Módulo 7 - Atividade Individual

Questões de Implementação e Estudo de Caso

Wellington M. Espindula

Julho de 2021

1. **GitHub.** Escolha um projeto da plataforma GitHub (https://github.com/). Faça:

   a. O clone do repositório;

   b. Uma modificação no código;

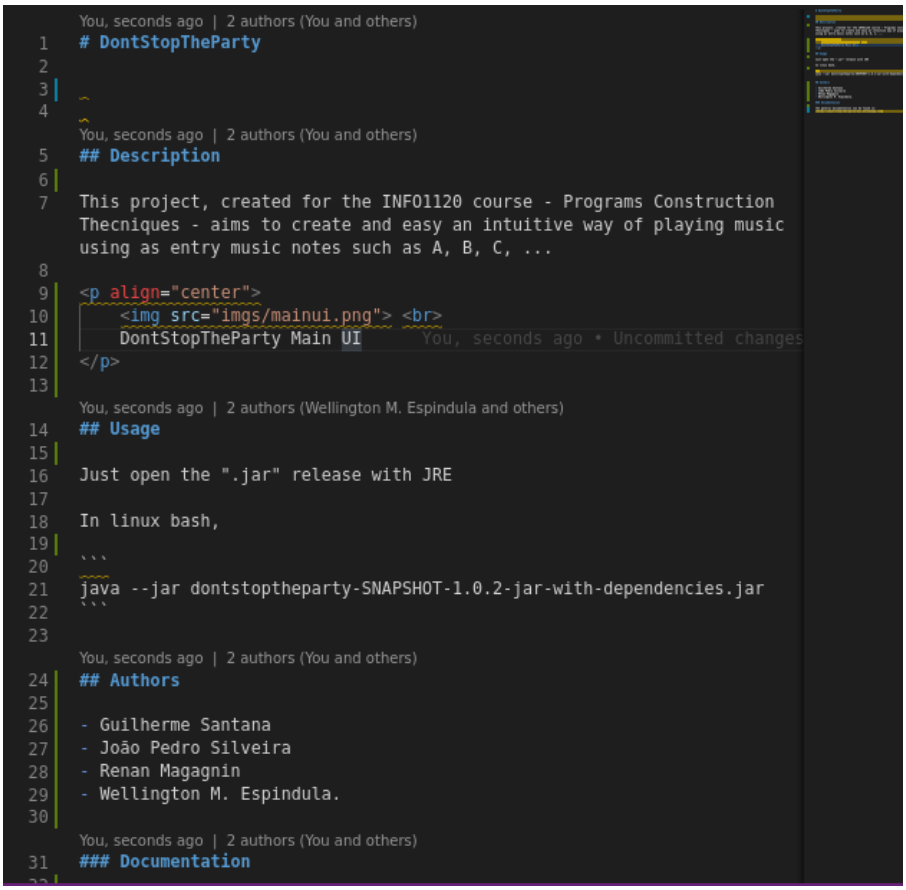
   c. Um commit (sem fazer o push).

   Coloque no relatório uma captura de tela de cada um desses passos.
   Obs.: Você pode fazer isso através de linha de comando ou utilizando um IDE, tal como o Eclipse. É esperado que busque na Internet (e.g. https://education.github.com/git-cheat-sheet-education.pdf) como fazer concretamente estes passos (que foram conceitualmente explicados na aula). Será necessária a criação de uma conta na plataforma GitHub

   Resposta: O repositório escolhido para a tarefa foi o do projeto DontStopTheParty - aplicativo desktop que converte de maneira interativa um texto em som. A figura **??** mostra o clone do repositório. Logo, na imagem **??**, eu realizo modificações no arquivo README do código do projeto. E, por fim, na figura **??**, eu demonstro a realização das modificação, o stash dessas e finalmente o commit.

```
wmespindula@wmachine1:~$ git clone https://github.com/dontstoptheparty/DontStopTheParty.git
Cloning into 'DontStopTheParty'...
remote: Enumerating objects: 1131, done.
remote: Counting objects: 100% (212/212), done.
remote: Compressing objects: 100% (109/109), done.
remote: Total 1131 (delta 89), reused 152 (delta 62), pack-reused 919
Receiving objects: 100% (1131/1131), 156.34 KiB | 433.00 KiB/s, done.
Resolving deltas: 100% (377/377), done.
```

