Trabalho de Conclusão de Curso

Estudo de caso sobre o desenvolvimento de código limpo

Instituto de Matemática e Estatística
Universidade de São Paulo

Aluno: Ruan de Menezes Costa
Orientador: Prof. Dr. Marco Dimas Gubitoso

São Paulo, Novembro de 2016

Sumário

1 Introdução

1.1 Objetivos

O objetivo desta monografia é mostrar o uso das técnicas descritas nos livros *Clean Code* [9], *Implementation Patterns* [1], *Agile software development: principles, patterns, and practices* [10], entre outros, no desenvolvimento de um projeto real, com o objetivo de obter um software de boa qualidade, isto é, **código limpo**.

Muitas das técnicas e princípios descritos nos livros citados acima são explicados e exemplificados, em uma tentativa de elucidar o papel destas técnicas no desenvolvimento de software.

1.2 Motivação

Durante a graduação, não temos a necessidade de desenvolver softwares bem escritos. Os exercícios-programa (EPs) são na maioria das vezes descartáveis, dado que com o término da disciplina, ou até mesmo antes disto, nunca mais precisaremos sequer olhar para o código daquele EP. Como consequência, somos levados a não mexer mais no programa a partir do momento em que ele faz o que é pedido. Sabemos que o código está ruim, mas simplesmente não compensa gastar o tempo necessário para melhorá-lo. Por outro lado, fora da universidade, software de qualidade é uma necessidade. Funcionários vêm e vão, mas o código permanece, precisando ser mantido e incrementado.

O custo total de um programa pode ser dividido em custo inicial (desenvolvimento) e custo de manutenção [1]:

$$\text{custo}_{\text{total}} = \text{custo}_{\text{inicial}} + \text{custo}_{\text{manter}}$$

Hoje sabemos que o custo de manutenção é o fator dominante nesta equação. A manutenção é cara pois entender código já existente é uma tarefa exaustiva e passível de erros. Fazer mudanças é fácil, desde que se saiba o que mudar e que o código não seja um empecilho. Após as mudanças, ainda é necessário testar e realizar o *deploy*. Assim, chegamos ao modelo:

$$\text{custo}_{\text{manter}} = \text{custo}_{\text{entender}} + \text{custo}_{\text{mudar}} + \text{custo}_{\text{testar}} + \text{custo}_{\text{deploy}}$$

Se durante a escrita do código nos atermos às técnicas e princípios descritos nesses livros, conseguiremos minimizar os custos de entendimento, mudança e realização de testes, e assim minimizar o custo total do software.

Uma segunda motivação é que esses livros costumam exemplificar o uso dessas técnicas com exemplos de vários domínios diferentes, muitas vezes domínios caricatos, como frutas e peças de carro, o que pode passar ao leitor a impressão de que o que está ali escrito não é relevante em domínios mais realísticos. Para contornar este problema, esse projeto propõe a construção de um software para um problema real, que mostra a utilização destas técnicas dentro de um mesmo domínio.

1.3 Métodos

O projeto desenvolvido constitui-se na construção de um serviço web para facilitar a captura automática de dados do sistema de ônibus de São Paulo. A prefeitura atualmente já disponibiliza dados em tempo real e estáticos sobre o sistema de ônibus, porém o meio de acesso à estes dois tipos de dados é diferente. Como muitas das informações contidas nesses dois conjuntos de dados são complementares, seria melhor que o meio de acesso fosse o mesmo. Além disso, algumas informações estáticas que são disponibilizadas pelo serviço em tempo real não funcionam corretamente, o que pode ser consertado integrando os dois serviços. Para corrigir esses problemas foi criada uma biblioteca, apelidada de SmartSampa, que junta esses dados, mitigando o problema de informação incompleta e provê um meio de acesso unificado, facilitando o acesso à informação.

O serviço web foi desenvolvido seguindo o padrão REST [3] (*Representational State Transfer*). Dessa maneira, o usuário poderá ter acesso aos dados por meio de simples requisições HTTP cujas respostas serão em formato JSON [8]. O serviço simplesmente usa a biblioteca criada para responder as requisições. A linguagem utilizada foi Java.

1.4 Organização do texto

Na Seção 2 são discutidos os dados tratados pela biblioteca e alguns termos que serão usados no decorrer da monografia. Na Seção 3 é feito um resumo da implementação da biblioteca. Na Seção 4 são mostrados princípios, padrões de design e boas práticas que foram levados em conta durante o

desenvolvimento. Na Seção 5 é relatado o que pôde se concluir sobre desenvolvimento de código limpo. O primeiro apêndice contém diagramas e explicações mais detalhadas sobre a implementação. O segundo apêndice descreve as requisições que podem ser feitas ao serviço web.

2 Entendendo os dados

A prefeitura de São Paulo disponibiliza duas modalidades de dados sobre o transporte público da capital, que são os dados em tempo real sobre as linhas e paradas de ônibus da cidade e os dados estáticos sobre as linhas, paradas e faixas de ônibus. Os dados em tempo real são disponibilizados por meio da API (*Application Programming Interface*) Olho Vivo [11], que é um serviço web mantido pela prefeitura. Os dados estáticos sobre linhas e paradas de ônibus são disponibilizados em um arquivo .zip, que é atualizado com frequência entre 1 e 2 dias e pode ser encontrado no site da SPTrans [6]. Estes dados seguem a especificação GTFS (*General Transit Feed Specification*) [4], que é um padrão internacional criado pela Google para dados estáticos sobre transporte. Daqui em diante, estes dados serão referidos apenas por GTFS. Os dados sobre as faixas de ônibus são disponibilizados na plataforma GeoSampa [5] por meio de um arquivo no formato shapefile (.shp).

Apesar do serviço criado também fornecer o acesso aos dados sobre as faixas de ônibus, o foco do projeto é a união dos dados GTFS e Olho Vivo, pois são dados sobre os mesmos objetos (linhas e pontos de ônibus) e esta união é a parte mais interessante do ponto de vista de design de classes. Por este motivo, não serão mostrados detalhes sobre a parte referente às faixas de ônibus.

2.1 Vocabulário

O domínio do sistema de ônibus possui algumas palavras chave que foram usadas durante o desenvolvimento e serão usadas nesta monografia. Como o código foi todo escrito em inglês, também apresentaremos estas palavras em inglês, juntamente ao seu significado dentro desse domínio.

- **Route**
  Linha de ônibus.

- **Trip**
  Viagem de uma linha de ônibus. Por exemplo, a linha 701U-10 possui duas viagens, a saber, Metrô Santana e Cidade Universitária. Uma linha não tem necessariamente duas viagens, pois uma linha pode ser circular. Nesse caso a linha teria apenas uma viagem.

- **Stop**
  Ponto de ônibus.
- **Shape**
  Desenho do traçado de uma viagem.
- **Corridor**
  Corredores de ônibus.
- **Bus lane**
  Faixas de ônibus.

2.2 Dados GTFS

O padrão GTFS define uma série de arquivos que um pacote deve/pode conter para atender à especificação. Seguem os arquivos principais:

- **stops.txt**
  Lista de todas os pontos de ônibus, incluindo id, nome, latitude, longitude.
- **routes.txt**
  Lista de todas as linhas de ônibus da cidade, incluindo id (número da linha), nome da linha.
- **trips.txt**
  Lista de todas as viagens do sistema, incluindo id, dias de funcionamento, nome no letreiro.
- **stoptimes.txt**
  Lista dos horários de chegada dos ônibus de cada linha em cada um dos pontos daquela linha.
- **frequencies.txt**
  Lista com os intervalos (em segundos) de saída dos ônibus dos terminais a cada hora do dia. Por exemplo, entre às 16:00h e 17:00h, os ônibus da linha 8022-10 saem do terminal Butantã a cada 420 segundos.
- **shapes.txt**
  Lista de várias coordenadas geográficas para cada viagem. Juntas essas coordenadas desenham o traçado da viagem.

2.3 API Olho Vivo

Para acessar os dados do serviço web Olho Vivo é necessário antes se cadastrar no site da SPTrans. Após o cadastro, será recebido um token com o qual é possível se autenticar na API para então os dados poderem ser acessados. Todas as interações com a API são feitas via requisições HTTP, utilizando os verbos GET e POST em diferentes caminhos da URL principal da API, que é `http://api.olhovivo.sptrans.com.br/v0`. Todas as respostas do serviço são em formato JSON. Antes de fazer um conjunto de requisições para o serviço é necessário se autenticar. Para isto, basta fazer uma requisição POST em `http://api.olhovivo.sptrans.com.br/v0/Login/Autenticar?token=` passando o token recebido após o cadastro como parâmetro. Após a autenticação, podem ser realizadas diversas requisições de dados em um intervalo de tempo que não é conhecido por nós usuários. Após o fim desse intervalo é necessário se autenticar novamente. Abaixo seguem todas as ações permitidas pelo serviço:

- Busca de viagens de ônibus por palavra/número da linha.
- Busca de paradas de ônibus por palavra.
- Busca de paradas por id da viagem.
- Busca de paradas por corredor de ônibus.
- Busca de todos os corredores.
- Busca das coordenadas geográficas, latitude e longitude, das posições dos ônibus que estão realizando uma viagem.
- Previsão de chegada dos ônibus de uma dada linha a uma dada parada de ônibus.
- Previsão de chegada dos ônibus de uma dada linha a cada uma das paradas desta linha.
- Previsão de chegada dos ônibus de todas as linhas que atendem a uma determinada parada.

