Trabalho A2 de Engenharia de Software

Amanda Perez Dominique de Vargas Gabriel de Melo
Juan Belieni Lindsey de Vargas

10 de dezembro de 2023

Sumário

1 Planejamento, análise e design do sistema

1.1 Diagramas de casos de uso

Os diagramas de caso de uso se encontram no diretório `/docs` no repositório.

1.2 Diagramas de classe

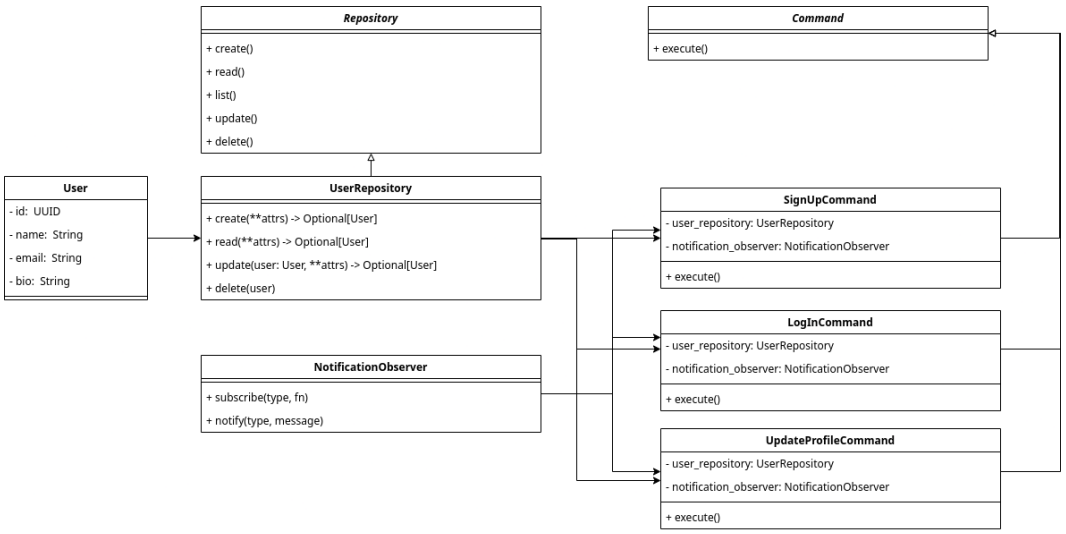

Figura 1: Diagrama de classe de `User`, `UserRepository`, `LogInCommand`, `SignUpCommand` e `UpdateProfileCommand`.

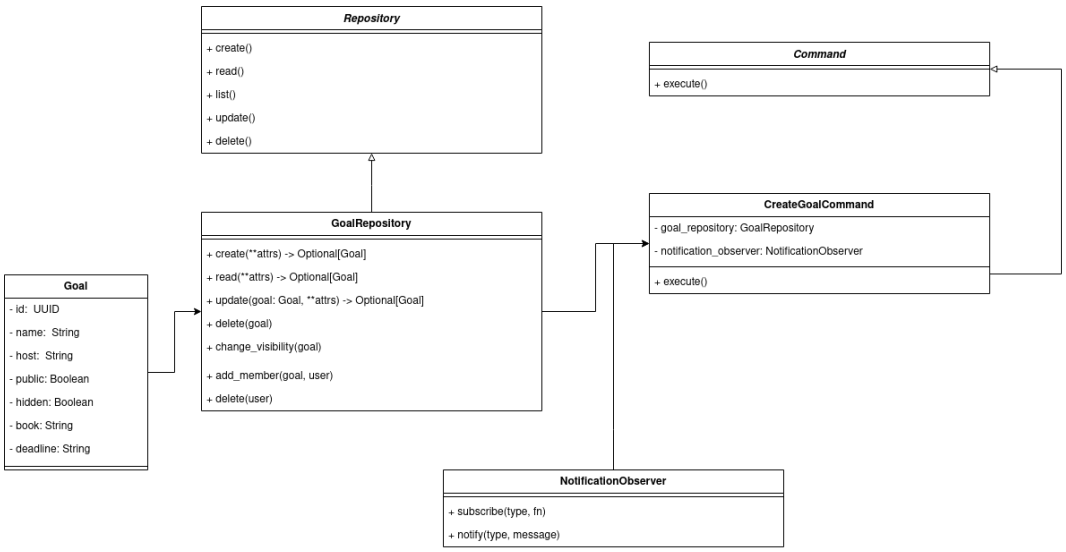

Figura 2: Diagrama de classe de `Goal`, `GoalRepository` e `CreateGoalCommand`.

