# ENGENHARIA DE SOFTWARE

## Relatório

**Alunos**
Gustavo Sanches Costa
Lucas Bryan Treuke
Rodrigo Gomes Hutz Pintucci
Tiago Barradas Figueiredo
Vanessa Berwanger Wille

# Sumário

# 1 Introdução

Esse relatório tem o propósito de introduzir e explicar o funcionamento por trás de nosso aplicativo de calendário em grupo, cujas características principais são a capacidade de realizar uma compatibilidade de agendas e agendamento de eventos entre pessoas de fusos horários distintos.

O processo de testes das funções do aplicativo foram documentadas e registradas em arquivos de *jupyter notebook* presentes dentro das pastas de testes de cada classe.

Nosso grupo focou a maior parte de seu tempo em garantir que o processo estivesse saíndo de forma fluída e consisa. Foram feitos commits na plataforma GitHub para cada etapa RED-GREEN-REFACTOR, além da utilização de branches para facilitar a revisão de código, e também levantamento de issues com eventuais problemas encontrados.

Nosso grupo focou a maior parte de seu tempo em garantir que o processo estive saíndo de forma fluída e consisa. Foram feitos commits na plataforma GitHub para cada etapa RED-GREEN-REFACTOR, além da utilização de branches para facilitar a revisão de código, e também levantamento de issues com eventuais problemas encontrados.

# 2 Arquitetura

A arquitetura do projeto se baseia, principalmente, em três classes distintas, que modelam os objetos mais importantes do aplicativo: Element, Schedule, e User.

Cada objeto da classe Element representa um elemento que pode estar presente em alguma agenda, podendo ser um evento, uma tarefa, ou um lembrete, contendo informações como sua data de início e fim, uma lista de agendas a qual o elemento pertence, e uma descrição e título. A classe ElementFactory é responsável por criar objetos Element, facilitando suas criações ao interpretar de forma distinta os parâmetros de entrada para cada um dos tipos de elemento.

Os objetos da classe Schedule representam as agendas que os usuários podem criar, e armazenam dados como o nível de permissão de cada usuário presente na agenda, uma lista de quais elementos foram adicionados nela, e sua descrição e título.

A classe User, por sua vez, representa os usuários do sistema. Ela contém informações como os dados de login, as preferências do usuário (como seu fuso horário), e a quais agendas o usuário pertence.

Cada uma dessas três classes é gerenciada pelas classes ElementManagement, ScheduleManagement, e UserManagement. Elas têm o papel de serem uma interface que armazena localmente todos os objetos dessas classes que foram lidos do banco para que seja possível a realização de operações em cima deles. Essas três classes interagem com o banco a partir da classe MongoModule, que gerencia a interação e conexão com o banco MongoDB. Esse banco, por sua vez, pode ser hosteado localmente através de uma imagem docker, ou pode ocorrer através de um acesso remoto a uma imagem docker hosteada em uma instância EC2 da AWS, que é a configuração padrão do projeto.

Vamos passar por cada classe, seus atributos e seus métodos:

## 2.1 Banco de Dados

### 2.1.1 DatabaseModule

A classe abstrata *DatabaseModule* serve como interface para módulos de banco de dados que iremos utilizar. Possui os seguinte métodos:

- **connect:** Conecta ao banco de dados;
- **disconnect:** Desconecta do banco de dados;
- **insert_data:** Insere informações no banco de dados;
- **update_data:** Atualiza informações no banco de dados;
- **delete_data:** Deleta informações no banco de dados;
- **select_data:** Retorna informações desejadads do banco de dados.

### 2.1.2 MongoModule

A classe *MongoModule* é uma realização da classe abstrata *DatabaseModule* e é específica para o *MongoDB*. Possui os seguintes atributos:

- **_instance:** Instância da classe, usada para implementar o *Singleton*;
- **_host:** Endereço *host* do banco de dados
- **_port:** Porta do banco de dados;
- **_database_name:** Nome do banco de dados;
- **_user:** Usuário do banco de dados;
- **_password:** Senha do banco de dados;
- **_client:** *MongoClient* do banco de dados;
- **_db:** Banco de dados;
- **collection:** Collection do banco de dados. Nossas collections são "users", "schedules"e "elements".

Além dos métodos herdados, possui um método **__new__** e um método *get_instance* para garantir a instância *Singleton*.

## 2.2 Usuário

### 2.2.1 User

A classe *User* é um *Subject* que representa um usuário da aplicação. Ele possui os seguintes atributos:

- **__id:** ID único do usuário;
- **username:** Nome de usuário;

- **email:** Email do usuário;
- **__schedules:** Lista de IDs de agendas do usuário;
- **__hashed_password:** Senha criptografada do usuário;
- **__user_preferences:** Dicionário de preferência dos usuários, tal como fuso horário.
- **__observers:** Lista de observers.

