some level of integration testing has been done and everything seems to be working fine.

Graphical user interface, text, application

Description automatically generated

8. Bibliography

<https://refactoring.guru/design-patterns/factory-method>

<https://refactoring.guru/design-patterns/strategy>