O retorno em JSON de cada uma dessas ações pode ser conferido na documentação da API [2].

3 Resumo da implementação

Como já foi dito, o propósito desse projeto é juntar as informações da API Olho Vivo, do conjunto de dados GTFS e dos dados sobre faixas de ônibus da plataforma GeoSampa. Para a implementação do serviço web foi criada uma biblioteca, apelidada de "SmartSampa", que cuida de todo o acesso aos dados. O serviço web simplesmente utiliza essa biblioteca para atender às requisições HTTP. A biblioteca é composta de alguns pacotes, dos quais os mais importantes são descritos a seguir.

3.1 Pacote `busapi`

Este pacote é responsável por usar os pacotes `olhovivoapi` e `gtfsapiwrapper` (que serão explicados mais à frente), para prover uma API unificada de acesso aos dados. Abaixo segue uma lista das principais classes seguidas de seu propósito.

- **Interfaces e classes abstratas**
  `Trip`, `Stop`, `Shape`, `BusLane`, `Bus`, `Corridor`, `PredictedBus` são os modelos dos principais objetos do sistema. Essas abstrações permitem que este pacote seja menos dependente de classes concretas.

- **Classe `BusAPI`**
  É a porta de entrada do sistema. É por meio dela que o usuário da biblioteca faz as requisições de dados.

- **Classe `DataFiller`**
  Responsável por realizar a junção (*merge*) de Trips e Stops de diferentes fontes de dados. Por exemplo, as classes `OlhovivoTrip` e `GtfsTrip` contém informações diferentes, mas ambas herdam de `Trip`. Dado um objeto de uma dessas classes, a classe `DataFiller` sabe como achar o objeto equivalente da outra classe e completar a informação.

3.2 Pacote `olhovivoapi`

Para acessar os dados da API Olho Vivo, a ideia foi que deveria haver uma classe cujos métodos realizariam requisições HTTP para a API, construiriam objetos a partir das respostas em JSON e retornariam esses objetos para quem estivesse usando a classe. Existem diversas bibliotecas para criar

objetos a partir de JSON. Optou-se por utilizar a biblioteca Jackson [7]. Por Java ser fortemente tipada, criar objetos a partir de texto não é uma tarefa tão simples quanto em outras linguagens. A biblioteca Jackson requer que sejam criadas classes para cada uma das possibilidades de retorno. Por exemplo, a resposta JSON da requisição "busque todas as viagens da linha de ônibus de número 8000", é:


```
[
    {
        "CodigoLinha": 1273,
        "Circular": false,
        "Letreiro": "8000",
        "Sentido": 1,
        "Tipo": 10,
        "DenominacaoTPTS": "PCA.RAMOS DE AZEVEDO",
        "DenominacaoTSTP": "TERMINAL LAPA",
        "Informacoes": null
    },
    {
        "CodigoLinha": 34041,
        "Circular": false,
        "Letreiro": "8000",
        "Sentido": 2,
        "Tipo": 10,
        "DenominacaoTPTS": "PCA.RAMOS DE AZEVEDO",
        "DenominacaoTSTP": "TERMINAL LAPA",
        "Informacoes": null
    }
]
```


Essa resposta corresponde a um array de objetos. Como dito, foi necessário criar uma classe que pudesse armazenar as informações contidas nessa resposta. A classe criada foi a seguinte:

```
public class OlhovivoTrip extends Trip {

    @JsonProperty("CodigoLinha") public int code;
    @JsonProperty("Circular") public boolean circular;
    @JsonProperty("Letreiro") public String numberSign;
    @JsonProperty("Sentido") public int heading;
    @JsonProperty("Tipo") public int type;
    @JsonProperty("DenominacaoTPTS") public String destinationSignMTST;
    @JsonProperty("DenominacaoTSTP") public String destinationSignSTMT;
    @JsonProperty("Informacoes") public String info;
    ...
}
```

Para cada uma das 9 ações que a API Olho Vivo fornece, foi necessário criar uma classe correspondente. Além disso, como algumas dessas respostas JSON consistiam em objetos encadeados, foram necessárias mais algumas classes.

Este pacote é formado pelas classes criadas para acomodar as respostas JSON e pela classe `OlhovivoAPI`, que realiza as requisições. Algumas dessas classes implementam (ou herdam) os modelos definidos no pacote `busapi`, o que justifica a presença desse pacote dentro da biblioteca, pois, caso contrário, esse pacote poderia constituir uma outra biblioteca que apenas seria usada pela SmartSampa.

3.3 Pacotes `gtfsapi` e `gtfsapiwrapper`

Analogamente ao pacote `olhovivoapi`, o pacote `gtfsapi` foi criado para lidar com o acesso aos dados GTFS. Por ser uma especificação conhecida, já existem bibliotecas capazes de realizar o acesso à seus dados. A biblioteca utilizada foi a onebusaway-gtfs [12]. Para regular o acesso a biblioteca, foi criada a classe `GtfsAPI`, que é a responsável por realizar consultas na base de dados GTFS. A biblioteca possui suas próprias classes, como por exemplo, `org.onebusaway.gtfs.model.Trip` e `org.onebusaway.gtfs.model.Stop`, que fazem parte do conjunto de tipos dos objetos retornados pela classe `GtfsAPI`. Essas classes não são do nosso interesse, pois não temos controle sobre elas. Por isso foi criado o pacote `gtfsapiwrapper`, cuja principal classe, `GtfsAPIFacade`, envolve os objetos retornados pela classe `GtfsAPI`

em objetos das classes `GtfsTripAdapter` e `GtfsStopAdapter`, as quais temos controle.

4 Aspectos e discussões sobre o código

Nas seções seguintes são discutidos alguns princípios, boas práticas e padrões de design considerados durante o desenvolvimento. Em cada seção é feita uma explicação de um conceito, sempre com base na bibliografia, e depois mostra-se como ele foi aplicado.

4.1 Boas práticas

4.1.1 Nomes

Nomes aparecem em todo lugar do código. Nomeamos variáveis, métodos, classes, argumentos, pacotes, etc. A principal função dos nomes é revelar o quê o código faz. Abaixo seguem algumas boas práticas a serem seguidas na hora de escolher 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`, `NullStop` e `NullCorridor`, que remetem ao padrão de design *Null Object*, e `GtfsTripAdapter`, `GtfsStopAdapter` e `GtfsShapeAdapter`, que remetem ao padrão *Adapter*.

- **Prefira informação a desinformação**
  Nomes longos e informativos devem ser preferidos a nomes curtos e não informativos/desinformativos. Por exemplo, no código abaixo retirado da classe `GtfsAPI`, fica claro que a função deste método é devolver uma lista de todas as viagens cujas linhas de ônibus contém um dado termo.

```
public List<Trip> getTripsWithRouteContainingTerm(String term) {
    Predicate<Trip> containsTerm =
        trip -> routeContainsTerm(trip.getRoute(), term);
    return filterToList("getAllTrips", containsTerm);
}
```

O mesmo princípio é valido para nomes de classes e interfaces. Evite prefixar nomes de interfaces com "I", como em `IShapeFactory` e classes concretas com "Impl", como em `ShapeFactoryImpl`, pois no melhor dos casos isso representa um excesso de informação. Como exemplo, veja a classe abstrata `Trip` e suas implementações `GtfsTripAdapter` e `OlhovivoTrip`.

- **Evite mapeamento mental**
  Ao lermos código é comum não entendermos algum trecho pois não lembramos o significado de uma variável. Nesse momento somos forçados a olhar a declaração dessa variável novamente. Esse fenômeno ocorre pois o nome da variável provavelmente não está comunicando bem a sua intenção e nos força a mapear mentalmente o nome ruim ao significado, impondo uma carga cognitiva desnecessária ao leitor. Bons nomes são auto-explicativos.

4.1.2 Métodos

Dentro do paradigma de orientação a objetos, é nos métodos que a maior parte do código se concentra e portanto é necessário escreve-los bem. Abaixo seguem alguns pontos que foram considerados nesse projeto.