Os atributos privados *__id*, *__schedules* e *__observers* possuem métodos *getters* e *__schedules* possui método *setter*. Ademais, temos os seguintes métodos:

- **get_schedules:** Retorna uma lista de objetos *Schedules* referentes ao usuário. Vale notar que isso é diferente do atributo *schedules*, que só possui seus *IDs*;
- **get_elements:** Retorna uma lista de objetos *Elements* referentes ao usuário. Para isso, passa primeiro por cada *Schedule*. Os *Schedules* a serem considerados podem ser passados através de uma lista com seus *IDs*;
- **get_hashed_password:** Retorna o atributo *__hashed_password*;
- **set_username:** Altera o *username* do usuário. Isso notificará seu *Observer*, *UserManagement*;
- **set_email:** Altera o *email* do usuário. Isso notificará seu *Observer*, *UserManagement*;
- **set_preferences:** Altera a *user_preferences* do usuário. Isso notificará seu *Observer*, *UserManagement*;
- **check_disponibility:** Verifica a disponibilidade do usuário dado um intervalo de tempo, retornando um bool;
- **to_dict:** Transforma a instância em um dicionário compatível com o banco de dados;
- **attach:** Se conecta a um *Observer*;
- **detach:** Se desconecta de um *Observer*;
- **notify:** Notifica todos os *Observers*;

#### 2.2.2 UserManagement

A classe *UserManagement* é um *Observer* que manuseia as instâncias de usuário. Ele possui os seguintes atributos:

- **_instance:** Instância da classe, usada para implementar o *Singleton*;
- **db_module:** Módulo de banco de dados da classe. Utilizamos a classe *MongoModule*;
- **users:** Dicionário que segura as instâncias ativas de usuário. As chaves são os IDs e os valores as instâncias.

Temos os seguintes métodos:

- **get_instance:** *@classmethod* que retorna a instância da classe, utilizada para garantir que nos referenciamos sempre a mesma instância;
- **create_user:** Caso não exista no banco, cria um usuário novo com as informações passads e o adiciona no banco. Uma instância é criada e introduzida no dicionário *users*;
- **update_user:** Atualiza o banco de acordo com os valores de uma instância *User* dessincronizada através do seu *ID*;
- **delete_user:** Através de seu *ID*, deleta um usuário do banco e do dicionário *users*. *Schedules* relacionadas também são atualizadas;
- **get_user:** Retorna uma instância de usuário através de seu *ID*. Caso o usuário não esteja no dicionário *users* mas exista no banco, um objeto é criado para ele;
- **user_exists:** Verifica a existência de um usuário no banco através de seu *ID*;
- **hash_password:** Criptografa a senha de um usuário;
- **add_schedule_to_user:** Adiciona uma schedule válida a um usuário através de suas *IDs*, com ambos sendo atualizados;
- **update:** Método de *Observer*, é chamada quando uma instância de usuário é modificada.

### 2.2.3 AuthenticationModule

A classe *AuthenticationModule* é utilizada para verificar o *login* de usuários. Possui um atributo:

- **user_management_module:** instância de *UserManagement*.

A classe possui dois métodos:

- **verify_password:** Recebe a senha introduzida e checa se bate com a senha criptografada do usuário.
- **authenticate_user:** Recebe um usuário e uma senha e verifica se as informações de *login* estão corretas.

## 2.3 Agenda

### 2.3.1 Schedule

A classe *Schedule* é um *Subject* que representa um usuário da aplicação. Ele possui os seguintes atributos:

- **__id:** ID único da agenda;
- **title:** Título da agenda;
- **description:** Descrição da agenda;

- **__permissions** Dicionário onde as chaves são os usuários da agenda e os valores são suas permissões.
- **__elements:** Lista de IDs dos *elements* presentes na agenda;
- **__observers:** Lista de observers.

Os atributos privados *__id*, *__permissions*, *__elements* e *__observers* possuem métodos *getters*. Temos também métodos *setters* para *permissions* e *elements*. Ademais, temos os seguintes métodos:

- **get_elements:** Retorna uma lista de objetos *Elements* inclusas na agenda, sendo possível filtrar pelo tipo de *Element*.
- **get_users:** Retorna uma lista de objetos *Users* da agenda, sendo possível filtrar pelo tipo de permissão.
- **set_title:** Altera o *title* da agenda. Isso notificará seu *Observer*, *ScheduleManagement*;
- **set_description:** Altera a *description* da agenda. Isso notificará seu *Observer*, *ScheduleManagement*;
- **to_dict:** Transforma a instância em um dicionário compatível com o banco de dados;
- **attach:** Se conecta a um *Observer*;
- **detach:** Se desconecta de um *Observer*;
- **notify:** Notifica todos os *Observers*;

### 2.3.2 ScheduleManagement

A classe *ScheduleManagement* é um *Observer* que manuseia as instâncias de agenda. Ele possui os seguintes atributos:

- **_instance:** Instância da classe, usada para implementar o *Singleton*;
- **db_module:** Módulo de banco de dados da classe. Utilizamos a classe *MongoModule*;
- **schedules:** Dicionário que segura as instâncias ativas de agenda. As chaves são os IDs e os valores as instâncias.