Figura 1: Clone do Repositório. Fonte: O Autor, 2021.



```
You, seconds ago | 2 authors (You and others)
1  # DontStopTheParty
2
3
4
You, seconds ago | 2 authors (You and others)
5  ## Description
6
7  This project, created for the INFO1120 course - Programs Construction
   Thecniques - aims to create and easy an intuitive way of playing music
   using as entry music notes such as A, B, C, ...
8
9  <p align="center">
10     <img src="imgs/mainui.png"> <br>
11     DontStopTheParty Main UI      You, seconds ago • Uncommitted changes
12 </p>
13
You, seconds ago | 2 authors (Wellington M. Espindula and others)
14 ## Usage
15
16 Just open the ".jar" release with JRE
17
18 In linux bash,
19
20 ```
21 java --jar dontstoptheparty-SNAPSHOT-1.0.2-jar-with-dependencies.jar
22 ```
23
You, seconds ago | 2 authors (You and others)
24 ## Authors
25
26 - Guilherme Santana
27 - João Pedro Silveira
28 - Renan Magagnin
29 - Wellington M. Espindula.
30
You, seconds ago | 2 authors (You and others)
31 ### Documentation
```

Figura 2: Alterações no README.md. Fonte: O Autor, 2021.

```
wmespindula@wmachine1:~/DontStopTheParty$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        imgs/

no changes added to commit (use "git add" and/or "git commit -a")
wmespindula@wmachine1:~/DontStopTheParty$ git add *
wmespindula@wmachine1:~/DontStopTheParty$ git commit -m "Update README: Add Main UI image"
[master 0e8ca98] Update README: Add Main UI image
 2 files changed, 19 insertions(+), 7 deletions(-)
 create mode 100644 imgs/mainui.png
```

Figura 3: Status, stash e commit das alterações. Fonte: O Autor, 2021.

2. O que é débito técnico? Quais as consequências de não considerar o gerenciamento de débito técnico em um projeto de software?

Resposta: Muitas vezes no decorrer de um projeto, por diversos motivos - tais como pressão do cronograma ou falta de conhecimento -, soluções precárias - ou soluções não ótimas - acabam sendo entregues. Quando isso acontece, é denominado débito técnico, dado que é como se o projeto debitasse por um tempo aquele "problema" - solução pouco adequada e afins. Dessa forma, quanto mais tempo essa solução permanece, maior o valor a ser pago para que isso seja resolvido (juros). O exemplo mais claro é quando se realizam implementações ruins em código, como trechos de código pouco legíveis, código duplicado, code smells, entre outros. Mas de forma similar o débito técnico pode ser encontrado em diversos processos do projeto e desenvolvimento de software, existindo também o "Débito de Projeto e Arquitetura", "Débito de Documentação"e "Débito de Testes".
Por conseguinte, sabendo que existe um juros ligado ao débito técnico, a falta de gestão deste débito pode levar até o projeto à "falência", dado que é necessário refatorar constantemente e pagar esse débito o mais rápido possível. Como desenvolvedor, cotidianamente, noto no projeto que participo alguns débitos técnicos de anos e como isso tem prejudicado o meu processo de desenvolvimento do software, dado que eventualmente tenho que gastar muito tempo de trabalho redesenhando essas soluções que foram pouco refatoradas - muitas vezes só foram aglutinadas outras soluções - para ser possível dar prosseguimento ao processo de desenvolvimento. Portanto, sabendo que existe um débito técnico no processo de desenvolvimento, faz-se necessário tentar gerir o mesmo a fim de reduzir os danos gerados por estes.

