Como Martin Fowler disse em TwoHardThings :

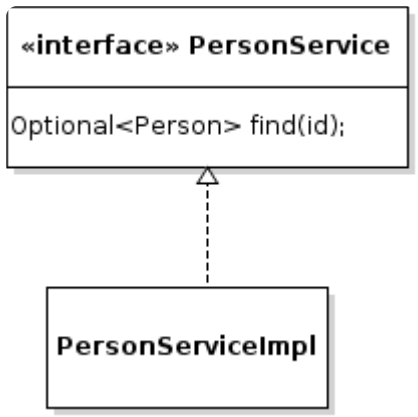
> Existem apenas duas coisas difíceis na Ciência da Computação: invalidação de cache e nomes de coisas.

Acabei de ver muitas vezes desenvolvedores usando frameworks de injeção de dependência como Spring ou Guice ou Dagger da maneira errada. Nomear classes com sufixo **Impl** é um antipadrão e vou explicar por quê.

Por que o Impl é ruim

ServiceImpl é uma prática comum

Muitos desenvolvedores, incluindo eu anos atrás, estão usando o padrão Interface + Impl para criar serviços que são injetados por sua interface. Isso geralmente se parece com isto:



Interface e sua implementação associada

Acho que muitos desenvolvedores simplesmente não entendem o ponto de injeção de dependência porque estão apenas fazendo serviços Impl. Concordo plenamente com isso: não adianta criar uma interface para um serviço que tem apenas uma única implementação conhecida. **A dissociação por uma questão de desacoplamento não tem sentido** .

Muitas coisas estão erradas com isso:

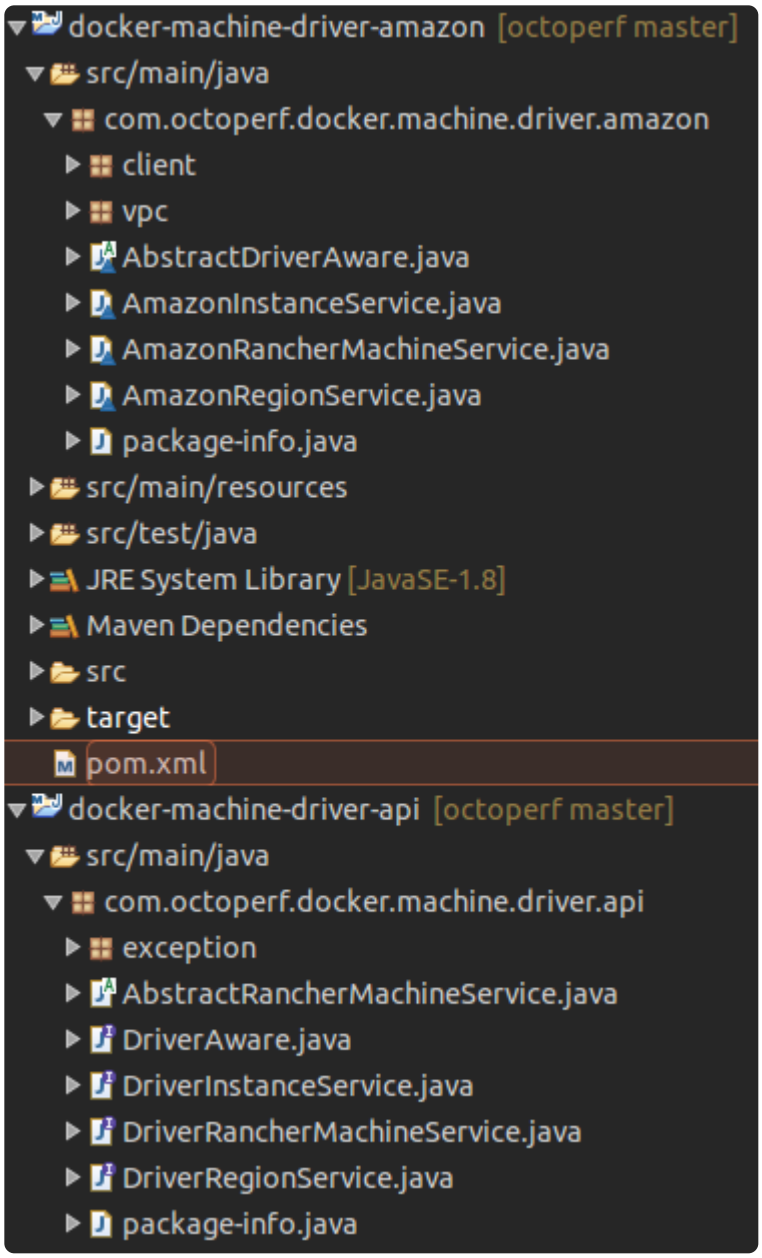
- Como o PersonServiceImpl realmente implementa o contrato de interface? Ele procura uma pessoa em um banco de dados?
- Se outra implementação da interface for necessária, ela será **PersonServiceImpl2** ?