Temos os seguintes métodos:

- **get_instance:** *@classmethod* que retorna a instância da classe, utilizada para garantir que nos referenciamos sempre a mesma instância;
- **create_schedule:** Caso não exista no banco, cria uma agenda nova com as informações passadas e a adiciona no banco. Uma instância é criada e introduzida no dicionário *schedules*;
- **update_schedule:** Atualiza o banco de acordo com os valores de uma instância *Schedule* dessincronizada através do seu *ID*;

- **delete_schedule:** Através de seu *ID*, deleta uma agenda do banco e do dicionário *schedules*. *Users* e *Elements* relacionados também são atualizados;
- **get_schedule:** Retorna uma instância de agenda através de seu *ID*. Caso o usuário não esteja no dicionário *schedules* mas exista no banco, um objeto é criado para ele;
- **schedule_exists:** Verifica a existência de uma agenda no banco através de seu *ID*;
- **add_element_to_schedule:** Adiciona um elemento válido a uma agenda através de suas *IDs*, com ambos sendo atualizados;
- **update:** Método de *Observer*, é chamada quando uma instância de usuário é modificada.

## 2.4 Elementos do Calendário

### 2.4.1 Element

A classe abstrata *Element* é um *Subject* que servirá de interface para todos os tipos de elemento que podem estar presentes numa agenda: *EventElement*, *TaskElement* e *ReminderElement*. Seu único atributo é **__observers:**, a lista de observers, para que não seja necessário definir isto em suas realizações.

Vamos comentar dos métodos que serão comuns para cada tipo de elemento:

- **get_display_interval:** Retorna uma tupla com o intervalo que o elemento ocupará durante a aplicação. No caso do *EventElement*, este intervalo será sua data de início e sua data de término. Nos outros dois casos, onde só é passado um ponto temporal, o primeiro elemento da tupla será um descréscimo de 10 minutos do ponto temporal, de forma que seja possível observá-los na aplicação;
- **get_schedules:** Retorna uma lista de objetos *Schedules* que contém o elemento.
- **get_users:** Retorna uma lista de objetos *Users* que estão conectados ao elemento. Para isso, passa primeiro por cada *Schedule*. Os *Schedules* a serem considerados podem ser passados através de uma lista com seus *IDs*;
- **set_title:** Altera o *title* do elemento. Isso notificará seu *Observer*, *ElementManagement*;
- **set_description:** Altera a *description* do elemento. Isso notificará seu *Observer*, *ElementManagement*;
- **to_dict:** Transforma a instância em um dicionário compatível com o banco de dados;
- **attach:** Se conecta a um *Observer*;
- **detach:** Se desconecta de um *Observer*;
- **notify:** Notifica todos os *Observers*;

### 2.4.2 EventElement

A classe *EventElement* é uma realização da classe abstrata *Element*, e representa um evento que possui data de início e fim definidas. Seus atributos são:

- **__id:** ID único do evento;
- **title:** Título do evento;
- **description:** Descrição do evento;
- **start:** Data de início do evento;
- **end:** Data de término do evento;
- **__schedules:** Lista de IDs dos *Schedules* que o evento faz parte;
- **__element_type:** Tipo do evento, neste caso "event";
- **__observers:** Lista de observers.

Os atributos *__id:*, *__schedules*, *__element_type* e *__observers* possuem método *getter*, sendo que *_schedules* também possui método *setter*. Possui um método único:

- **set_interval:** Altera o *start* e o *end* do evento. Isso notificará seu *Observer*, *ElementManagement*.

### 2.4.3 TaskElement

A classe *TaskElement* é uma realização da classe abstrata *Element*, e representa um tarefa que possui um prazo. Seus atributos são:

- **__id:** ID único da tarefa;
- **title:** Título da tarefa;
- **description:** Descrição da tarefa;
- **due_date:** Prazo da tarefa;
- **state:** Estado atual da tarefa, podendo ser "incomplete", "complete"ou "cancelled";
- **__schedules:** Lista de IDs dos *Schedules* que a tarefa faz parte;
- **__element_type:** Tipo da tarefa, neste caso "task";
- **__observers:** Lista de observers.

Os atributos *__id:*, *__schedules*, *__element_type* e *__observers* possuem método *getter*, sendo que *_schedules* também possui método *setter*. Possui dois métodos únicos:

- **set_due_date:** Altera o *due_date* da tarefa. Isso notificará seu *Observer*, *ElementManagement*.
- **set_state:** Altera o *state* da tarefa. Isso notificará seu *Observer*, *ElementManagement*.