3. **Escolha um padrão de projeto GoF não apresentado em aula. Escreva um pequeno código que ilustre o seu uso. Faça o diagrama de classes e de sequência que represente este código e coloque no relatório.**

Resposta: Quando se trata de realizar ações, um padrão muito útil é o *Command*, também conhecido por *Action* ou *Transaction*. A principal ideia deste padrão é encapsular uma solicitação em um objeto. Isso torna possível parametrizar diferentes solicitações, enfileirar ou registrar (*logging*), além de suportar o desfazer das operações e a estruturação um sistema sobre operações de alto nível construídas em cima de operações primitivas. Para título de exemplo, nas figuras **??** e **??**, utilizei o padrão *Command* para comandos de Banco de Dados. Para tanto, os comandos de banco de dados irão realizar chamadas assíncronas. Dessa forma, na figura **??**, pode-se notar que um comando de banco de dados (*DbCommand*) seria composto, de forma simples, de um comando de execução (*execute()*) e de um comando de desafazer (*rollback()*).
Para exemplificar o uso deste padrão, eu criei uma gerenciadora do fluxo de transações em Banco de Dados (*DatabaseAdministrator*) responsável por gerenciar uma fila de solicitações de Comandos. Na classe *Main*, simulando um cliente qualquer, eu consumi o *DatabaseAdministrator*. Da mesma forma que utilizei esse administrador de banco de dados para o exemplo do uso do *Command*, eu poderia ter utilizado classes *DAO* fazendo uso desses comandos.
Desta forma, o Gerenciado de Banco consegue utilizar os Comandos encapsulados como objetos, salvando um histórico de usos e possibilitando o rollback quando necessário. Ademais, o padrão *Command* permite que um comando tenha uma vida independente da solicitação original. Pensando nisso, tentei exemplificar tendo em vista que os comandos de banco de dados podem ser assíncronos e o *DbCommandStateChangedReceiver* irá informar quando a solicitação foi concluída.
O código-fonte deste exemplo foi disponibilizado em https://github.com/WellingtonEspindula/INF01127-CommandExample.

```
public class Main {

    public static void main(String[] args) {
     DatabaseAdministrator dbAdmin = new
         DatabaseAdministrator();
     dbAdmin.enqueue(new InsertionDbCommand("mTable", new
         HashMap<>(){{
         put("id", "001");
         put("name", "Wellington Espindula");
         put("telephone", "+5551000000000");
        }}));

     dbAdmin.start();

     dbAdmin.enqueue(new InsertionDbCommand("mTable", new
         HashMap<>(){{
         put("id", "002");
         put("name", "Fulano Espindula");
         put("telephone", "+5551000000000");
        }}));

     dbAdmin.enqueue(new InsertionDbCommand("mTable", new
         HashMap<>(){{
            put("id", "003");
         put("name", "Fulano Ciclano");
         put("telephone", "+5551000000000");
        }}));

     dbAdmin.enqueue(new UpdateDbCommand("mTable", "003",
         new HashMap<>(){{
         put("name", "Fulano Ciclano da Silva");
         put("telephone", "+55510000000012");
        }}));
    }
}
```

```
public class DatabaseAdministrator implements Runnable {

    private boolean isRunning;
    private boolean currentCommandIsFinished;
    private List<DbCommand> history;
    private Queue<DbCommand> operationsQueue;

    private final DbCommandStateChangedReceiver
        dbCommandStateChangedReceiver = newState -> {
        if (newState == DbCommandState.FINISHED){
            finishedCurrentCommand();
        }
    };

    public DatabaseAdministrator() {
        isRunning = false;
        history = new ArrayList<>();
        operationsQueue = new LinkedList<>();
        currentCommandIsFinished = false;
    }

    public void start() {
        isRunning = true;
        Thread newThread = new Thread(this);
        newThread.start();

    }

    public void stop() {
        isRunning = false;
    }

    private void finishedCurrentCommand() {
        currentCommandIsFinished = true;
    }

    public void enqueue(DbCommand command){
        command.setReceiver(dbCommandStateChangedReceiver);
        operationsQueue.add(command);
    }

    public List<DbCommand> getHistory(){
        return history;
    }

    @Override
    public void run() {
        while (isRunning) {
            // Supressed
        }
    }
}
```