- **Devem fazer apenas uma coisa**
  Um bom método deve se restringir a fazer apenas uma coisa. Funções com muitas responsabilidades deixam o leitor confuso, pois provavelmente misturam diferentes níveis de abstração. No exemplo abaixo, retirado de uma versão antiga da classe `BusAPI`, o método deve retornar todos os pontos de ônibus de um corredor:

```
public static List<Stop> getStopsFrom(Corridor corridor) {
    List<Stop> olhovivoStops =
        olhovivo.getStopsByCorridor(corridor.getId());
    return olhovivoStops.stream()
            .map(olhovivoStop -> {
                Stop gtfsStop =
                    getGtfsStopsByTerm(olhovivoStop.getName())
                        .stream()
                        .filter(olhovivoStop::equals)
                        .findAny()
                        .orElse(null);
                olhovivoStop.merge(gtfsStop);
                return olhovivoStop;
            })
            .collect(toList());
}
```

Para nos ajudar a escrever o método, podemos tentar descreve-lo usando um parágrafo "PARA":

PARA devolver todas as paradas de um corredor, precisamos achá-las na API Olho Vivo e depois mesclá-las com suas correspondentes no conjunto de dados GTFS.

Dessa maneira fica evidente que o método acima está fazendo coisas demais, pois está entrando no mérito de *como* achar um ponto de ônibus no sistema GTFS. Veja como a leitura fica melhor ao quebrarmos o método em dois:

```
static List<Stop> getStopsFromCorridor(Corridor corridor) {
    List<Stop> olhovivoStops =
        olhovivo.getStopsByCorridor(corridor.getId());
    return mergeOlhovivoStopsWithGtfsStops(olhovivoStops);
}

private static List<Stop>
mergeOlhovivoStopsWithGtfsStops(List<Stop> olhovivoStops) {
    return olhovivoStops.stream()
            .map(olhovivoStop -> {
                Stop gtfsStop =
                    gtfsAPIWrapper.getStopById(olhovivoStop.getId());
                return olhovivoStop.merge(gtfsStop);})
            .collect(toList());
}
```

4.1.3 Comentários

Ao contrário do senso comum, o uso de comentários deve ser bastante restrito. Em geral, eles escondem código que não comunica bem sua intenção e portanto necessitam prover uma ajuda extra ao leitor. Porém, essa ajuda nunca vai ser o suficiente para tornar o código comunicativo. Os comentários tendem a se tornar uma chateação, pois sempre precisamos lê-los para entender o propósito de uma variável ou de um método, quando na verdade isso deveria estar exprimido nos próprios nomes de variáveis e métodos. Assim, comentários devem ser encarados como uma falha do programador ao se comunicar. Ao escreve-los, devemos tentar reescrever o código de modo a exprimir aquilo que o comentário está tentando complementar.

Existe um caso em particular em que comentários são válidos, que é quando queremos explicar o motivo de uma decisão de implementação. Note que nesse caso, o motivo para o comentário vai além do objetivo básico do código de qualidade, que é explicar o que ele faz e como, para explicar o porquê. Como explicar a razão de si mesmo não é função do código, o comentário é bem-vindo nesse caso.

Como exemplo, podemos citar o comentário inserido no método privado `getTheOtherTrip`, da classe `GtfsAPIFacade`.

```java
/*
* This method creates a clone of this GtfsTripAdapter,
* but changes the heading and the destinationSign.
* It's necessary because the gtfs files have only one entry
* for circular trips, while the Olhovivo API
* has two entries for circular trips.
* Therefore we "falsify" one of the sides of the equivalentTrip.
* */
private GtfsTripAdapter
getTheOtherTrip(org.onebusaway.gtfs.model.Trip gtfsRawTrip) {
    return new GtfsTripAdapter(gtfsRawTrip) {
        @Override
        public Heading getHeading() {
            return Heading.reverse(super.getHeading());
        }

        @Override
        public String getDestinationSign() {
            String oldSign = super.getDestinationSign();
            String longName = gtfsRawTrip.getRoute().getLongName();

            String newSign = longName.replace(oldSign, "").trim();
            return newSign.replaceAll("^-|-$", "").trim();
        }
    };
}
```

4.1.4 Estrutura de Dados vs. Objetos

Com exceção da classe `OlhovivoAPI`, os atributos de todas as classes do pacote `olhovivoapi` foram deixados como públicos, contrariando a conhecida regra de nunca expor os atributos de uma classe. Essa decisão foi tomada pois essas classes não são classes comuns, mas sim **estruturas de dados**.

Ao aprendermos Programação Orientada a Objetos (POO), nos é ensinado que uma classe nunca deve expor seus atributos, caso contrário estaria expondo detalhes de implementação para seus usuários. Isso leva muita gente a simplesmente adicionar *getters* e *setters* na classe e achar que está tudo bem. O real motivo para não expor os atributos é que não se deve quebrar a abstração de uma classe. Dessa maneira, adicionar *getters* e *setters* também é uma violação da regra, pois ao invés dar ao usuário um modo de

manipular a **essência** dos dados, *getters* e *setters* simplesmente lidam com o dado bruto.

Seguindo a lógica inversa, classes que não possuem abstração, que são meras estruturas de dados, não precisam ter métodos de acesso a seus atributos.

É papel do programador saber quando uma classe representa um objeto que possui comportamento e abstração ou apenas uma estrutura de dados. No caso das classes listadas foi fácil, pois foram criadas exclusivamente em função da biblioteca Jackson.

4.2 Princípios

4.2.1 Princípio da Responsabilidade Única (SRP)

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. Se uma classe tem mais de uma responsabilidade, ela está sujeita a mais de um motivo para mudar e estas mudanças podem nem sempre ocorrer ao mesmo tempo. A classe então fica mais suscetível a erros, pois a mudança de implementação de uma responsabilidade pode resultar em consequências inesperadas para a outra responsabilidade. Além disso, como para cada mudança temos que recompilar, re-testar e fazer o re-deploy, é interessante minimizar o número de responsabilidades de cada classe, para assim minimizar o número execuções dessas tarefas.

Foi por causa desse princípio que surgiram as classes `GtfsDownloader` e `GtfsHandler`. No início essas classes estavam juntas, porém depois ficou claro que ali havia duas responsabilidades. O mesmo ocorreu com a classe `GtfsAPIFacade`, que é responsável por "embrulhar" os objetos da biblioteca onebusaway-gtfs em classes desse projeto. Antes, a responsabilidade disso era da classe `BusAPI`, que já possui a responsabilidade de ser a porta de entrada da biblioteca.

4.2.2 Princípio da Inversão de Dependência (DIP)

Esse princípio pode ser descrito com as seguintes frases:

- Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.

- Abstrações não devem depender de implementação. A implementação deve depender de abstrações.

O Princípio da Inversão de Dependência é um dos pilares para uma boa arquitetura de software, focada na resolução do problema e flexível quanto a detalhes de implementação, como bancos de dados, serviços web, leitura/escrita de arquivos, etc.

Esse princípio reforça que a abstração está mais relacionada ao seu cliente do que ao servidor (a classe que realiza a abstração). A ideia é que a aplicação tenha um pacote de classes abstratas e interfaces que definam grande parte do comportamento do programa. Se o comportamento é definido de forma abstrata, a implementação pode variar livremente, contanto que continue atendendo às exigências da abstração. Desse modo, inverte-se a dinâmica natural de que os módulos de alto nível são dependentes de módulos de baixo nível. Um exemplo clássico, descrito em [10], é o seguinte: considere um botão que quando apertado, acende uma lâmpada. O design que imediatamente surge a cabeça é

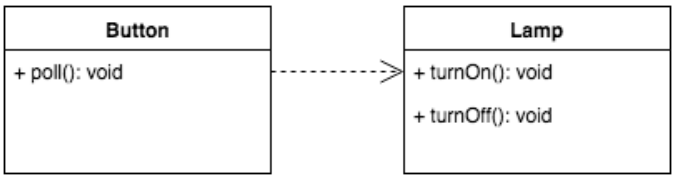


Figura 1: *Button depende de Lamp*

Fonte: *Agile software development: principles, patterns, and practices* [10]

Veja que nesse design `Button` depende de `Lamp`. Não seria possível usar esta mesma classe botão para controlar um motor, por exemplo. Esse botão é exclusivo para lâmpadas. Agora veja o design usando DIP.

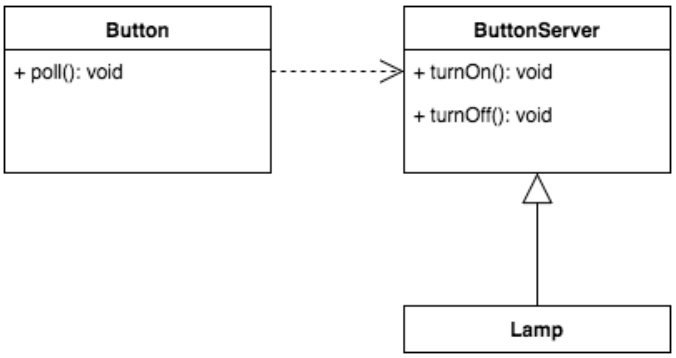


Figura 2: *Button* depende da abstração *ButtonServer*

Fonte: *Agile software development: principles, patterns, and practices* [10]

A classe `Button` não mais depende de `Lamp`, mas sim da abstração `ButtonServer`. `Lamp`, por sua vez, saiu da condição de pai na relação de dependência e virou filho, ou seja, agora ela também depende de algo, e esse algo é uma abstração. É aí que se configura a inversão de dependência.

