Tarefa 2

Classe Order

Arthur Vieira de Lima Gomes

August 14, 2024

Objetivo

Implementar a classe `Order` aplicando conceitos de encapsulamento, imutabilidade e validação.

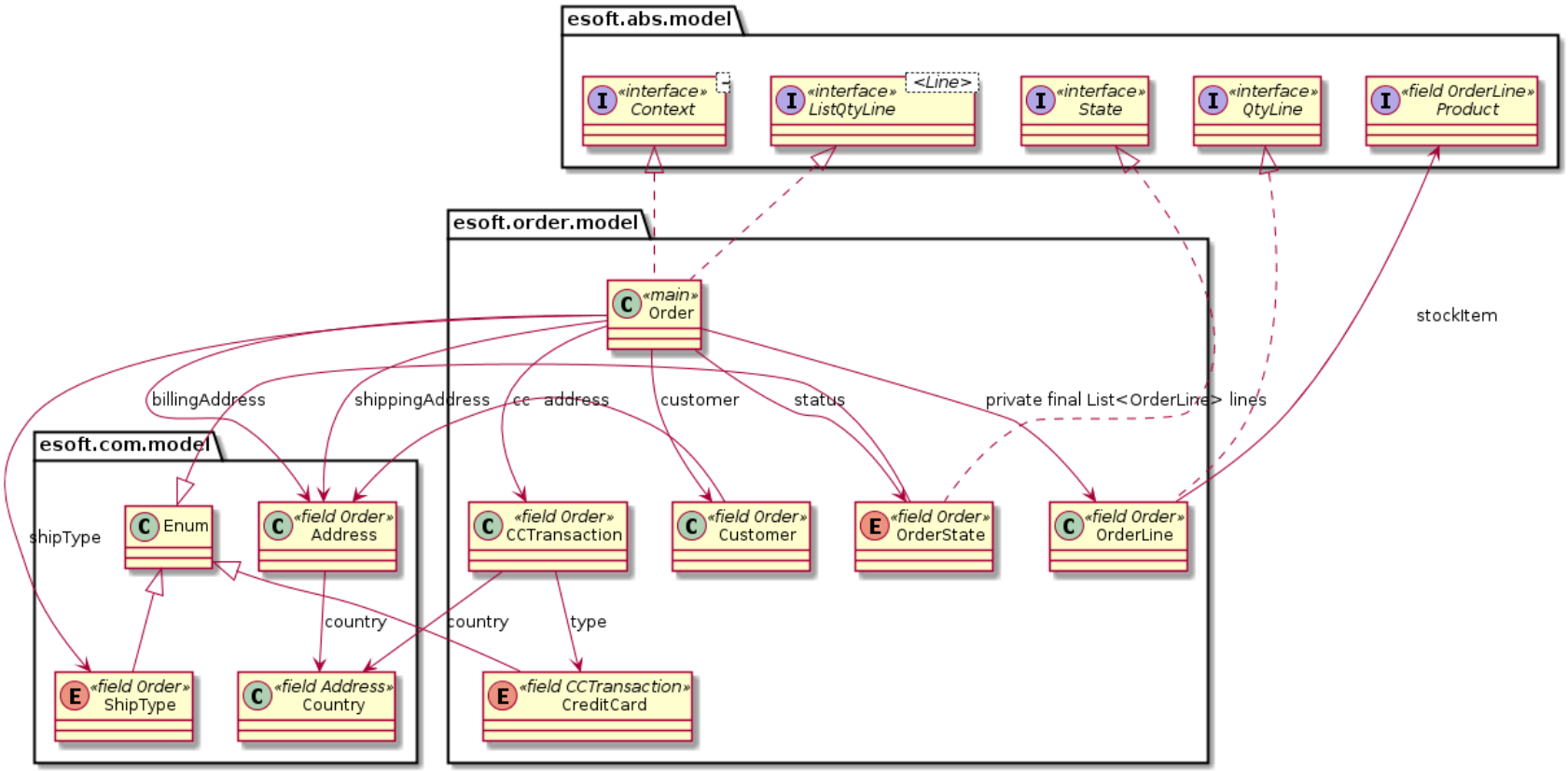
Comentário

Encapsulamento e **imutabilidade** são fundamentais para criar classes robustas e seguras. Encapsular significa proteger o estado interno de uma classe, garantindo que alterações externas não comprometam sua integridade. Imutabilidade assegura que, uma vez criado, o estado de um objeto não possa ser alterado. Esses conceitos evitam inconsistências, tornam o código mais previsível e seguro, especialmente em ambientes concorrentes.