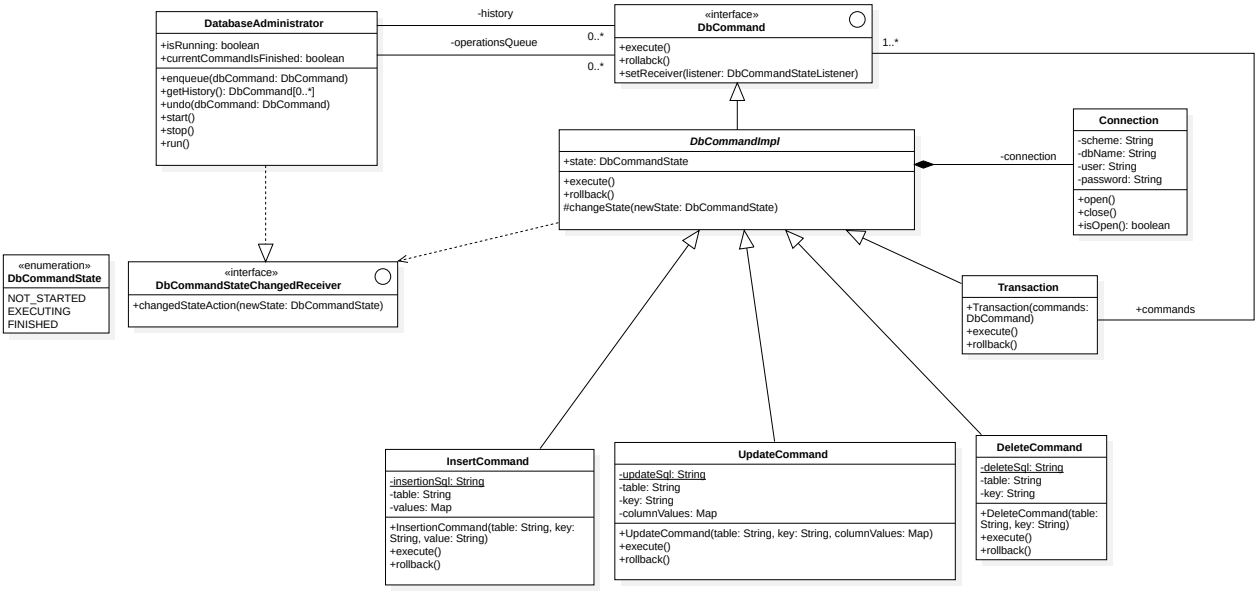


Figura 4: Exemplo de Diagrama de Classes do padrão *Command*. Fonte: O Autor, 2021.