### 2.4.4 ReminderElement

A classe *ReminderElement* é uma realização da classe abstrata *Element*, e representa um lembrete. Seus atributos são:

- **__id:** ID único do lembrete;
- **title:** Título do lembrete;
- **description:** Descrição do lembrete;
- **reminder_date:** Prazo do lembrete;
- **__schedules:** Lista de IDs dos *Schedules* que o lembrete faz parte;
- **__element_type:** Tipo do lembrete, neste caso "reminder";
- **__observers:** Lista de observers.

Os atributos *__id:*, *__schedules*, *__element_type* e *__observers* possuem método *getter*, sendo que *_schedules* também possui método *setter*. Possui um método único:

- **set_reminder_date:** Altera o *reminder_date* do lembrete. Isso notificará seu *Observer*, *ElementManagement*.

### 2.4.5 ElementFactory

A classe *ElementFactory* é uma *Factory* com o objetivo de criar objetos dos demais tipos da interface *Element*. Possui um único método:

- **create_element:** Cria um objeto de uma das realizações da interface *Element* através dos atributos passados. Estes podem ser *EventElement*, *TaskElement* ou *ReminderElement*, definidos pelo atributo *element_type*. Caso o atributo não seja "event", "task"ou "reminder", um erro é levantado.

### 2.4.6 ElementManagement

A classe *ElementManagement* é um *Observer* que manuseia as instâncias de *Element*. Ele possui os seguintes atributos:

- **_instance:** Instância da classe, usada para implementar o *Singleton*;
- **db_module:** Módulo de banco de dados da classe. Utilizamos a classe *MongoModule*;
- **elements:** Dicionário que segura as instâncias ativas de *Elements*. As chaves são os IDs e os valores as instâncias.

Temos os seguintes métodos:

- **get_instance:** *@classmethod* que retorna a instância da classe, utilizada para garantir que nos referenciamos sempre a mesma instância;

- **create_element:** Caso não exista no banco, cria um *Element* com as informações passadas e a adiciona no banco. Uma instância é criada com ajuda do *ElementFactory* e introduzida no dicionário *elements*;
- **update_element:** Atualiza o banco de acordo com os valores de uma instância *Element* dessincronizada através do seu *ID*;
- **delete_element:** Através de seu *ID*, deleta um element do banco e do dicionário *elements*. *Schedules* relacionados também são atualizados;
- **get_element:** Retorna uma instância de *Element* através de seu *ID*. Caso o usuário não esteja no dicionário *elements* mas exista no banco, um objeto é criado para ele;
- **element_exists:** Verifica a existência de um element no banco através de seu *ID*;
- **update:** Método de *Observer*, é chamada quando uma instância de usuário é modificada.

# 3 Diagramas

## 3.1 Diagrama de Casos de Uso

O diagrama de Casos de Uso foi baseado nos casos de Uso entregues na primeira etapa do projeto, mostrando as interações entre atores e o sistema e representando os requisitos funcionais do sistema e como os usuários interagem com ele.

Foram realizadas pequenas modificações de nomeações e acrescentados mais alguns casos de uso (como login e criação de conta), que foram implementados.

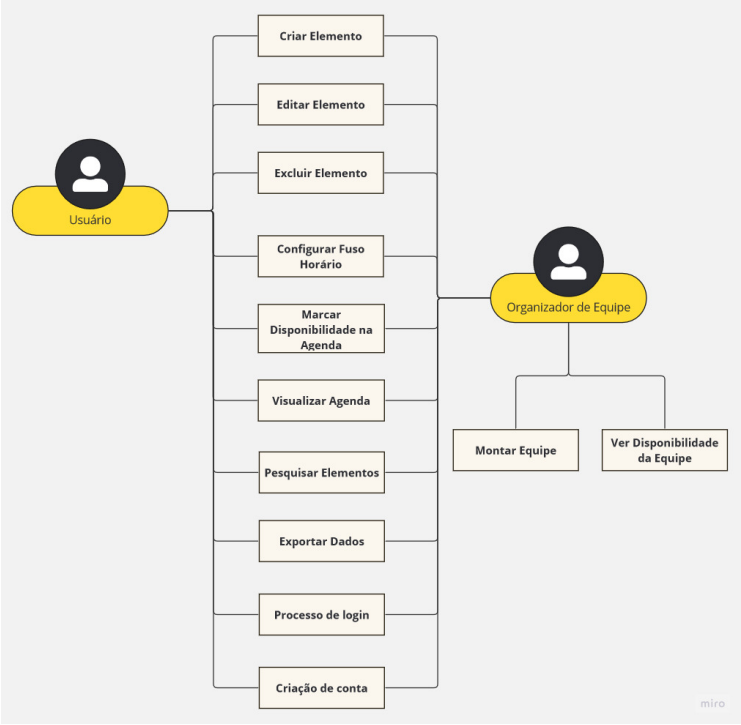


Figura 1: Diagra de Casos de Uso

## 3.2 Diagrama de Classe

Esse diagrama descreve bem a estrutura estática do nosso sistema, mostrando classes, seus atributos, métodos e relacionamentos entre elas. Ele foi bem útil para entender a estrutura do código e a organização das classes no sistema.