No projeto, DIP foi bastante usado. O pacote `busapi` contém todas as abstrações e os outros pacotes contém as implementações. Ambos, em sua maior parte, usam abstrações para se comunicar. [1]

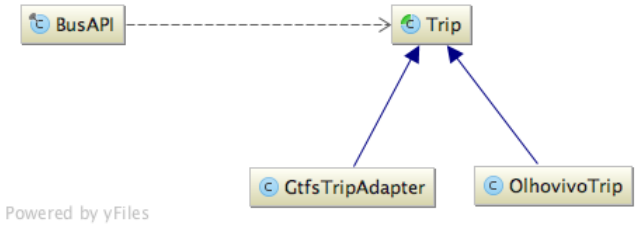


Figura 3: A classe `BusAPI`, assim como o resto do seu pacote, não utiliza as classes concretas `GtfsTripAdapter` e `OlhovivoTrip`, mas sim a abstração `Trip`.

4.2.3 Princípio do aberto-fechado (OCP)

A seguinte frase define o princípio:

Entidades de software (classes, módulos, funções, etc) devem ser abertas

[1] Algumas classes concretas foram deixadas no pacote de abstrações pois pareceu desnecessária a criação de abstrações para tudo.

para extensão mas fechadas para modificações.

Módulos que seguem esse princípio tem duas principais características:

- **Aberta para extensão**
  Isso significa que o comportamento do módulo pode ser estendido. Conforme os requisitos da aplicação mudam, nós somos capazes de adicionar comportamento de modo a satisfazer os novos requisitos.

- **Fechado para modificações**
  Adicionar comportamento ao módulo não implica em mudanças em seu código fonte.

Ao primeiro olhar essas características parecem ser contraditórias. Para estender o comportamento de uma classe, geralmente adicionamos um método em seu código. A chave aqui, assim como no princípio DIP, é a abstração. Uma aplicação feita com base em abstrações pode ter sua implementação mudada sem maiores problemas. Podemos reimplementar uma interface ou estender uma classe abstrata quantas vezes quisermos, mudando o comportamento de infinitas maneiras. Para utilizar o comportamento novo, basta substituir a implementação. Se a coisa for bem feita, é provável que nem seja necessário recompilar o módulo cujo comportamento foi alterado.

Sobre abstrações, podemos levantar dois questionamentos:

1. Será que por trás de todo conceito sempre existe uma abstração natural, fechada para qualquer tipo mudança?

2. Quando devemos procurar por abstrações? Devemos abstrair tudo?

Para a primeira pergunta, a resposta é não. Um objeto do domínio não é algo isolado, mas sim com contexto. Usando o exemplo canônico de formas geométricas, imagine que em um primeiro momento gostaríamos de desenhar uma lista dessas formas na tela. Poderíamos ter uma interface chamada `Forma`, com o método `desenha()` que é implementado pelas classes `Circulo` e `Quadrado`. Para desenhar a lista de formas, simplesmente percorreríamos a lista chamando o método `desenha()`. Caso quiséssemos adicionar a forma `Triangulo`, bastaria criar a classe e implementar a interface. Não seria necessário mudar nada no código que percorre a lista e desenha cada forma. Assim, poderíamos dizer que a classe `Forma` está fechada para mudanças em relação aos tipos de forma. Porém, imagine agora que os requisitos mudem,

e agora queiramos desenhar quadrados antes de triângulos ou círculos antes de quadrados. O nosso design não está fechado para isso, seria necessário percorrer a lista múltiplas vezes para obedecer à essas regras. Para ajustar o design, precisaríamos incorporar à ele alguma noção de ordem. O ponto aqui é que a abstração não está ligada à um objeto do domínio, mas sim ao contexto de uso desse objeto. Como um objeto pode ser usado em múltiplos contextos e não existe uma abstração natural à todos os contextos, chegamos a conclusão assustadora de que não é possível proteger o seu design de toda e qualquer mudança, o que nos leva ao segundo item.

Durante o desenvolvimento devemos procurar pelos **eixos de mudança**, isto é, aquilo que provavelmente terá variações. É justamente nesses eixos que devemos construir abstrações, para assim nos protegermos de futuras mudanças. Identificar os eixos é a grande dificuldade. O desenvolvedor precisa conhecer bem os usuários e a indústria, para assim proteger os focos mais prováveis de mudança.

Abstrair em excesso também não é interessante, já que abstrações aumentam o custo do modelo, pois aumentam a complexidade. A última coisa que queremos é ter de manter abstrações que não são utilizadas. Por isso, a abordagem mais recomendada é abstrair tudo aquilo que temos quase certeza que sofrerá mudanças e deixar para abstrair outros pontos quando os requisitos de fato mudarem. Desta maneira, sofremos um pouco quando uma mudança não antecipada ocorre mas não sofremos por antecedência com abstrações desnecessárias.

Pela natureza acadêmica e com "prazo de validade" desse projeto, a mudança com o tempo não é problema. Por isso o uso de abstrações com o propósito de seguir o princípio OCP foi limitado. O princípio foi seguindo na classe `DataFiller`, que é responsável por completar os dados que faltam a um dado objeto `Trip` ou `Stop`. A primeira implementação consistia em manualmente verificar cada atributo e, caso fosse nulo, preenche-lo com o valor do atributo correspondente de uma outra fonte de dados.

```
private Trip mergeTrips(Trip thisTrip, Trip thatTrip) {
    if (thisTrip == null && thatTrip == null) return Trip.emptyTrip();
    if (thisTrip == null) return thatTrip;
    if (thatTrip == null) return thisTrip;

    if (thisTrip.getDestinationSign() == null)
        thisTrip.setDestinationSign(thatTrip.getDestinationSign());
    if (thisTrip.getFarePrice() == null)
        thisTrip.setFarePrice(thatTrip.getFarePrice());
    if (thisTrip.getGtfsId() == null)
        thisTrip.setGtfsId(thatTrip.getGtfsId());
    if (thisTrip.getHeading() == null)
        thisTrip.setHeading(thatTrip.getHeading());
    if (thisTrip.getNumberSign() == null)
        thisTrip.setNumberSign(thatTrip.getNumberSign());
    if (thisTrip.getOlhovivoId() == null)
        thisTrip.setOlhovivoId(thatTrip.getOlhovivoId());
    if (thisTrip.getShape() == null)
        thisTrip.setShape(thatTrip.getShape());
    if (thisTrip.getWorkingDays() == null)
        thisTrip.setWorkingDays(thatTrip.getWorkingDays());
    if (thisTrip.isCircular() == null)
        thisTrip.setCircular(thatTrip.isCircular());

    return thisTrip;
}
```

Note que esse design não é fechado para a adição de atributos nas classes `Stop` e `Trip`. Se um atributo fosse adicionado em alguma dessas classes, precisaríamos modificar o método `merge` correspondente para refletir a mudança. Como o surgimento de novos atributos é a parte mais suscetível a mudança de todo o projeto, resolveu-se aplicar OCP nesse ponto. Isso foi feito usando a biblioteca de reflexão de java para descobrir os atributos das classes dinamicamente. Além disso, o método ficou genérico o suficiente para ser usado tanto para a classe `Stop` quanto `Trip`.

```
private  <T> T merge(Class<T> tClass, T thisObject, T thatObject) {
    try {
        if (thisObject == null && thatObject == null)
            return (T) tClass.getMethod("empty" +
                tClass.getSimpleName()).invoke(null);
        if (thisObject == null) return thatObject;
        if (thatObject == null) return thisObject;

        for (Field field : tClass.getDeclaredFields()) {
            Method getterMethod =
                getGetterMethod(field.getName(), tClass);
            Method setterMethod =
                getSetterMethod(field.getName(), tClass);

            Object thisValue = getterMethod.invoke(thisObject);
            Object thatValue = getterMethod.invoke(thatObject);

            if (thisValue == null)
                setterMethod.invoke(thisObject, thatValue);
        }
    } catch (IllegalAccessException | IntrospectionException |
            InvocationTargetException | NoSuchMethodException e) {
        throw new APIConnectionException(
            "An internal unexpected error occurred", e);
    }
    return thisObject;
}
```

4.2.4 Law of Demeter

Law of Demeter (LoD) é uma heurística de design que diz que uma unidade de software deveria apenas se comunicar com unidades próximas a esta unidade dentro do design. Em outras palavras, uma unidade deveria apenas se comunicar com "amigos" e não com "desconhecidos".

Se aplicada a objetos, a heurística diz que métodos de um objeto podem apenas chamar métodos nos seguintes objetos:

- O próprio objeto.
- Argumentos do método.

- Objetos criados dentro do método.
- Atributos do próprio objeto.
- Objetos globais.