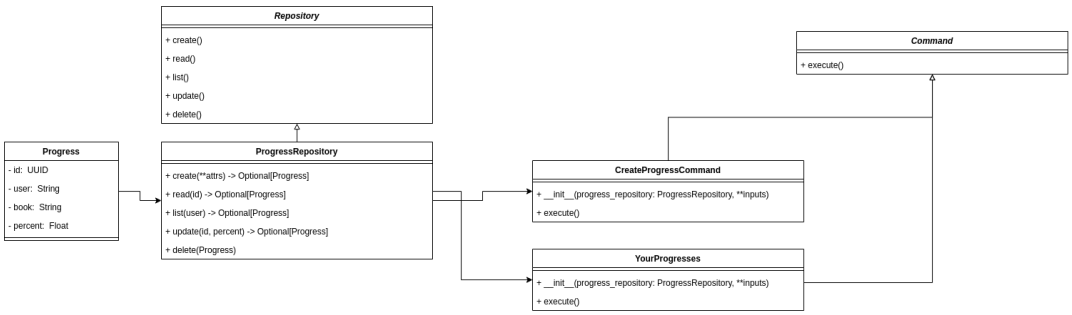

Figura 3: Diagrama de classe de `Progress` e `ProgressRepository`.

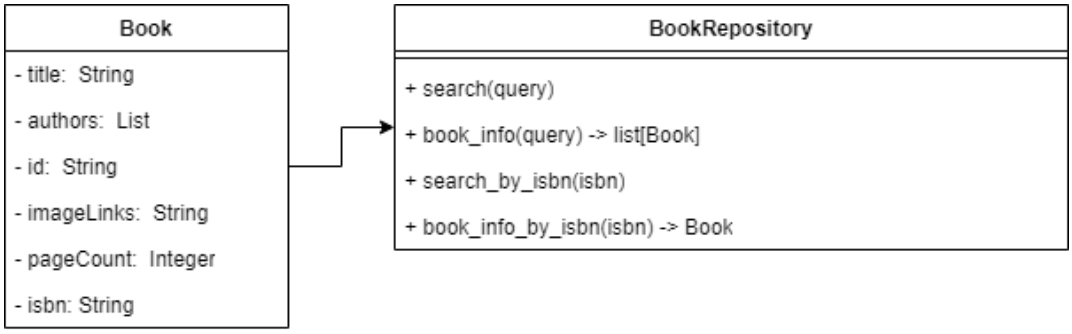

Figura 4: Diagrama de classe de `Book` e `BookRepository`.