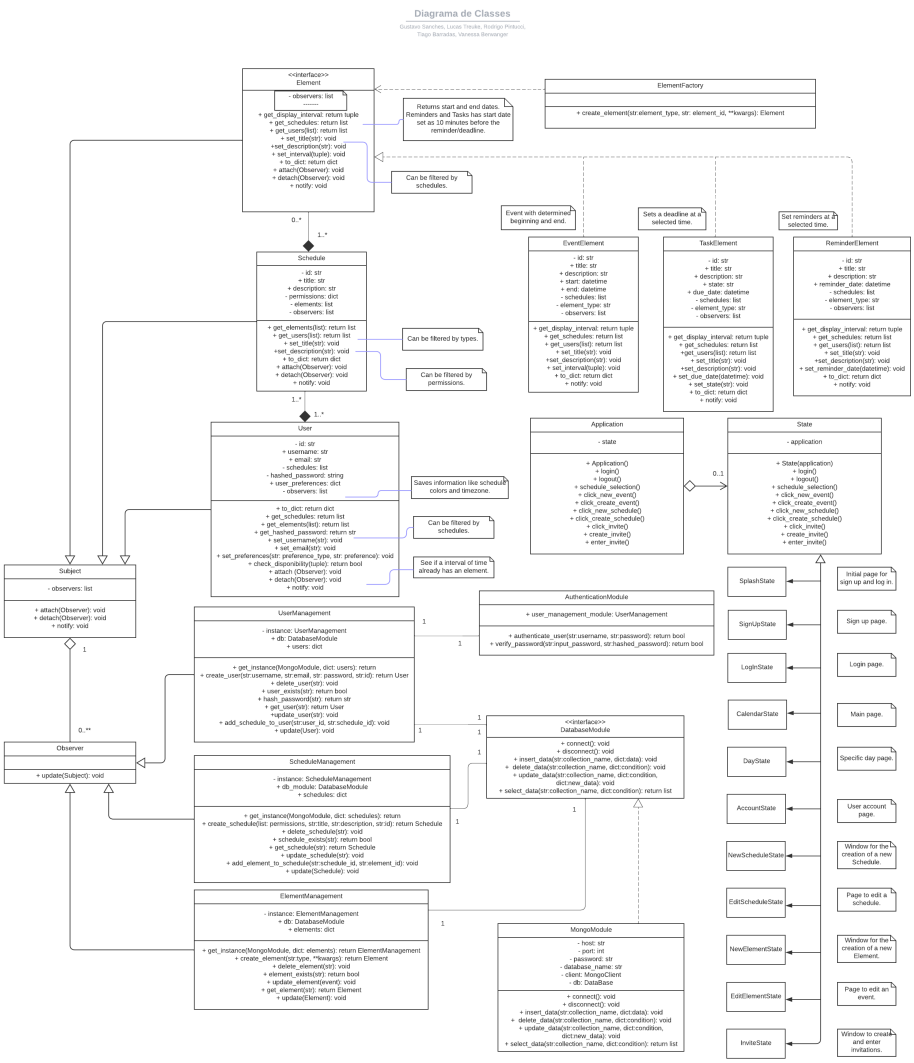


Figura 2: Diagra de Classe

## 3.3 Diagrama de Pacote

O diagrama de Pacote abaixo mostra como os elementos do sistema desenvolvido estão agrupados em pacotes, oferecendo uma visão geral da organização lógica do sistema. Assim, podemos entender melhor a modularidade e a estrutura de dependência entre diferentes partes do sistema.