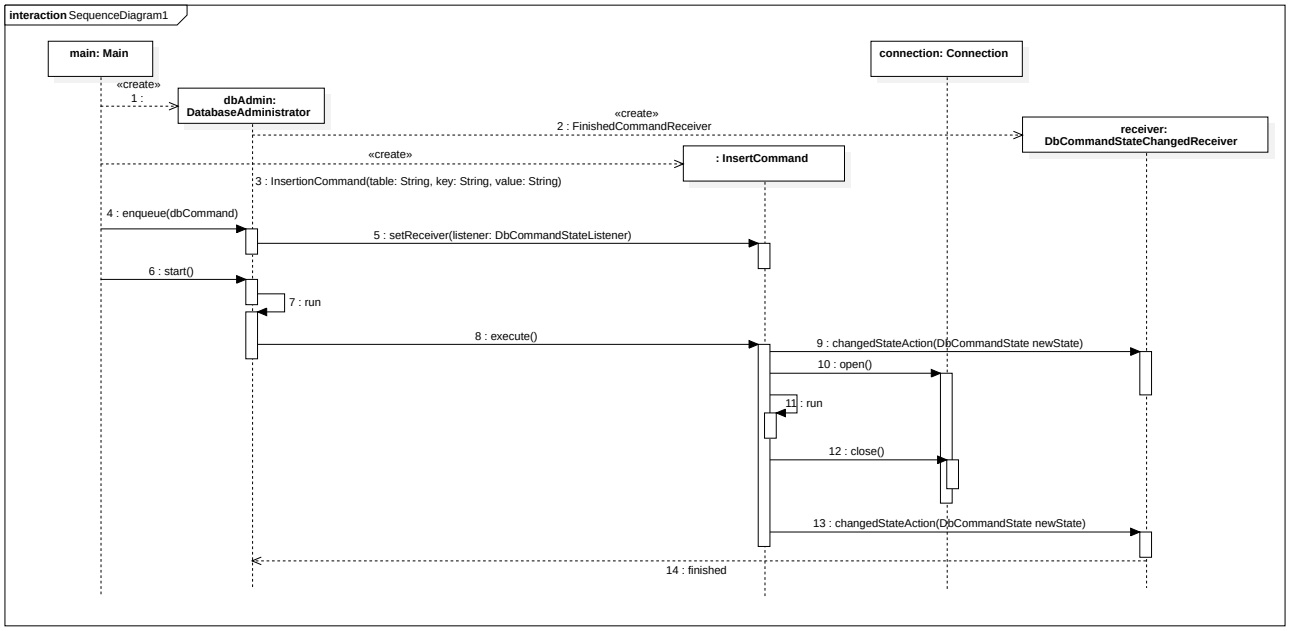


Figura 5: Exemplo de Diagrama de Sequência do padrão *Command*. Fonte: O Autor, 2021.

4. **Faça um resumo (cerca de meia página) sobre *frameworks* , desenvolvimento baseado em componentes e linhas de produto de software. Explique o que são cada uma dessas formas de reuso, como o reuso é realizado, quais suas vantagens e quais são os seus riscos.**

   Resposta: Os *Frameworks* são largamente utilizados pela indústria de software, existindo diversos tipos para diversas finalidades em diferentes linguagens de programação. Os *frameworks* fornecem um tipo de **reuso para os clientes através da inversão de controle**. Na prática, isso significa que os clientes irão estender os componentes do *frameworks* ao invés de fazer chamadas diretas - como ocorre no uso de bibliotecas -, fazendo que o *framework* retenha o fluxo de controle do software e realize as chamadas de componentes. Assim os *frameworks* geram determinados pontos de extensão (*hot-spots*) em que o cliente conecta suas partes com suas devidas particularidades. Esses pontos de extensão podem ser (*i*) **white-box**: por herança direta de uma classe abstrata; (*ii*) **black-box**: realizam-se configurações através de arquivos de configuração (xml, yaml, etc) ou por wizard; e (*iii*) **gray-box**: envolve ambas composição (herança) e configuração. Por conseguinte, os *frameworks* maximizam o reuso, a confiabilidade, diminuem a quantidade de manutenção, tornam o código mais conciso e mais consistente. Em contrapartida, a eficiência pode ficar comprometida dada a quantidade de flexibilidade e generalizações. A nível de desenvolvimento do *framework*, o seu próprio desenvolvimento é difícil, juntamente com sua documentação a manutenção a fim de mantê-lo retro-compatível com versões anteriores.
   O Desenvolvimento Baseado em Componentes (CBSE) realiza basicamente no **reuso mais tradicional e efetivo de utilizar componentes prontos acessando-os através de suas interfaces (cada componente tem uma interface provida e requerida)**. Portanto, o desenvolvimento é baseado em componentes independentes entre si, altamente coesos, com implementação escondida e interfaces bem definidas. Para tanto, é necessário ter confiança nos componentes utilizados e não existe necessariamente uma análise de certificação destes. Assim, muitas vezes tem de se fazer uma análise de trade-offs comparando as características de componentes a fim de encontrar o que atende melhor aos requisitos esperados.
   Por fim, as linhas de produto é uma forma de desenvolvimento que faz alusão à linha de produção de um carro; sendo assim, a ideia base da linhas de produto é **realizar o desenvolvimento de software utilizando a mesma arquitetura para diversos softwares** com o mesmo propósito e alterando as *constraints* para cada um dos softwares entregues conforme as demandas dos clientes. A maior desvantagem é que existe alta demanda do tempo no início do processo de criação e desenvolvimento da linha de produtos. Em contrapartida, assim que está no mercado, reduz-se o custo de desenvolvimento e manutenção bem como o tempo para tal.

5. **Qual o *framework* que realiza a injeção de dependência no estudo de caso da Biblioteca?**

   Resposta: O *framework* que realiza a injeção de dependência no sistema Biblioteca é o *Spring*. Dadas os módulos e outras configurações declaradas nas configurações do *Spring*, ele atuará nesse sistema orquestrando os módulos de forma a instanciá-los e injetá-los como parâmetros nos objetos dependentes deste.