Mais especificamente, métodos não deveriam chamar métodos de objetos devolvidos por outros métodos, por exemplo, `a.getB().fazAlgo()`. A motivação para isso está em outro princípio, o ***Tell, don't ask***, que prega que devemos mandar objetos executarem tarefas e não pedir dados e nós mesmos realizarmos a tarefa. No exemplo anterior, deveríamos nos perguntar se não seria melhor que a chamada fosse `a.fazAlgo()`. Nesse sentido, a LoD serve como uma heurística para encontrar violações do princípio *Tell, don't ask*.

Note que as classes tendem a ficar menos acopladas umas as outras pois dependem menos do funcionamento interno de outros objetos. Além disso, a LoD incentiva que cada método faça pouca coisa, o que condiz com os princípios destacados na seção Métodos. Como aspecto negativo, cabe dizer que usar a LoD em exagero pode levar a criação muitas classes que fazem muito pouco.

A LoD foi amplamente respeitada no projeto. Ela só não foi respeitada quando não era possível. No método abaixo, por exemplo, não há como seguir a LoD, pois o objeto `gtfsRawTrip` é de uma classe da biblioteca onebusaway-gtfs, que não pode ser modificada.

```
public String getNumberSign() {
    return gtfsRawTrip.getRoute().getShortName();
}
```

Já como exemplo do princípio *Tell, don't ask*, podemos citar a criação do método `containsTerm(String term)`, da classe abstrata `Trip`.

```
public boolean containsTerm(String term) {
    return StringUtils.containsIgnoreCase(getDestinationSign(), term) ||
           StringUtils.containsIgnoreCase(getNumberSign(), term);
}
```

Antes desse método os próprios clientes da classe `Trip` *perguntavam* pelo `destinationSign` e `numberSign` e realizavam esta operação.

4.3 Padrões de design

4.3.1 Injeção de Dependência

Esse padrão, mais conhecido como *Dependency Injection* (DI), é uma aplicação do princípio de Inversão de Controle (IoC) para a resolução de dependências. Uma dependência é um objeto que pode ser usado por outro. Isso frequentemente aparece na forma de um atributo de uma classe. A inversão de controle move responsabilidades secundárias de um objeto para outro, que é dedicado para isto, o que condiz com o Princípio da Responsabilidade Única (SRP). No do contexto de tratamento de dependências, IoC prega que um objeto não deveria ter a responsabilidade de instanciar suas próprias dependências, mas que deveria delegar essa responsabilidade para outro objeto. Injeção de dependência implementa essa inversão de uma maneira ainda mais agressiva, onde ao invés de delegar a criação de sua dependência, o objeto recebe sua dependência pronta. O objeto dependente não toma nenhuma iniciativa a respeito de satisfazer suas dependências. Todas são dadas à ele por entidades superiores. Existem três jeitos comuns para implementar a injeção de dependência:

- **Via construtor**
  No momento da criação do objeto, suas dependências são passadas como argumento pelo construtor. Considere o seguinte exemplo:

```
class Car {

    private Wheel wheel= new NepaliRubberWheel();
    private Battery battery= new ExcideBattery();
    ...
}
```

  Note que a classe `Car` instancia suas próprias dependências, o que a torna não apenas dependente dos tipos `Wheel` e `Battery`, mas também das classes concretas `NepaliRubberWheel` e `ExcideBattery`.

O código abaixo elimina a dependência de classes concretas usando a injeção de dependência.

```
public class Car {

    private Wheel wheel;
    private Battery battery;

    public Car(Wheel wheel,Battery battery) {
        this.wheel = wheel;
        this.baterry = battery;
    }
}
```

- **Via método *setter***
  A dependência é injetada via um método *setter*. No exemplo acima, teríamos:

```
public class Car {

    private Wheel wheel;
    private Battery battery;

    public void setWheel(Wheel wheel) {
        this.wheel = wheel;
    }
    public void setBattery(Battery battery) {
        this.battery = battery;
    }
}
```

- **Via interface**
  Nessa técnica, cada dependência possui uma interface que define como ela deve ser injetada. As classes devem implementar as interfaces das dependências que possuem. No nosso exemplo, teríamos:

```
public interface InjectWheel {
    public void setWheel(Wheel wheel);
}

public interface InjectBattery {
    public void setBattery(Battery battery);
}

public class Car implements InjectWheel, InjectBattery {

    private Wheel wheel;
    private Battery battery;

    public void setWheel(Wheel wheel) {
        this.wheel = wheel;
    }
    public void setBattery(Battery battery) {
        this.battery = battery;
    }
}
```

Com isso, é possível fazer com que a própria dependência injete a si mesma no dependente. Por exemplo, para a interface `Wheel` e suas implementações teríamos:

```
public interface Wheel {

    public void inject(InjectWheel dependent);
}

public class NepaliRubberWheel implements Wheel {

    public void inject(InjectWheel dependent) {
        dependent.setWheel(this);
        // do something else
    }
}
```

Veja que agora as dependências cuidam de se autoinjetar nos depen-

dentes. Ainda é necessário algum código externo que instanciará a dependência e o dependente, e chamará o método `inject` da dependência passando o dependente como argumento. Note que esta técnica só faz sentido se dentro do método `inject` fizermos mais do que simplesmente chamar `setWheel` passando a si mesmo como argumento, como por exemplo contar o número de dependentes ou ter uma lista com todos os dependentes, caso contrário as outras técnicas fazem mais sentido.

A injeção de dependência foi utilizada nas classes `GtfsAPI` e `GtfsHandler`. Ambas as classes recebem a dependência por argumento no construtor. O padrão não é seguido a risca, pois `GtfsDownloader` é uma classe concreta, mas caso surgisse a necessidade de diferentes implementações de "downloaders", não seria difícil fazer a mudança.

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

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

```
public class GtfsAPI {
    ...
    private GtfsDao gtfsDao;

    public GtfsAPI(GtfsDao gtfsDao) {
        this.gtfsDao = gtfsDao;
    }
    ...
}
```

4.3.2 *Null Object*

É comum escrevermos código como o seguinte:

```
Employee e = DB.getEmployee("Bob");
if (e == null && e.isTimeToPay(today)
    e.pay();
```

Nós pedimos para banco de dados que nos retorne um objeto e caso o objeto não seja encontrado, `null` é devolvido. Isso nos obriga a testar se o retorno é nulo ou não, o que além de esteticamente feio, é grande foco de erro, pois frequentemente esquecemos de testar. Poderíamos lançar uma exceção, mas blocos *try/catch* são ainda mais feios que testar o retorno e ao lançar uma exceção temos de declará-las na assinatura do método, o que deve ser evitado quando possível pois essa mudança se propaga para todas as chamadas do método. O padrão *Null Object* frequentemente elimina o teste e simplifica o código. Nele, cada tipo além de ter suas implementações normais, tem uma implementação nula. Nesta implementação, os métodos não fazem nada. O exemplo acima ficaria assim:

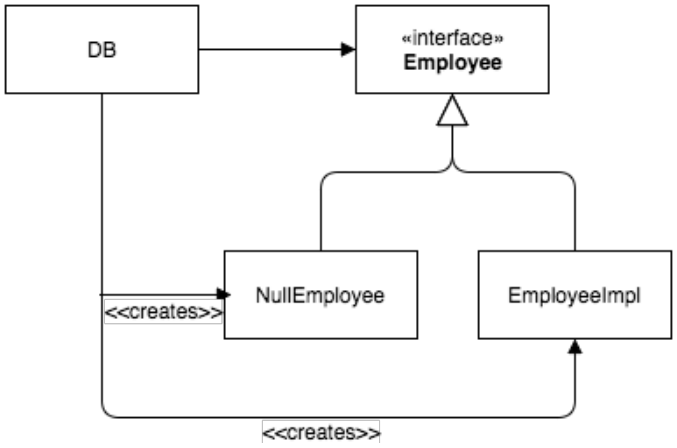


Figura 4: Padrão *Null Object*

Fonte: *Agile software development: principles, patterns, and practices* [10]

Nesse projeto, o padrão foi usado para as classes `Trip`, `Stop` e `Corridor`. No caso da classe `Stop`, a classe `NullStop` define o comportamento do

null object. `Stop` possui o método estático `emptyStop()` que retorna uma instância única do *null object*. As classes `NullTrip` e `NullCorridor` são análogas.