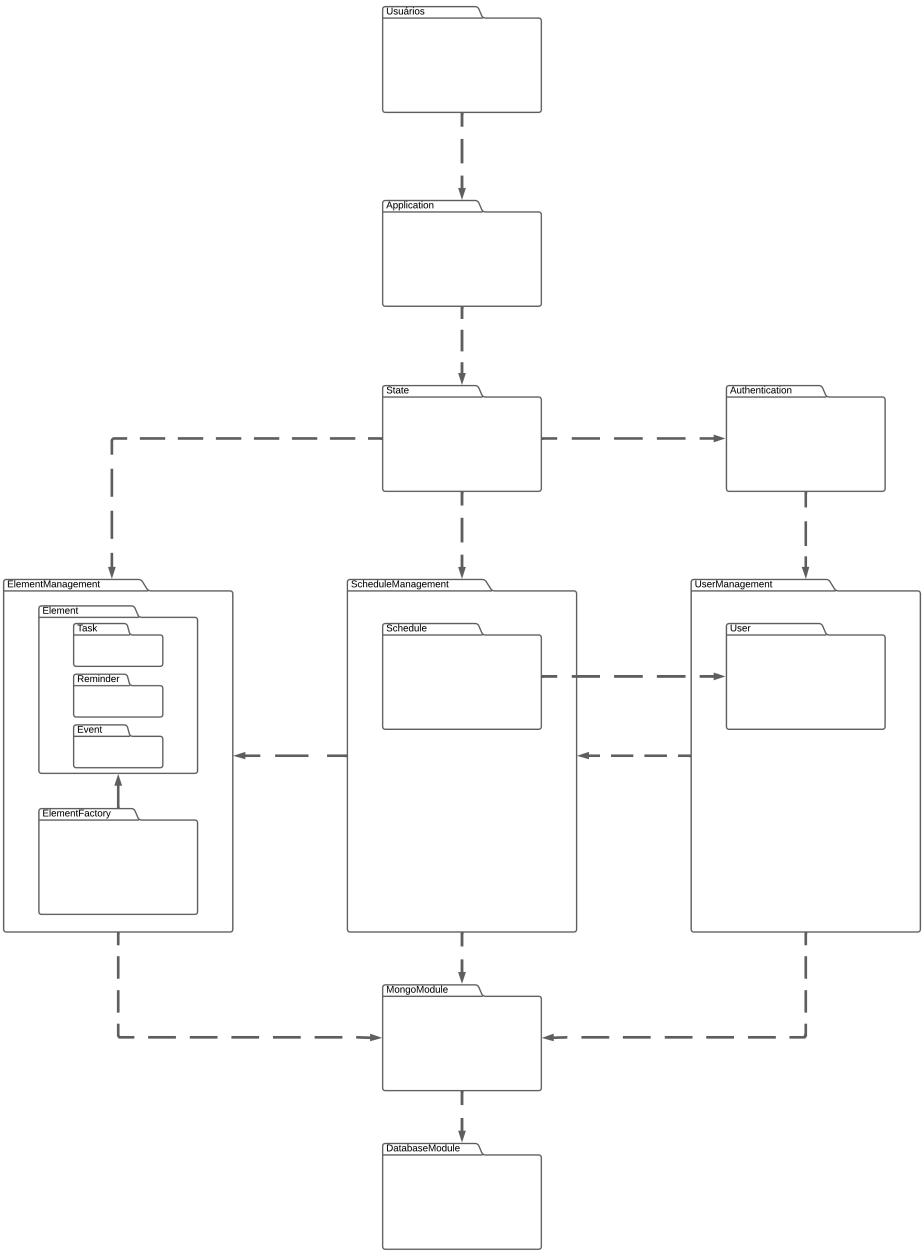


Figura 3: Diagra de Pacote

## 3.4 Diagrama de Componentes

Pelo diagrama abaixo pode ver os componentes de software do sistema e suas relações. Esse diagrama se mostra útil para comunicar e explicar as funções do nosso sistema às partes interessadas.

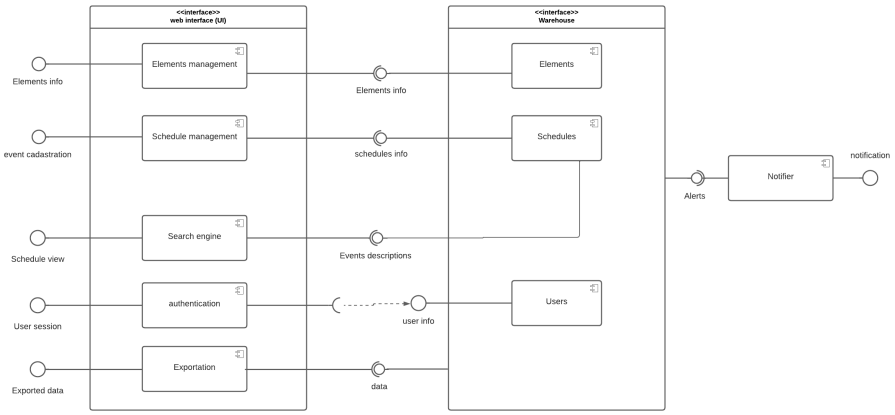


Figura 4: Diagra de Componentes

## 3.5 Diagrama de Sequência

O diagrama de Sequência modela a interação entre objetos em uma determinada sequência temporal, mostrando como os objetos se comunicam ao longo do tempo para realizar uma funcionalidade específica. Assim, diferentes cenários foram representados por diagramas de sequência separados, ilustrando como a interação entre objetos ocorre em cada situação específica. Esses diagrmas podem ser encontrados no caminho '/docs/diagramas de sequencia'.

# 4 Padrões de Sistema

Ao longo do desenvolvimento do nosso projeto, utilizamos principalmente de seis padrões de projeto:

## 4.1 Criação (2)

Os padrões de criação estão relacionados ao processo de instanciação de objetos. Eles lidam com a forma como os objetos são criados, garantindo flexibilidade e eficiência.