1.3 Diagrama de pacote

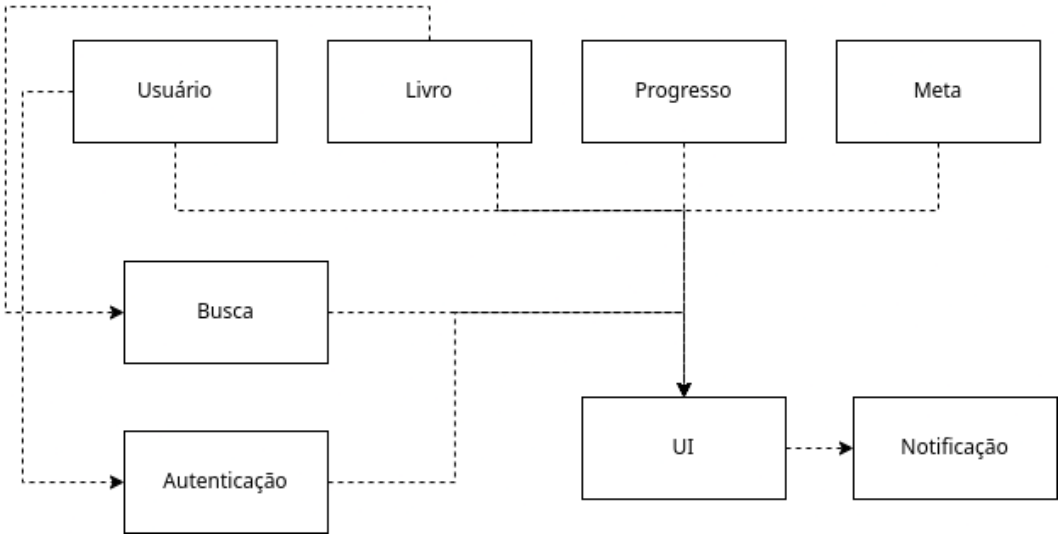

Figura 5: Diagrama de pacotes incluindo os principais componentes do sistema.

2 Desenvolvimento

2.1 Padrões de projeto

2.1.1 Façade (estrutural)

O *façade* é um padrão de projeto que se baseia em criar uma inteface simplificada para outro sistema. Nesse trabalho, utilizamos esse padrão para simplificar as chamadas ao banco a partir da definição de uma classe abstrata `Repository`.

```
class Repository(ABC):
    @abstractmethod
    def create(self): ...

    @abstractmethod
    def read(self): ...

    @abstractmethod
    def list(self): ...

    @abstractmethod
    def update(self): ...

    @abstractmethod
    def delete(self): ...
```

2.1.2 Command (comportamental)

O padrão de projeto *command* se baseia em transformar um pedido em um objeto independente que contém toda a informação sobre o pedido. Um exemplo de implementação desse padrão pode ser visto a seguir.

```
class SignUpCommand(Command):
    app: App
    user_repository: UserRepository
    name_input: TextInput
    email_input: TextInput
    password_input: TextInput
    email_input: TextInput

    def __init__(
        self,
        app: Optional[App] = None,
        user_repository=user_repository,
        **inputs: TextInput
    ):
        self.app = app or App.get_running_app()
        self.user_repository = user_repository
        self.name_input = inputs['name_input']
        self.email_input = inputs['email_input']
        self.password_input = inputs['password_input']
        self.bio_input = inputs['bio_input']

```

```
    def execute(self):
        name = self.name_input.text
        email = self.email_input.text
        password = self.password_input.text
        bio = self.bio_input.text

        user = self.user_repository.create(
            name=name,
            email=email,
            password=password,
            bio=bio,
        )

        if user is not None:
            self.app.root.current = 'log_in'
```

2.1.3 Template (comportamental)

Utilizamos esse padrão de projeto para modelar alguns padrões de classes que usamos com frequência. Por exemplo, criamos as classes abstratas `Repository` (definida anteriormente) e `Command` (definida a seguir) seguinte esse padrão:

```
class Command(ABC):
    @abstractmethod
    def execute(self): ...
```

2.1.4 Observer (comportamental)

O *observer* é um padrão que permite que várias partes do código possam observar mudanças ou eventos de um determinado objeto. A principal implementação de *observer* que tivemos foi na class `NotificationObserver`, que permite a notificação de mensagens de dois tipos, *success* e *failure*.

```
class NotificationObserver:
    subscribers: list

    def __init__(self):
        self.subscribers = defaultdict(list)

    def subscribe(self, type: Type, fn):
        self.subscribers[type].append(fn)

    def unsubscribe(self, type: Type, fn):
        self.subscribers[type].remove(fn)

    def notify(self, type: Type, message: str):
        for fn in self.subscribers[type]:
            fn(message)
```

2.1.5 Singleton (criação)

O padrão *singleton* é muito simples, pois se resume em criar uma única instância de uma classe com acesso global. Ela é bem útil para diminuir a duplicação de código. Em todo o projeto, é possível encontrar *singlestons* (principalmente de repositórios e de utilitários).