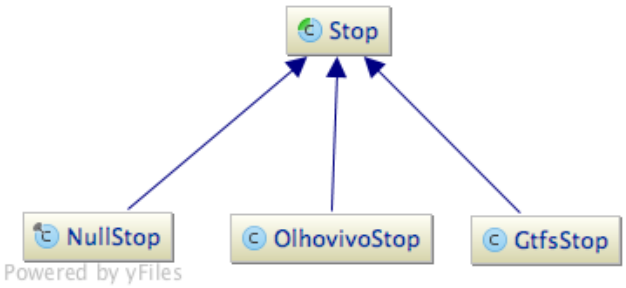


Figura 5: Hierarquia `Stop`

```
final class NullStop extends Stop {

    private static final Stop ourInstance = new NullStop();

    private NullStop() {}

    static Stop getInstance() {return ourInstance;}

    @Override
    public Integer getId() { return null; }

    @Override
    public String getName() { return null; }

    @Override
    public String getReference() { return null; }

    @Override
    public String getAddress() { return null; }

    @Override
    public Double getLatitude() { return null; }

    @Override
    public Double getLongitude() { return null; }
}
```

```
public abstract class Stop {

    ...
    public static Stop emptyStop() { return NullStop.getInstance(); }
    ...
}
```

O método abaixo da classe `GtfsAPIWrapper` retorna um `NullStop` quando não consegue achar o objeto pedido.

```
public Stop getStop(Stop equivalentStop) {
    return getAllStops().stream()
            .filter(equivalentStop::equals)
            .findAny()
            .orElse(Stop.emptyStop());
}
```

4.3.3 *Facade*

Esse padrão é usado para prover uma interface mais simples a um subsistema (múltiplos pacotes e classes) mais complexo. Ao unificar a interface desse subsistema, seu uso é facilitado e o acoplamento dos clientes com o subsistema é reduzido. Como exemplo, considere uma aplicação que use um banco de dados. Em Java, esta aplicação teria de lidar com diversas classes do pacote `java.sql`, como `Connection`, `PreparedStatement`, `SQLException`. Para reduzir o acoplamento decorrente de muitos clientes precisarem usar esse subsistema, podemos usar uma fachada (*Façade*) `DB`, que isola os usuários do banco.

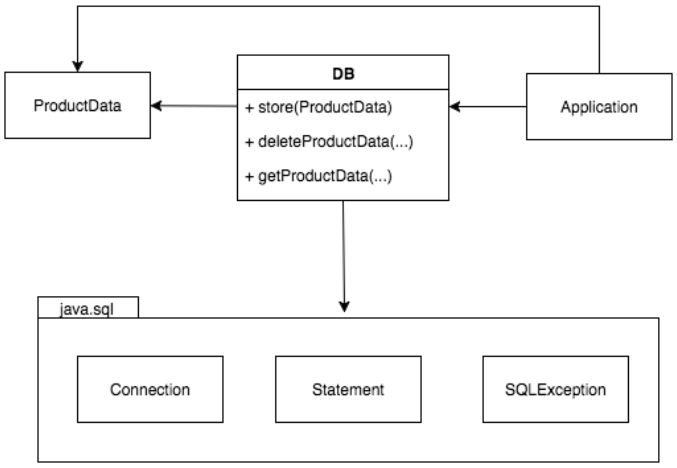


Figura 6: Padrão *facade*

Fonte: *Agile software development: principles, patterns, and practices* [10]

No projeto, o padrão foi usado na classe `GtfsAPIFacade`. Ela usa a classe `GtfsAPI` para fazer consultas à base de dados GTFS, porém isola os clientes de lidarem diretamente com os tipos de dados de-

finidos pela biblioteca onebusaway-gtfs, como por exemplo as classes `org.onebusaway.gtfs.model.Trip` e `org.onebusaway.gtfs.model.Stop`, além de sua interface ser de mais alto nível em relação a classe `GtfsAPI`.

No exemplo abaixo, a chamada `gtfsAPI.getStopsByTerm(term)` devolve um conjunto do tipo `org.onebusaway.gtfs.model.Stop`, que é transformado para um conjunto do tipo `GtfsStopAdapter`, que nada mais é do que uma implementação de `Stop`. Assim, o cliente precisa lidar apenas com a abstração `Stop`.

```
public Set<Stop> getStopsByTerm(String term) {
    return gtfsAPI.getStopsByTerm(term).stream()
            .map(GtfsStopAdapter::new)
            .collect(toSet());
}
```

4.3.4 *Singleton*

Esse padrão garante a existência de apenas um objeto de uma determinada classe. É útil quando não precisamos de múltiplas instâncias ou quando queremos forçar a existência de apenas uma, por motivos de concorrência, por exemplo. Para isso, o construtor da classe é definido como *private* e fica a cargo da classe guardar a única instância de si mesma e prover um método de acesso à essa instância.

Nesse projeto, foi usado na implementação de outro padrão, o *Null Object*. Para representar a ideia de um valor nulo para um tipo, basta termos uma única instância.

```
final class NullTrip extends Trip {

    private static final Trip ourInstance = new NullTrip();

    private NullTrip() {}

    static Trip getInstance() { return ourInstance; }

    ...
}
```

A implementação acima é a versão *eager* do padrão, onde a instância única é criada quando a classe é carregada, independentemente se alguém chamou o método `getInstance()` ou não. Na versão *lazy*, `ourInstance` seria inicializado com *null* e receberia uma instância apenas quando o método de acesso fosse chamado pela primeira vez.

4.3.5 *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*, enquanto esse delega a funcionalidade para o componente antigo. Assim, é possível usar a funcionalidade de uma classe antiga, sem ficar refém de sua interface antiga. Existem alguns motivos para não querer mudar a interface de um componente antigo. Por exemplo, talvez não tenhamos acesso ao código desse componente, o que nos impede de implementar a interface desejada, ou talvez não queiramos mexer na interface pois o uso no sistema novo não justifica tal mudança. Como exemplo, imagine que em um sistema antigo existe uma classe `Engine`, que tem os métodos `start()` e `stop()`. O cliente usa a interface `Switchable`, que define os métodos `turnOn()` e `turnOff()`. Usando um *adapter*, podemos usar um `Engine` como um `Switchable`.

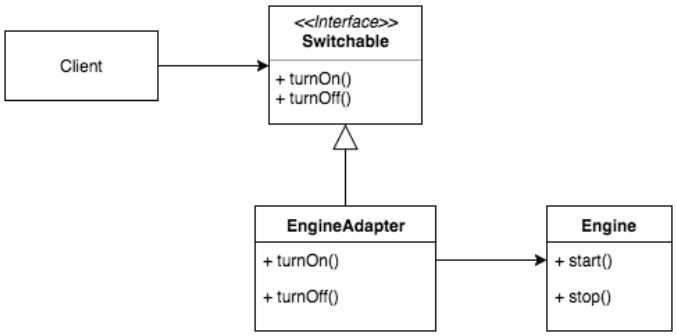


Figura 7: Padrão *adapter*

Fonte: *Agile software development: principles, patterns, and practices* [10]

Na biblioteca SmartSampa, *adapters* foram usados para podermos encaixar os tipos `org.onebusaway.gtfs.model.Trip` e `org.onebusaway.gtfs.model.Stop`, definidos pela biblioteca onebusaway-gtfs, nas abstrações definidas pelas classes abstratas `Trip` e `Stop`, do pacote `busapi`.

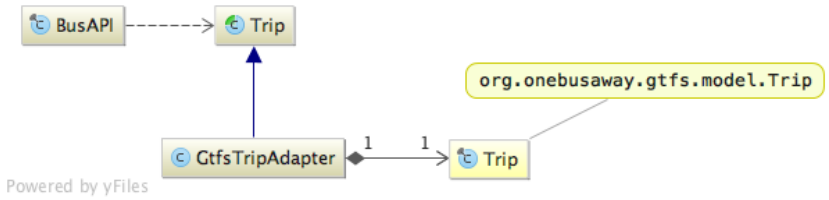


Figura 8: Uso do padrão *adapter*. As funcionalidades da classe `org.onebusaway.gtfs.model.Trip` podem ser aproveitadas devido a existência do *adapter*.

5 Conclusão

Desenvolver código limpo é um objetivo a princípio abstrato, mas que pode ser alcançado por meio das técnicas aqui descritas e de outras. Os princípios nos dão os fundamentos, que se compreendidos e seguidos, naturalmente levam a um design 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 criar código que melhor comunica suas intenções ao leitor. Além disso, esse projeto mostra que as oportunidades para usar os conceitos aqui mencionados não são exclusividade de aplicações de alta complexidade.

Referências

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

[2] *Documentação API Olho Vivo*. URL: `http://www.sptrans.com.br/desenvolvedores/APIOlhoVivo/Documentacao.aspx?1` (acesso em 18/11/2016).

[3] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". Tese de doutorado. University of California, Irvine, 2000.

[4] *General Transit Feed Specification web site*. URL: `https://developers.google.com/transit/gtfs/?hl=pt-br` (acesso em 18/11/2016).

[5] *GeoSampa website*. URL: `http://geosampa.prefeitura.sp.gov.br/PaginasPublicas/_SBC.aspx` (acesso em 18/11/2016).

[6] *GTFS SPTrans*. URL: `www.sptrans.com.br/desenvolvedores/Default.aspx` (acesso em 18/11/2016).

[7] *Jackson website*. URL: `https://github.com/FasterXML/jackson` (acesso em 18/11/2016).