### 4.1.1 Singleton

O primeiro padrão de todos escolhido foi o **Singleton**. Esse padrão se resume a garantir que objetos específicos não possam ter mais de uma instância. A implemntação feita não considera o caso multi-threading, uma vez que o nosso aplicativo não está instância nenhuma classe paralelamente.

Dessa maneira, conseguimos garantir que o Banco de dados e as classes "managers"tenham apenas uma instância, não havendo disparidade de informação.

Duas implementações foram feitas, a primeira foi definindo um método especial "get_instance()"e a segunda foi verificando se a classe já foi instânciada antes no método "__new__".

### 4.1.2 Factory

O **Factory** com poucas horas de estudo se mostrou ideal para o nosso caso de uso. Possuímos diversos tipos de eventos, com alguns métodos e atributos em comuns, outros nem tanto. Sendo assim, implementamos uma classe "*Factory*"que lida com a criação de todo e qualquer evento, criando as instâncias das classes "menores".

## 4.2 Estruturais (1)

Os padrões estruturais estão focados na composição de classes e objetos para formar estruturas maiores e mais complexas. Eles ajudam a garantir que os sistemas sejam flexíveis e fáceis de compreender.

Dessa categoria foram utilizados apenas um padrão de projeto, sendo que os outros três são comportamentais.

### 4.2.1 Decorator

A escolha do **Decorator** foi orientada por uma consideração primordial: versatilidade. Este padrão revela-se uma ferramenta incrivelmente flexível, abrindo portas para uma variedade de aplicações. No nosso contexto específico, decidimos explorar essa versatilidade ao criar um decorator que, de maneira elegante, lança uma exceção 'Timeout' sempre que envolve uma função.

Em questões específicas de implementação, fez-se necessário verificar se o usuário está utilizando um *SO Unix*, ou não. Pois o módulo "singnal"não é compátivel com outros sistemas operacionais.

## 4.3 Comportamentais (3)

Os padrões de comportamento estão relacionados à interação eficiente e responsável entre objetos. Eles definem como os objetos colaboram e se comunicam.

### 4.3.1 State

O **State** cria um um fluxo para as telas da aplicação. Na prática, cada tela de interface é um state, na qual pode sofrer transição para o próximo estado, estado anterior e etc.

Com essa implementação, conseguimos abordar de forma muito fluída a orientação de objetos necessária para a utilização do TKinter.

### 4.3.2 Observer

O **Observer** têm como objetivo averiguar a ocorrência de alguma evento definido. Na nossa aplicação, cada uma das classes "manager"são observadores dos métodos de updates de suas respectivas classes menores. Então, por exemplo, a "UserManager"observa o "User.update"para que sempre que ocorrer, outros aspectos do user também sejam atualizados, como as schedules.

### 4.3.3 Template

Por fim, o último padrão de projeto desenvolvido foi o template, que provê uma padrão a ser seguido pelas classes filhas, atravez de métodos abstratos, funções e hookers. Esse padrão foi implementado para ser um *template* para o módulo do banco de dados e para os elementos do calendário.

## 4.4 Arquivos

| **Padrão** | **Caminho** relativo |
|---|---|
| Singleton | src/database/mongo_module.py<br>src/auth/user_management.py<br>src/schedule/schedule_management.py<br>src/calendar_elements/element_management.py |
| Factory | src/calendar_elements/element_factory.py |
| Decorator | src/database/utils.py |
| State | src/app/app_states.py |
| Observer | src/observer/observer.p y |
| Template | src/database/database_module.py<br>src/calendar_elements/element_interface.py |

Tabela 1: Caminhos para os padrões de projeto

# 5 DocTest para unidades simples

Aplicamos o uso de DocTests na documentação da classe User, no arquivo *user_model.py*, em unidades simples, que não possuem caminhos alternativos, como funções getter.

```
def get_hashed_password(self) -> str:
    """
    Get the user hashed password

    Returns:
        The user hashed password

    >>> user = User("id", "username", "email", ["id1", "id2"], "hashed_password")
    >>> user.get_hashed_password()
    'hashed_password'
    """
    return self.__hashed_password
```

Figura 5: Doc test para função *get_hashed_password*

```
def set_username(self, username: str):
    """
    Set the user name

    Args:
        username: user name

    >>> user = User("id", "username", "email", ["id1", "id2"])
    >>> user.set_username("newusername")
    >>> user.username
    'newusername'
    """
```

Figura 6: Doc test para função *set_user*

```
def set_email(self, email: str):
    """
    Set the user name

    Args:
        username: user name

    >>> user = User("id", "username", "email", ["id1", "id2"])
    >>> user.set_email("new_email")
    >>> user.email
    'new_email'
    """
```

Figura 7: Doc test para função *set_user_email*

# 6 Testes unitários

Foram realizados diversos testes, que podem ser encontrados na pasta *'tests'*. É importante ressaltar que nessa parte fizemos uso de *'MagicMock'*, classe fornecida pelo módulo unittest.mock em Python, usada principalmente para criar objetos fictícios (ou *mocks*) que imitam o comportamento de objetos reais. Com isso, buscamos a eliminação de dependências externas (que podem introduzir variabilidade nos testes), isolando mais o código que está sendo testado, garantindo que qualquer falha seja devido ao próprio código testado e não a problemas externos, tornando-o mais flexível e fácil de manter.