Você começa a ver o que há de errado aqui.

Impl é ruído

Na verdade, nomear uma classe de implementação com o sufixo **Impl** é uma tautologia . Cada aula concreta é, obviamente, uma implementação de algo. É como nomear a interface de serviço **IPersonService** , o prefixo **I** traz nada mais do que ruído para o nome.

Todo o Java IDE moderno já aponta a diferença entre implementações e interfaces em sua IU.



Indicadores de tipo de Eclipse no Package Explorer

Na captura de tela acima, você pode ver que as classes de implementação dentro do módulo **docker-machine-driver-amazon** maven são protegidas por pacote. As interfaces implementadas estão localizadas dentro do módulo **docker-machine-driver-api** maven. Veremos mais tarde porque as interfaces e classes de implementação devem ser separadas umas das outras.

Bons desenvolvedores buscam ser eficientes. Ao colocar ruído desnecessário na nomenclatura de classes, o desenvolvedor perde tempo tendo que verificar o que a implementação realmente faz. Ele precisa ler a implementação do serviço para saber como funciona. Isso deve ser evitado a todo custo para preservar a produtividade dos desenvolvedores.

Impl não tem sentido

Nomear uma classe com o sufixo **Impl** é como dizer *que não sei como nomeá-la* . Se você puder encontrar um nome simples para sua classe de implementação, é um sinal de cheiro de código . Coisas que não podem ser nomeadas corretamente são muito complexas ou fazem muitas coisas, geralmente quebrando o princípio da Responsabilidade Única .

Separação de Coisas

Na maioria das vezes, a interface e a implementação estão dentro do mesmo pacote, ou pelo menos dentro do mesmo módulo maven. Claro, você tem apenas uma classe **ServiceImpl** , por que você separaria a interface da implementação?

É inútil se você vincular o contrato e a implementação . O objetivo de uma implementação é ser substituível quando se trata de injeção de dependência. Ao injetar interfaces em vez de implementações, você permite que a implementação seja substituída por outra, pois o código não depende dela.

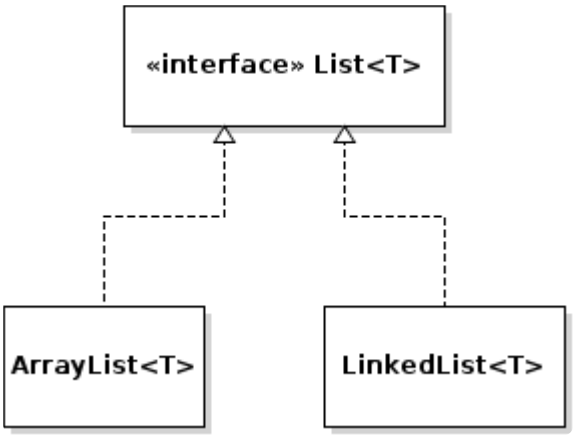
Ao enviar a interface e a implementação juntas, **você nunca poderá substituir a implementação por outra sem editar o código** . Isso é o que a maioria dos desenvolvedores não entende, porque eles acabam de lançar interfaces com o Impl a vida inteira.

A interface de transporte e sua implementação em conjunto são como jantes de transporte com pneus insubstituíveis. Quem quer isso?

O Exemplo de Lista

Vamos considerar uma das interfaces mais comuns usadas pelos desenvolvedores Java: **List** . Existem várias implementações da interface *List* :

- **ArrayList** : armazena itens dentro de uma matriz,
- **LinkedList** : armazena itens por encadeamento duplo dos elementos,
- e mais.

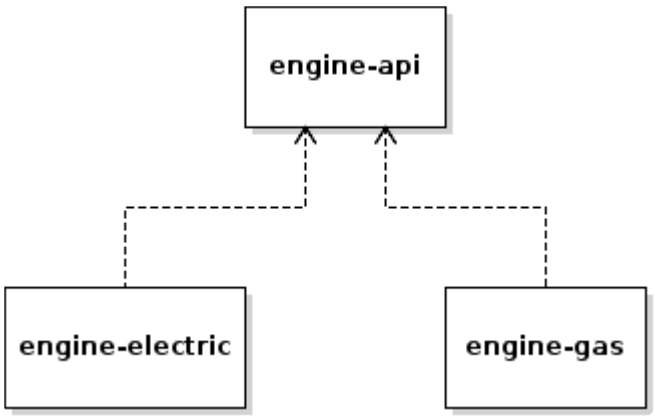


Interface Java List e algumas implementações

Você já viu um **ListImpl** dentro do JDK? Não, você não precisa. Por que você nomearia suas classes com o sufixo **Impl** então?

