SmartSampa

Um estudo de caso sobre o desenvolvimento de código limpo

Ruan de Menezes Costa
Orientador: Marco Dimas Gubitoso
Universidade de São Paulo, Instituto de Matemática e Estatística

Email: ruan.costa@usp.com
Página: https://linux.ime.usp.br/~ruan/mac0499/
Github: https://github.com/ruan0408

IME-USP

Introdução

Código limpo é aquele que comunica bem suas intenções ao leitor e que minimiza a dificuldade de ser modificado. O conjunto de **princípios**, **padrões** e **boas práticas** descritos nos livros *Clean Code* [2], *Implementation Patterns* [1] e *Agile software development: principles, patterns, and practices* [3], nos ajuda a aproximar nosso código desses ideais.

Este projeto visa mostrar o uso das técnicas descritas nesses livros no desenvolvimento de um projeto real, com o objetivo de obter um software de boa qualidade, isto é, código limpo.

O projeto desenvolvido constitui-se na construção de um serviço web, apelidado de SmartSampa, que facilita a captura automática de dados do sistema de ônibus de São Paulo. A prefeitura já disponibiliza dados em tempo real e estáticos sobre o sistema, porém o meio de acesso a estes dois tipos de dados é diferente. Como muitas das informações contidas nesses dois conjuntos de dados são complementares, seria melhor que o modo de acesso fosse o mesmo.

Dados

Os dados em tempo real são disponibilizados pelo serviço web Olho Vivo e os estáticos, em um arquivo no site da SPTrans. Esses últimos aderem a especificação GTFS (General Transit Feed Specification).

Algumas das palavras chave do domínio do sistema de ônibus são:

Route: Linha de ônibus.

Trip: Viagem de uma linha de ônibus. Por exemplo, a linha 701U-10 possui duas viagens, Metrô Santana e Cidade Universitária.

Stop: Ponto de ônibus.

Shape: Desenho do traçado de uma viagem.

Boas Práticas

Nomes aparecem em todos os lugares do código. Nomeamos variáveis, métodos, classes, argumentos, pacotes, etc. A principal função deles é revelar o quê o código faz. Abaixo seguem algumas boas práticas para a escolha de nomes.

- **Domínio do problema/solução**
  Procure sempre utilizar nomes do espaço do problema ou solução. Neste projeto alguns termos do espaço do problema são Trip, Route, Shape, entre outros, e foram amplamente usados na escrita do código. Já o domínio da solução consiste em termos familiares a programadores, como nomes de padrões e algoritmos. Como exemplo, temos os nomes das classes `NullTrip` e `GtfsTripAdapter`, que remetem aos padrões *NullObject* e *Adapter*, respectivamente.
- **Evite mapeamento mental**
  O mapeamento mental ocorre quando um nome de variável ou método, por exemplo, não comunica sua função de maneira clara. Isso obriga o leitor a mapear o nome ruim à função que aquela variável/método exerce, impondo uma carga cognitiva desnecessária ao leitor. É preferível um nome longo e explicativo a um nome curto e misterioso, como as tradicionais variáveis de uma letra só.

Princípios

Princípios definem características genéricas que o código deve ter para que o o software seja mais facilmente mantido e estendido.

- **Inversão de dependência**
  Segundo esse princípio, classes de alto nível não devem depender de classes de baixo nível, mas apenas de abstrações. Já as abstrações não devem depender de detalhes de implementação, mas o inverso deve ocorrer.

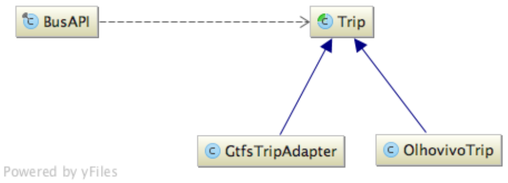

Figura 1: `BusAPI` não depende das classes de baixo nível `GtfsTripAdapter` e `OlhovivoTrip`, mas sim da abstração `Trip`.

- **Responsabilidade única**
  Esse princípio propõe que uma classe deve ter apenas uma responsabilidade. Um trecho de código ter uma responsabilidade implica em estar sujeito a mudanças caso queiramos mudar a implementação dessa responsabilidade. Classes com mais de uma responsabilidade são menos coesas e mais frágeis, pois mudanças em uma de suas atribuições podem comprometer o funcionamento das outras.
  As classes `GtfsDownloader` (que lida com o download do arquivo GTFS) e `GtfsHandler` (que lida com a localização do arquivo no sistema), surgiram devido a esse princípio.

Padrões de Design

Padrões de design são soluções prontas para problemas frequentes e costumam aderir aos princípios de código limpo.

- **Injeção de dependência**
  Uma dependência é um objeto que pode ser usado por outro. Nesse padrão, ao invés de um objeto instanciar suas dependências ele as recebe prontas. Dessa forma, a responsabilidade de criar suas dependências é retirada do objeto, o que condiz com o Princípio da Responsabilidade Única.

```
public class GtfsHandler {
    ...
    private GtfsDownloader gtfsDownloader;

    public GtfsHandler(GtfsDownloader downloader) {
        gtfsDownloader = downloader;
    }
    ...
}
```

Listing 1: A dependência `GtfsDownloader` é injetada na classe `GtfsHandler`

- **Adapter**
  O padrão *adapter* consiste na criação de um componente intermediário que traduz um componente antigo para interface desejada por um sistema novo. O sistema usa a interface nova, implementada pelo *adapter*, que delega a funcionalidade para o componente antigo. Assim, é possível usar a funcionalidade de uma classe mais velha (ou de terceiros), sem ser obrigado a usar sua interface.

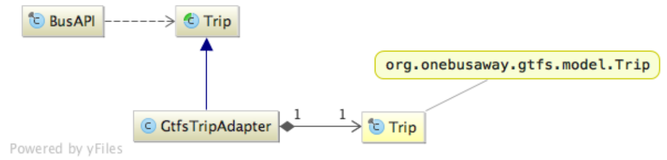

Figura 2: O padrão *adapter* permite que reutilizemos a classe `Trip` de uma outra biblioteca e implementemos a abstração `Trip` da biblioteca SmartSampa

O Serviço Web

Figura 3: Serviço web em funcionamento

Conclusão

Desenvolver código limpo é um objetivo a princípio abstrato, mas que pode ser alcançado por meio dessas e outras técnicas. Os princípios nos dão os fundamentos, que se compreendidos e seguidos, naturalmente levam à um design de classes mais limpo. Os padrões são soluções prontas para problemas comumente encontrados e que seguem os princípios. Já as boas práticas são convenções, em sua maioria estéticas, que nos ajudam a escrever código que melhor comunica suas intenções ao leitor.

Referências

[1] Kent Beck. *Implementation patterns*. Pearson Education, 2007.

[2] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[3] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.