Assim, alinhamos práticas de padrões de qualidade de código ao fazer o uso de *'MagicMock'* para criar testes mais confiáveis e focados no comportamento específico do código que está sendo testado. Passamos um tempo considerável tentando compreender o funcionamento do MagicMock e, em várias ocasiões, enfrentamos dificuldades, nas quais passamos bastante tempo.

Todos os testes podem ser encontrados na pasta *tests*, além dos históricos de *TDDs*. Também obtemos a cobertura do código utilizando o módulo *coverage.py* com o Pytest, e a registramos no arquivo *documentacao/Relatório_coverage.pdf*.

# 7 TDDs e refactor

Os TDDs (códigos antes e depois de passarem em testes) podem ser encontrados em notebooks de *'history'* de algumas classes, registradas em arquivos de *jupyter notebook*, encontrados dentro das pastas de tests:

- /tests/test_auth/auth_history
- /tests/test_events/history_task
- /tests/test_mongomodule/mongo_module_history
- /tests/test_schedule/history_schedule
- /tests/test_schedule/history_schedule_management
- /tests/test_schedule/history_schedule_observer
- /tests/test_users/tes_user_model

Alguns desses arquivos contêm exemplos de refatoração, mas eles foram feitos principalmente por meio de commits. O objetivo era aprimorar o código existente, que já funcionava, mas apresentava espaço para melhorias.

# 8 Tratamento de excessões com User-Defined Data Type

Criamos diversas classes de excessões que explicam melhor os casos de erro que podem ser encontrados nas funções do projeto:

- **NonExistentIDError:** Levantado quando uma instância de *user*, *schedule* ou *element* com *ID* não registrado é requisitado;
- **DuplicatedIDError:** Levantado quando tenta-se criar ou inserir uma instância de *user*, *schedule* ou *element* caso seu *ID* já exista no banco ou na lista a inserir;
- **ConnectionDBError:** Levantado quando não é possível se conectar ao DB ou quando tenta-se conectar mais de uma vez;
- **UserNotFound**: Levantado quando o nome de usuário passado não condiz com uma instância de usuário registrado.
- **EmptyPermissionsError:** Levantado quando o atributo *permissions* de um *schedule* está vazio.
- **UserNotInSchedule:** Levantado quando um *schedule* inválido é passado como filtro na função *get_schedules* do *user*.
- **UsernameCantBeBlank:** Levantado quando o atributo *username* fornecido é vazio.
- **EmailCantBeBlank:** Levantado quando o atributo *email* fornecido é vazio.
- **DatabaseNotProvidedError:** Levantado quando um *database* não é fornecido.

- **TimeExceedError:** Levantado quando a função decorada permanece muito tempo rodando, apenas para o banco.

```python
class DuplicatedIDError(Exception):
    """Raised when the ID already exists"""

class NonExistentIDError(Exception):
    """Raised when the ID does not exist"""

class ConnectionDBError(Exception):
    """Raised when the connection to the database fails"""

class MongoModule(DatabaseModule):
    """
    This class implements the DatabaseModule interface for MongoDB.
```

Figura 8: Exemplos de UDTs.

# 9 Revisão e qualidade de código

As revisões de código foram conduzidas usando *pull requests*. Os membros trabalhavam em branches separadas e, ao abrir um *pull request*, outro membro revisava o código, fornecendo comentários e fazendo o merge quando necessário.

Além disso, utilizamos o SonarQube e SonarScanner como ferramentas de verificação de qualidade do código, para nos ajudar a perceber que partes do código poderiam ser refatoradas. A avaliação do estado final do código estará disponível no arquivo *documentacao/Qualidade_Codigo.pdf*.

# 10 Uso de DocString. PEP 257 e PEP 8

Buscamos deixar o código bem documentado usando PEP 257 (*DocStrings*), utilizando, por exemplo:

- Documentação de funções, métodos, classes e módulos com docstrings.
- Descrição do propósito da função/método, detalhes dos parâmetros, valores de retorno e informações relevantes.
- Siguindo as convenções para docstrings de uma linha e para docstrings multi-linha.
- Uso de três aspas (simples ou duplas) para envolver o texto da docstring.

Também, procuramos deixar o código limpo e organizado em módulos (PEP 8), utilizando, por exemplo:

- Limite de caracteres em linhas.
- Identação de 4 espaços.
- Uso de convenções de nomenclatura: nomes de variáveis em minúsculas separadas por sublinhados (*snake_case*), nomes de classes em *CamelCase*.

- Utilize espaços em torno de operadores e após vírgulas.
- Estilo consistente ao longo do código.