O Padrão API

É aqui que o padrão API vem para o resgate. Vimos que a maioria dos desenvolvedores apenas envia interfaces com seus **Impl**associados . O padrão API visa separar o contrato, ou seja, a interface e a implementação. Dessa forma, quando você precisar substituir uma implementação por outra dentro de sua aplicação, você só precisa trocar a dependência apontando para a implementação correta. **Sem mudança de código** .

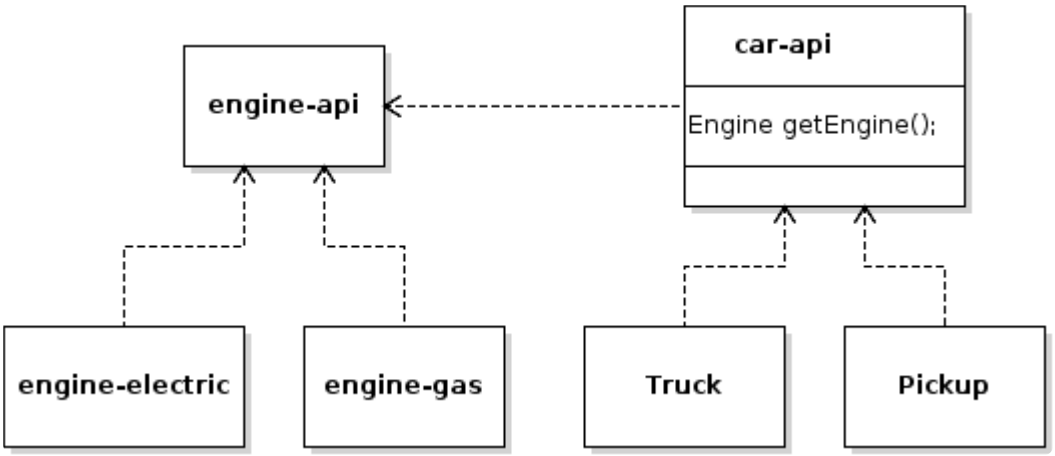


Módulos Maven da API do motor

A imagem acima mostra os módulos Maven que foram criados:

- **engine-api** : contém as interfaces que definem o contrato do motor,
- **motor-elétrico** : implementação de um motor elétrico que depende de motor-API,
- **motor a gás** : implementação de um motor a gás que depende de engine-api.

Qualquer outro módulo maven que dependa do motor depende apenas da API.



Um carro depende de um motor, sem uma suposição sobre o tipo de motor

A beleza por trás desse padrão é que o carro não se importa com o motor que foi colocado sob o teto. Cada motor fornece o mesmo método de *ignição* para iniciá-los. **Os módulos de implementação dependem apenas dos módulos API** .

O módulo de aplicativo final é responsável por conectar todas as coisas, especificando a implementação a ser usada:

```xml
<dependency>
  <groupId>com.octoperf</groupId>
  <artifactId>engine-electric</artifactId>
  <version>1.0.0</version>
</dependency>
```

Se o motor elétrico precisar ser substituído por um motor a gás, a dependência é simplesmente substituída na lista de dependências do aplicativo. **O módulo de aplicação é o único que depende de implementações concretas** .

O módulo API

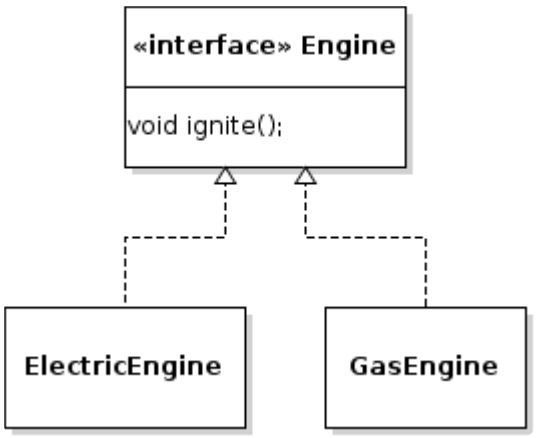
O módulo API fornece o contrato para implementar, também conhecido como interfaces em Java. Ele também fornece os testes de unidade para verificar se uma implementação segue estritamente o contrato. Nunca deve conter mais nada. Nesse caso, é o módulo **engine-api** .

