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.

Engenharia de software é sobre o processo Valorizar o processo no desenvolvimento de software é crucial para alcançar resultados de qualidade e duradouros. Muitas vezes, a pressa em atingir o produto final pode levar a atalhos que comprometem a integridade e eficiência do software. O desenvolvimento de software é um processo complexo que exige planejamento cuidadoso, design sólido, codificação precisa e testes rigorosos. Cada etapa desse processo contribui para a construção de um produto final robusto e confiável.

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.

3 Padrões de Sistema

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

3.1 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.

3.1.1 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__".

3.1.2 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".

3.2 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.

3.2.1 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.

3.3 Comportamentais (3)

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

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.4 Arquivos

Padrão	**Caminho** relativo
Singleton	src/database/mongo_module.py src/auth/user_management.py src/schedule/schedule_management.py 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 src/calendar_elements/element_interface.py

Tabela 1: Caminhos para os padrões de projeto

4 TDDs e refactor

Os TDDs podem ser encontrados em notebooks de 'history' de algumas classes, encontrados dentro das pastas de tests. Os documentos com os TDDs (ou seja, códigos antes e depois de passarem em testes) estão em:

- /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 incluem exemplos de refactor, mas, no geral, eles foram feitos através de commits, visando melhorar o código antigo, que já estava funcional mas poderia ser melhorado.

5 Revisão de código

As revisões de código foram feitas por meio de pull request, onde os integrantes trabalhavam em branchs e, ao dar pull request, outro integrante verificava o código, comentando e dando merge.

6 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.

7 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, como 'UsernameCantBeBlank' ou 'UserNotInSchedule'.

8 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, isolando mais o código que está sendo testado, eliminando dependências externas e garantindo que qualquer falha seja devido ao próprio código testado e não a problemas externos.

Dependências externas podem introduzir variabilidade nos testes. Assim, códigos com poucas dependências externas é mais flexível e fácil de manter.

Assim, procuramos fazer o uso de 'MagicMock' para criar testes mais confiáveis e focados no comportamento específico do código que está sendo testado, práticas alinhadas com

padrões de qualidade de código. Passamos um tempo considerável tentando compreender o funcionamento do MagicMock e, em várias ocasiões, enfrentamos dificuldades que nas quais passamos bastante tempo.

9 Uso de DocString. PEP 257 e PEP 8

Buscamos deixar o código bem documentado usando PEP 257 (DocStrings):

- 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):

- 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.