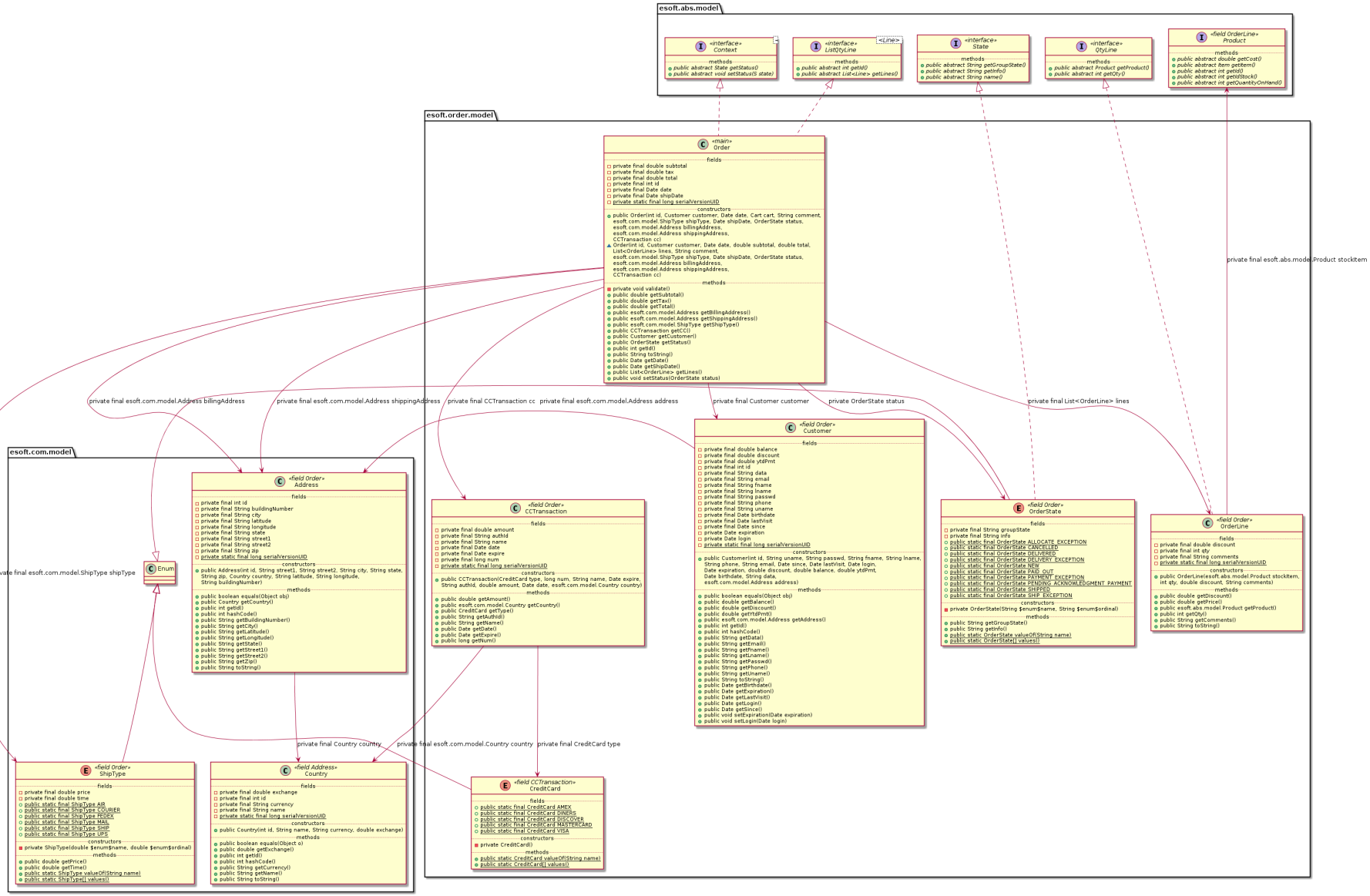
Requisitos

Então, quando a gente está criando classes, o lance não é só validar, mas também garantir que nada vá bagunçar o estado interno delas, tipo deixar tudo encapsulado. Por exemplo, a classe `Order` deve garantir a imutabilidade (com exceção do atributo "status"), ou seja, uma vez que você cria, não dá pra ficar mudando os estados dela. O único atributo de Order que pode ser modificado é o "status", que deve ser alterado usando uma máquina de estado (será discutido mais adiante).