2.1.6 Builder (criação)

O padrão *Builder* é útil quando um objeto complexo precisa ser construído passo a passo. Ele separa a construção de um objeto complexo da sua representação, permitindo que o mesmo processo de construção crie diferentes representações do objeto. Neste projeto ele foi utilizado para construir containers para as vizualizações.

```
class BoxLayoutBuilder(BoxLayout):

    def set_orientation(self, orientation):
        self.orientation = orientation
        return self

    def set_spacing(self, spacing):
        self.spacing = spacing
        return self

    def set_padding(self, padding):
        self.padding = padding
        return self

    def set_size(self, size):
        self.size = size
        return self

    def set_size_hint(self, size_hint):
        self.size_hint = size_hint
        return self

    def build(self): ...

class BoxProgress(BoxLayoutBuilder):

    def build(self):
        self.set_orientation('vertical')
        self.set_spacing(10)
        self.set_padding(10)
        self.set_size((400, 200))

        title_label = Label(text='Seus Progressos',
                            size_hint_y=None, height=44)
        self.add_widget(title_label)

        return self

    def add_widget_progress(self, book, percent, id):

        book_label = Label(text=book, size_hint_y=44, height=20)
        self.add_widget(book_label)

        porcent_label = Label(text=str(percent),
                              size_hint_y=44, height=20)
        self.add_widget(porcent_label)

```

```
        delete_button = Button(text='Deletar',
                               size_hint_y=None, height=44)
        delete_button.bind(
            on_press=lambda _: YourProgressesCommand(command="delete", id=id).execute())
        self.add_widget(delete_button)
```

2.2 Histórico de modificações

2.2.1 UserRepository

```
class UserRepository(Repository):
    db: DB

    def __init__(self, db: DB):
        self.db = db

    def create(self): ...

    def read(self): ...

    def update(self): ...

    def delete(self): ...
```

```
# ...
    def create(self, **attrs) -> Optional[User]:
        id = str(uuid4())
        name = attrs.get("name")
        email = attrs.get("email")
        bio = attrs.get("bio") or ""
        password = sha256(attrs["password"].encode("utf-8")).hexdigest()

        result = self.db.execute(
            "insert into user value (?, ?, ?, ?, ?)",
            (id, name, email, bio, password)
        )

        if result is None or len(result) == 0:
            return None

        return User(
            id=result[0][0],
            name=result[0][1],
            email=result[0][2],
            bio=result[0][3],
        )

    def read(self, id=None, password=None) -> Optional[User]:
        if id is not None:
            result = self.db.execute(
                "select id, name, email, bio in user where id = ?",
                (id,)
            )
        elif password is not None:
```

```
            password = sha256(password.encode("utf-8")).hexdigest()
            result = self.db.execute(
                "select id, name, email, bio in user where password = ?",
                (password,)
            )
        else:
            return None

        if result is None or len(result) == 0:
            return None

        return User(
            id=result[0][0],
            name=result[0][1],
            email=result[0][2],
            bio=result[0][3],
        )

    def update(self, user: User, **attrs) -> Optional[User]:
        name = attrs.get("name") or user.name
        bio = attrs.get("bio") or user.bio

        result = self.db.execute(
            "update user name = ?, set bio = ? where id = ?",
            (name, bio, id)
        )

        if result is None or len(result) == 0:
            return None

        return User(
            id=result[0][0],
            name=result[0][1],
            email=result[0][2],
            bio=result[0][3],
        )

    def delete(self, user: User):
        self.db.execute(
            "delete user where id = ?",
            (user.id,)
        )
```

```
# ...
    def create(self, **attrs) -> Optional[User]:
        # ...
        return User(*result[0])

    def read(self, id=None, password=None) -> Optional[User]:
        # ...
        return User(*result[0])

    def update(self, user: User, **attrs) -> Optional[User]:
        # ...
```

```
        return User(*result[0])
# ...
```

```
# ...
    def read(self, **attrs) -> Optional[User]:
        wheres = []
        params = tuple()

        for key, value in attrs.items():
            if key == "password":
                value = sha256(value.encode("utf-8")).hexdigest()

            wheres.append(f"{key} = ?")
            params = (*params, value)

        where = " and ".join(params)

        result = self.db.execute(
            f"select id, name, email, bio in user where {where}",
            params
        )

        if result is None or len(result) == 0:
            return None

        return User(*result[0])
# ...
```

```
# ...
def create(self, **attrs) -> Optional[User]:
    id = str(uuid4())
    name = attrs.get("name")
    email = attrs.get("email")
    bio = attrs.get("bio") or ""
    password = sha256(attrs["password"].encode("utf-8")).hexdigest()

    result = self.db.execute(
        "insert into user values (?,?,?,?,?)",
        (id, name, email, bio, password)
    )

    if result is None:
        return None

    return self.read(id=id)
# ...
   def update(self, user: User, **attrs) -> Optional[User]:
        name = attrs.get("name") or user.name
        bio = attrs.get("bio") or user.bio

        result = self.db.execute(
            "update user set name = ?, bio = ? where id = ?",
            (name, bio, user.id)
        )