[8] *JSON website*. URL: `http://www.json.org` (acesso em 18/11/2016).

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

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

[11] *Olho Vivo API website*. URL: `http://www.sptrans.com.br/desenvolvedores/APIOlhoVivo/Documentacao.aspx` (acesso em 18/11/2016).

[12] *OneBusAway website*. URL: `https://github.com/OneBusAway/onebusaway-gtfs-modules` (acesso em 18/11/2016).

Apêndice: Diagramas da implementação

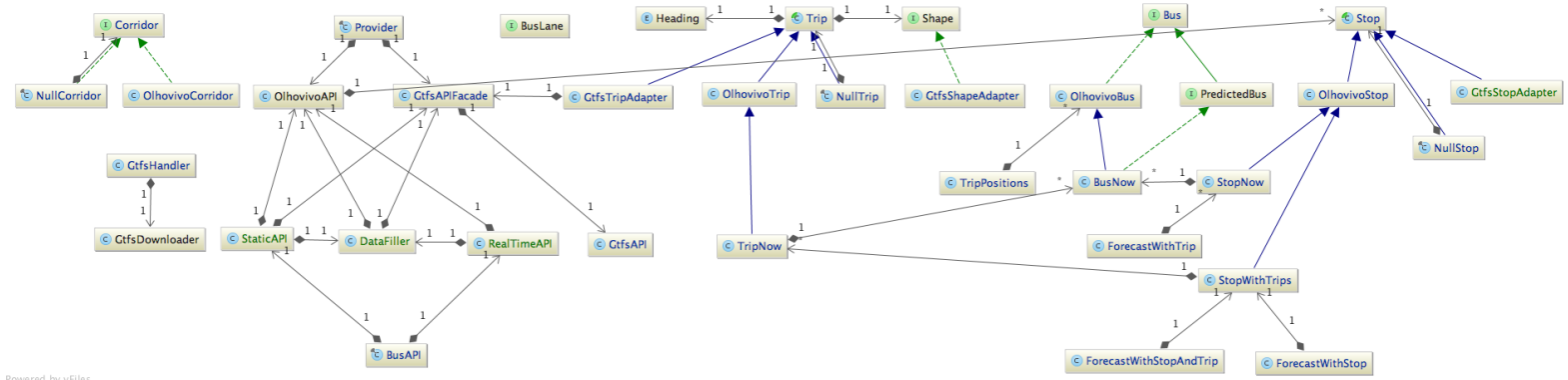


Figura 9: Visão geral da biblioteca SmartSampa. As setas roxas indicam que a classe de baixo estende a classe de cima. As setas verdes indicam que a interface de baixo estende a de cima. As setas verdes tracejadas indicam que a classe implementa a interface de cima. As outras setas indicam que a classe com o losango na ponta contém uma (1) ou várias (*) referências para objetos da outra classe.

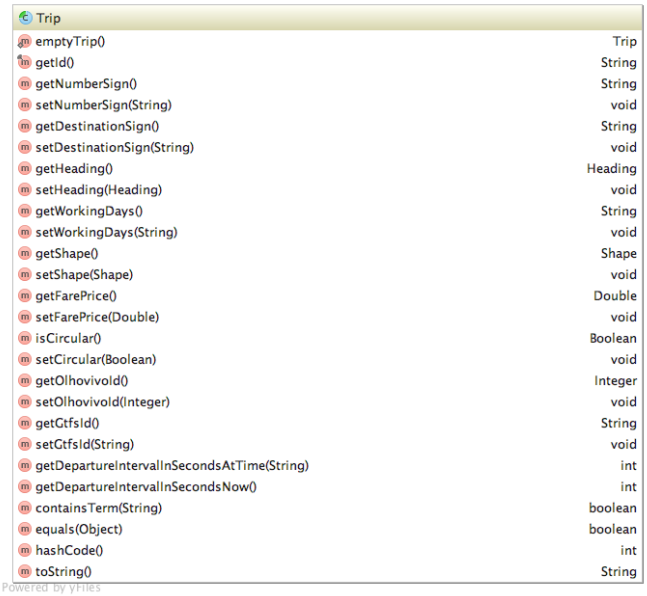


Figura 10: A classe abstrata `Trip` define as informações básicas que devem ser possíveis de se extrair de uma Trip.

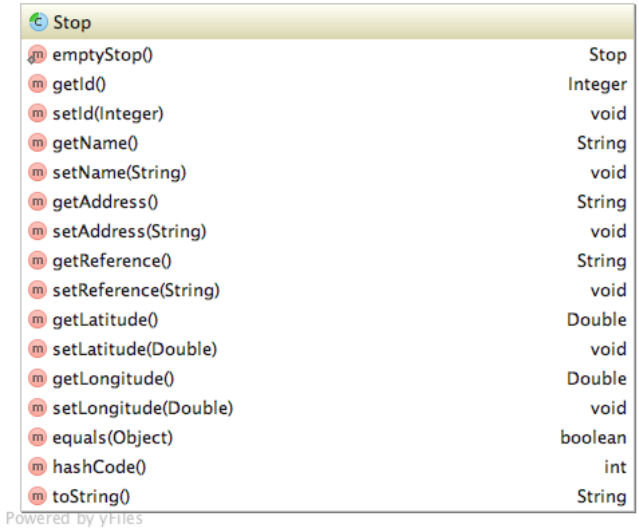


Figura 11: A classe abstrata `Stop` define as informações básicas que devem ser possíveis de se extrair de uma Stop.

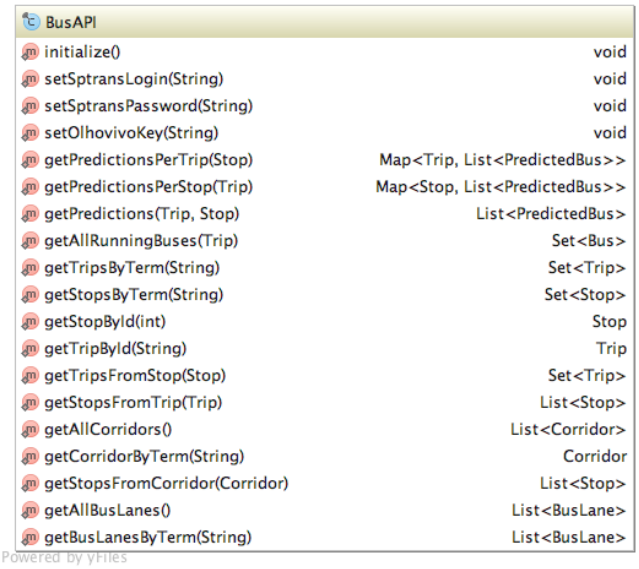


Figura 12: Porta de entrada da biblioteca. Define todas as operações possíveis de serem realizadas.

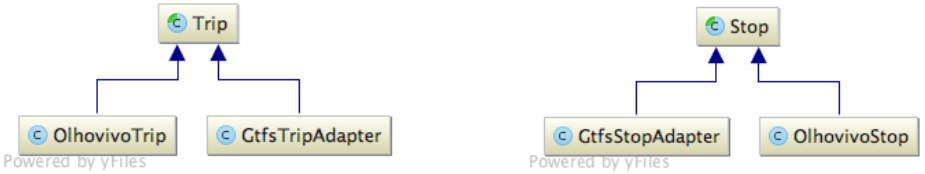
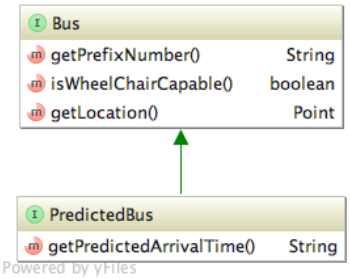
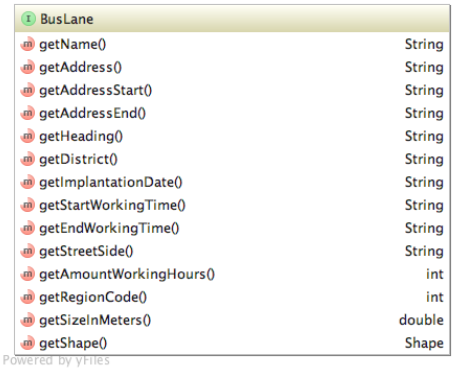


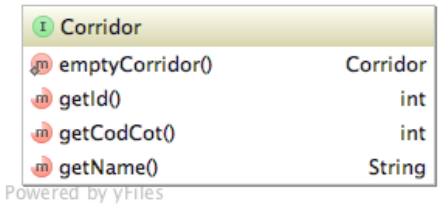
Figura 13: Implementações das classes abstratas `Trip` e `Stop`.



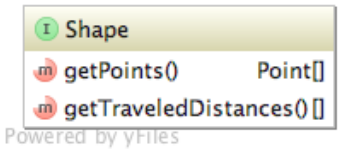
(a) Hierarquia `Bus`.