Para criar uma Order, deve-se utilizar a classe `Cart`. No `Cart`, dá pra adicionar, remover e dar aquele tapa nos itens. Mas ó, assim que o carrinho (`Cart`) fecha e a ordem (`Order`) é criada a partir do carrinho, nada de ficar mexendo na ordem.
A ideia de deixar a classe `Order` quase imutável é uma jogada esperta pra garantir que as coisas fiquem na linha e previsíveis.

Classe Order

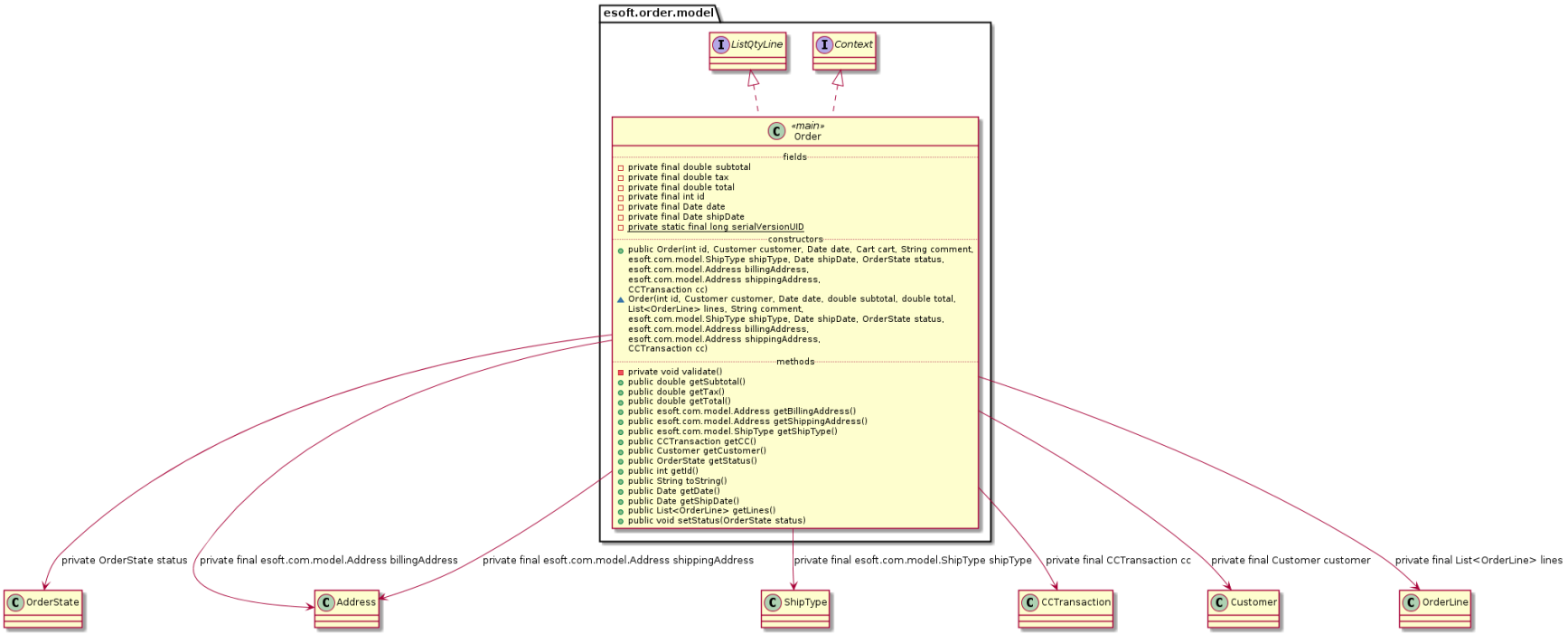


Classe Order



Classe Order

Agora, observe a nossa classe `Order`, ela está nos trinques?