```

```
        if result is None:
            return None

        return self.read(id=user.id)
# ...
```

```
.... o último
```

2.2.2 SignUpCommand

```
class SignUpCommand(Command):
    # ...

    def __init__(
        self,
        user_repository=user_repository,
        **inputs: TextInput
    ):
        self.user_repository = user_repository
        self.name_input = inputs['name_input']
        self.email_input = inputs['email_input']
        self.password_input = inputs['password_input']
        self.bio_input = inputs['bio_input']

    def execute(self): ...
```

```
# ...
    def execute(self):
        name = self.name_input.text
        email = self.email_input.text
        password = self.password_input.text
        bio = self.bio_input.text

        return user_repository.create(
            name=name,
            email=email,
            password=password,
            bio=bio,
        )
```

```
# ...
    def __init__(
        self,
        app: Optional[App] = None,
        user_repository=user_repository,
        **inputs: TextInput
    ):
        self.app = app or App.get_running_app()
        self.user_repository = user_repository
        self.name_input = inputs['name_input']
        self.email_input = inputs['email_input']
        self.password_input = inputs['password_input']
        self.bio_input = inputs['bio_input']

```

```
    def execute(self):
        name = self.name_input.text
        email = self.email_input.text
        password = self.password_input.text
        bio = self.bio_input.text

        user = self.user_repository.create(
            name=name,
            email=email,
            password=password,
            bio=bio,
        )

        if user is not None:
            self.app.root.current = 'log_in'
```

```
# ...
    def __init__(
        self,
        app: Optional[App] = None,
        user_repository=user_repository,
        notification_observer=notification_observer,
        **inputs: TextInput
    ):
        self.app = app or App.get_running_app()
        self.user_repository = user_repository
        self.notification_observer = notification_observer
        self.name_input = inputs['name_input']
        self.email_input = inputs['email_input']
        self.password_input = inputs['password_input']
        self.bio_input = inputs['bio_input']

    def execute(self):
        name = self.name_input.text
        email = self.email_input.text
        password = self.password_input.text
        bio = self.bio_input.text

        user = self.user_repository.create(
            name=name,
            email=email,
            password=password,
            bio=bio,
        )

        if user is not None:
            self.app.root.current = 'log_in'
        else:
            self.notification_observer.notify(
                "failure",
                "Não foi pessoal criar a conta"
            )
```