(b) Interface `BusLane`



(c) Interface `Corridor`



(d) Interface `Shape`

Figura 14: Interfaces que definem as operações dos outros objetos do sistema. Como estes objetos não estão presentes nas duas fontes de dados simultaneamente, uma interface é o suficiente para representá-los. Dado que as implementações são completas, o uso de classes abstratas é desnecessário.

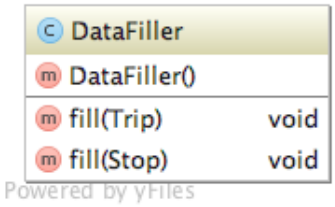


Figura 15: Classe responsável por "preencher" Trips e Stops. Ela usa as classes `OlhovivoAPI` e `GtfsAPIFacade` para achar os objetos passados como parâmetro em cada fonte de dados.

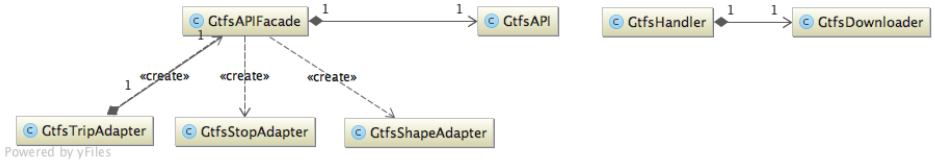


Figura 16: Pacote `gtfsapi`. `GtfsDownloader` é responsável por baixar os dados GTFS. `GtfsHandler` lida com o diretório de armazenamento dos arquivos. `GtfsAPIFacade` usa `GtfsAPI` para fazer consultas nos dados e usa os *adapters* para encaixar as classes da biblioteca onebusaway-gtfs dentro do sistema.

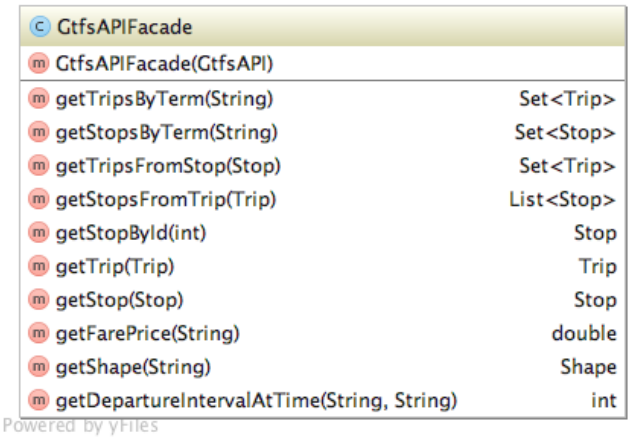


Figura 17: Classe `GtfsAPIFacade`

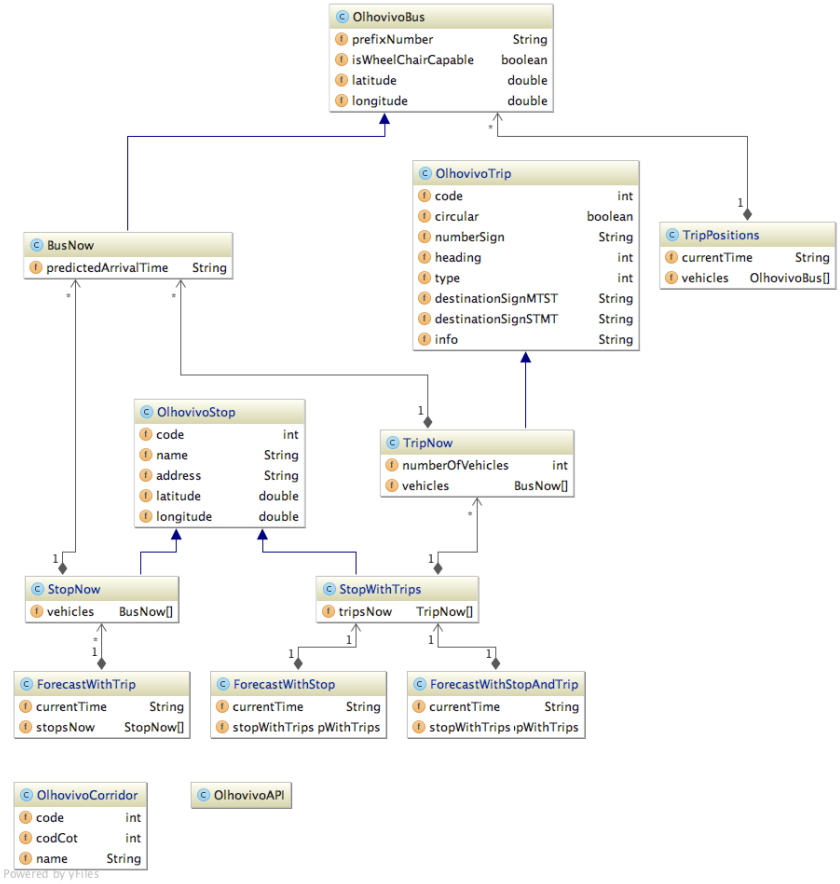


Figura 18: Pacote `olhovivoapi`. Mostrando apenas os atributos das classes. Classes criadas apenas para poder usa a biblioteca Jackson para lidar com as respostas JSON.

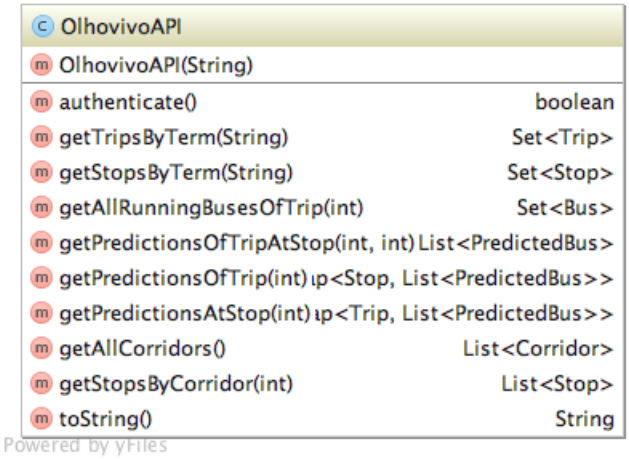


Figura 19: Métodos da classe `OlhovivoAPI`.

Apêndice: O serviço web

O serviço web está rodando no endereço `https://smartsampa.herokuapp.com`. Abaixo são listadas as diversas rotas do serviço web:

- Rotas sobre viagens:
  - `/trips/search?term={termodepesquisa}`

    Retorna informações básicas de viagens que contenham o termo pesquisado.

    Ex: `smartsampa.herokuapp.com/trips/search?term=alvim`.
  - `/trips/{tripId}`

    Retorna mais informações da viagem de id `tripId`.

    Ex: `smartsampa.herokuapp.com/trips/2732-10-1`.
  - `/trips/{tripId}/stops`

    Retorna informações sobre os pontos de ônibus de uma viagem.

    Ex: `smartsampa.herokuapp.com/trips/2732-10-2/stops`.
  - `/trips/{tripId}/predictions/buses`

    Retorna informações sobre os ônibus em circulação que estão realizando a viagem.
  - `/trips/{tripId}/predictions/stops`

    Retorna informações sobre o horário em que o próximo ônibus chegará a cada um dos pontos de ônibus da viagem indicada.
  - `/trips/{tripId}/predictions/stops/{stopId}`

    Retorna informações sobre o horário em que o próximo ônibus da viagem `tripId` chegará ao ponto cujo id é `stopId`.
- Rotas sobre pontos de ônibus:
  - `/stops/search?term={termodepesquisa}`

    Retorna informações básicas sobre os pontos de ônibus que contenham o termo pesquisado.

    Ex: `smartsampa.herokuapp.com/stops/search?term=gualberto`.
  - `/stops/{stopId}`

Retorna mais informações sobre o ponto de ônibus de id `stopId`
Ex: `smartsampa.herokuapp.com/stops/120010353`.

- `/stops/{stopId}/trips`

  Retorna as viagens que passam pelo ponto de ônibus de id `stopId`.

- `/stops/{stopId}/predictions/trips`

  Retorna o horário previsto de chegada do próximo ônibus de cada uma das viagens que passam pelo ponto de id `stopId`.

- `/stops/{stopId}/predictions/trips/{tripId}`

  Retorna informações sobre o horário em que o próximo ônibus da viagem `tripId` chegará ao ponto cujo id é `stopId`.

- Rotas sobre corredores de ônibus:

  - `/corridors`

    Retorna informações sobre todos os corredores de ônibus da cidade.

  - `/corridors/search?term={termodepesquisa}`

    Retorna informações sobre os corredores cujo nome contém o termo pesquisado.

- Rotas sobre faixas de ônibus:

  - `/buslanes`

    Retorna informações sobre todas as faixas de ônibus da cidade.

  - `/buslanes/search?term={termodepesquisa}`

    Retorna informações sobre as faixas cujo nome/endereço contenha o termo pesquisado.