Observação 1

Observe o método `getLines()` que retorna a estrutura de dados `lines`. Beleza, mas o perigo é que quem chama esse método pode fazer o que quiser com a lista. Adicionar, remover, trocar, é festa. Mas, saca, se não queremos essa baderna, tem duas saídas. Primeiro, a opção hard é substituir esse getter por vários outros métodos pra cada ação que o pessoal pode querer fazer (tipo, `getLineBySubject(Subject)` e `getLine(int)`). Isso funciona, mas é meio trabalhoso e cria um acoplamento monstrão. Vai ter que ficar mexendo toda hora na classe.
A segunda opção, que é bem mais de boa, é retornar uma espécie de 'read only' da coleção.

```
this.lines = Collections.unmodifiableList(lines);
```

Observação 2

Observe mais um pouco a classe Order, não tem setters e há “final” em praticamente todas as variáveis, parâmetros e atributos. Sacou, usar “final” é tipo a chave mágica para poder compartilhar esses objetos por um monte de threads, métodos, regras, sem dor de cabeça. Tipo, sem precisar de sincronização, sem riscos de geral inconsistências, deadlocks e outros bugs chatos. É zero custo e zero dor de cabeça, uma maravilha!

Tarefa 2 - Parte 1: Implementação da Classe

Implemente a classe `Order` conforme: requisitos, observações, diagramas de classe e descrição abaixo, aplicando as práticas de validação, encapsulamento e imutabilidade.

- **Classe** `Order`:
  - Com exceção do atributo "status", os outors atributos de Order devem ser imutáveis e devem ser criados a partir de um `Cart`.
  - Utilize uma lista imutável de itens para garantir que a `Order` não possa ser modificada após a criação.
  - Adicione validações aos atributos da `Order`:
    - Valide que `customer`, `date`, `billingAddress`, `shippingAddress` e `cc` não sejam nulos.
    - Valide que `subtotal`, `tax` e `total` sejam valores positivos.
    - Garanta que a lista de `OrderLine` não seja nula ou vazia.

Tarefa 2 - Parte 2: Testes Unitários

Implemente testes unitários para validar o comportamento da classe, assegurando que as validações e as regras de encapsulamento e imutabilidade estejam funcionando corretamente.

Nota

A nota da tarefa 2 será baseada em dois critérios principais:

- O sucesso do processo de build no Jenkins, que inclui testes e compilação.
- A porcentagem de cobertura dos testes unitários.

A cobertura de testes é avaliada usando a ferramenta **JaCoCo**. Esta ferramenta é integrada ao processo de build e gera um relatório detalhado que mostra a cobertura de testes para cada classe do projeto.
A nota será atribuída com base na porcentagem de cobertura de testes de todas as classes do pacote `esoft.order.model`, conforme indicado pelo relatório de JaCoCo.

Nota

- Igual ou acima de 95% de cobertura de testes: **nota máxima (10 pontos)**
- Igual ou acima de 90% de cobertura de testes: **9 pontos**
- Igual ou acima de 80% de cobertura de testes: **8 pontos**
- Igual ou acima de 70% de cobertura de testes: **7 pontos**
- Igual ou acima de 60% de cobertura de testes: **6 pontos**
- Igual ou acima de 50% de cobertura de testes: **5 pontos**
- Igual ou acima de 40% de cobertura de testes: **4 pontos**
- Se o código não passar nos testes: **nota 3**
- Se o código não compilar: **nota 1**
- Se o código não for alterado: **nota 0**