2.2.3 LogInCommand

```
class LogInCommand(Command):
    # ...

    def __init__(
        self,
        user_repository=user_repository,
        **inputs: TextInput
    ):
        self.user_repository = user_repository
        self.email_input = inputs['email_input']
        self.password_input = inputs['password_input']

    def execute(self): ...
```

```
# ...
    def execute(self):
        email = self.email_input.text
        password = self.password_input.text

        user_repository.read(
            email=email,
            password=password,
        )
```

```
# ...
    def __init__(
        self,
        app: Optional[App] = None,
        user_repository=user_repository,
        **inputs: TextInput
    ):
        self.app = app or App.get_running_app()
        self.user_repository = user_repository
        self.email_input = inputs['email_input']
        self.password_input = inputs['password_input']

    def execute(self):
        email = self.email_input.text
        password = self.password_input.text

        user = self.user_repository.read(
            email=email,
            password=password,
        )

        if user is not None:
            self.app.user = user
            self.app.root.current = "profile"
```

```
# ...
    def __init__(
        self,
        app: Optional[App] = None,
```

```
        user_repository=user_repository,
        notification_observer=notification_observer,
        **inputs: TextInput
    ):
        self.app = app or App.get_running_app()
        self.user_repository = user_repository
        self.notification_observer = notification_observer
        self.email_input = inputs['email_input']
        self.password_input = inputs['password_input']

    def execute(self):
        email = self.email_input.text
        password = self.password_input.text

        user = self.user_repository.read(
            email=email,
            password=password,
        )

        if user is not None:
            self.app.user = user
            self.app.root.current = "profile"
        else:
            self.notification_observer.notify("failure", "Usuário não encontrado")
```

2.2.4 UpdateProfileCommand

```
class UpdateProfileCommand(Command):
    app: App
    user_repository: UserRepository
    notification_observer: NotificationObserver
    name_input: TextInput
    bio_input: TextInput

    def __init__(
        self,
        app: Optional[App] = None,
        user_repository=user_repository,
        notification_observer=notification_observer,
        **inputs: TextInput
    ):
        self.app = app or App.get_running_app()
        self.user_repository = user_repository
        self.notification_observer = notification_observer
        self.name_input = inputs['name_input']
        self.bio_input = inputs['bio_input']

    def execute(self): ...
```

```
# ...
    def execute(self):
        name = self.name_input.text
        bio = self.bio_input.text

```

```
        user = self.user_repository.update(
            self.app.user,
            name=name,
            bio=bio,
        )

        if user is not None:
            self.app.user = user
            self.notification_observer.notify("success", "Perfil atualizado")
        else:
            self.notification_observer.notify(
                "failure",
                "Não foi pessoal atualizar o perfil"
            )
```

2.2.5 NotificationObserver

```
class NotificationObserver:
    subscribers: list

    def __init__(self): ...

    def subscribe(self, type: Type, fn): ...

    def unsubscribe(self, type: Type, fn): ...

    def notify(self, type: Type, message: str): ...
```

```
# ...
    def __init__(self):
        self.subscribers = defaultdict(list)

    def subscribe(self, type: Type, fn):
        self.subscribers[type].append(fn)

    def unsubscribe(self, type: Type, fn):
        self.subscribers[type].remove(fn)

    def notify(self, type: Type, message: str):
        for fn in self.subscribers[type]:
            fn(message)
```

3 Testes e qualidade de software

Os testes foram desenvolvidos utilizando as biblitecas `pytest` (para rodar os testes) e `unittest` (para realizar *mocks*). A cobertura de testes se mostrou boa, e para visualizá-la foi utilizado o código disponível no arquivo `Makefile`. Para rodar o código e exibir a cobertura de testes obtida, basta rodar `make coverage`. A tabela a seguir apresenta os resultados obtidos.

3.1 Cobertura de código

Name	Stmts	Miss	Cover	Missing
src/__init__.py	0	0	100%	
src/models/book.py	68	17	75%	28, 30, 32, 60-77
src/models/goal.py	79	2	97%	105, 152
src/models/progress.py	44	1	98%	60
src/models/user.py	53	2	96%	125, 148
src/utils/command.py	4	0	100%	
src/utils/db.py	20	13	35%	10-27
src/utils/notification.py	15	0	100%	
src/utils/repository.py	12	0	100%	
src/views/auth/log_in.py	48	15	69%	107-136
src/views/auth/profile.py	53	19	64%	109-155
src/views/auth/sign_up.py	57	19	67%	117-154
src/views/home/your_goals.py	72	23	68%	39, 47, 60, 86-134
TOTAL	525	111	79%	

Tabela 1: cobertura de código.