Os módulos de implementação

Um módulo de implementação deve conter um tipo de implementação. É por isso que temos o módulo **motor elétrico** e o módulo **gás do motor** . Esses módulos contêm apenas classes de implementação de um determinado contrato. Esses módulos nunca devem conter qualquer contrato, pois eles já enviam implementações.

Separação de preocupações

Criei um aplicativo de exemplo completo aqui: Design por contrato . Suponha que queremos modelar um *carro* que tem um *motor* . Hoje em dia é possível encomendar um Carro com Motor a **Gasolina ou Elétrico** .



Motores elétricos e a gás implementando o contrato do motor

Não **há EngineImpl** . Seria um motor elétrico ou a gasolina? Neste exemplo, as implementações são bastante óbvias porque o código está relacionado a algo físico. **ElectricEngine** e **GasEngine** não têm nada em comum, é por isso que cada um está em um módulo maven separado.

Testabilidade de unidade

Ambos os motores podem ser testados individualmente. Melhor ainda, as classes que dependem de uma instância do *Engine também*podem ser testadas em unidades sem serem forçados a usar nenhuma das implementações acima. Pode até haver um **TestEngine**dentro de um módulo de **teste**

de motor cujo objetivo é fornecer um motor para os testes.

Muitas vezes acontece que seu teste de unidade pode exigir que algumas dependências sejam injetadas . Pode se tornar um pesadelo rapidamente injetar e iniciar o serviço corretamente se todas as dependências estiverem seguindo o padrão **Impl** . Toda a árvore de serviços será inicializada e há grandes chances de falha devido à falta de um recurso, como um banco de dados. Isso resulta em um **isolamento de teste** insatisfatório : o teste provavelmente falhará quando uma dependência também falhar. Isso também é conhecido como falha em cascata e leva ao desperdício de horas na depuração.

Por outro lado, se todos os seus módulos de implementação dependem apenas dos módulos API, você pode injetar dependências *simuladas em* vez dos serviços reais. Todos os serviços que precisam ser injetados podem ser substituídos por classes de teste, garantindo o isolamento adequado do teste. Se o teste falhar, você pode ter certeza que é porque seu serviço falhou e nada mais. Nenhum tempo é perdido depurando dependências obscuras.

```
@RunWith(MockitoJUnitRunner.class)
public abstract class AbstractClusterServiceTest {

  @Test
  public void shouldBeLeader() {
    final ClusterService service = newService();
    final boolean isLeader = service.isLeader();
    assertTrue(isLeader);
  }

  protected abstract ClusterService newService();
}
```

Exemplo de uma API que fornece um teste de unidade

E o serviço de implementação que simplesmente depende do teste de unidade do contrato e fornece a implementação:

```
public class JVMClusterServiceTest extends AbstractClusterServiceTest {
  @Override
  protected ClusterService newService() {
    return new JVMClusterService();
  }
}
```

Serviço de clustering de JVM sendo testado

Paralelizando desenvolvimentos

Um dos maiores benefícios de desacoplar contratos (interfaces) de implementações (classes concretas) **é o aumento da produtividade**:

- A API está sendo projetada com a equipe do cliente e a equipe da API,
- A equipe do cliente pode começar a trabalhar no código do cliente que usa a API, uma vez que o contrato foi definido,
- Enquanto a equipe do cliente trabalha no código do cliente, as equipes da API trabalham nas implementações desejadas.

Isso fornece um padrão altamente paralelizável que ajuda muito **a dimensionar uma equipe de desenvolvimento** . Isso reduz o tempo de espera entre as equipes, pois todos podem começar a trabalhar, mesmo que cada equipe ainda não tenha terminado sua parte. O único gargalo é a

definição do contrato de API. Pode-se até imaginar delegar parte dos desenvolvimentos terceirizando a execução dos contratos necessários. Desenvolvedores externos podem ser encarregados de fornecer uma implementação para o contrato exposto.

Conclusão

O objetivo de eliminar os maus hábitos e práticas no desenvolvimento de software é aumentar a produtividade e a velocidade do desenvolvimento. Claro, nós, como bons desenvolvedores, gostamos de trabalhar com código puro. Porque? Porque isso coloca muito menos estresse em nossa mente para trabalhar com um código bem estruturado. **Podemos nos concentrar no envio, em vez de mexer com códigos ruins e perder tempo com a depuração.**