

refatorando o código para garantir a consistência e a integridade dos dados.

Com essa refatoração, o código ficou mais organizado, manutável e adequado aos princípios de boas práticas de programação.

Extraia a lógica de validação entre o construtor e o método `edit()` para um método privado reutilizável.

O arquivo de teste tá dando erro nessa parte:

`@Test  
@DisplayName("Set Question Test")`

O erro no seu teste `setQuestion()` ocorre porque você removeu os setters da classe `Card`. A linha `card.setQuestion(newQuestion)` não existe mais, resultando em um erro de compilação ou erro em tempo de execução (dependendo do seu ambiente de desenvolvimento).

Como você removeu os setters e adicionou o método `edit()`, você deve usar esse método para modificar a pergunta e a resposta. No entanto, se você só quer testar a alteração da pergunta, você pode criar um novo método na classe `Card` que permita isso, ou, para manter a consistência com o design atual, criar um novo card com a pergunta desejada.

Aqui estão duas soluções:

Solução 1: Usando o método `edit()` (Recomendado)

Esta é a solução mais enxilhada com o design atual da sua classe `Card`, que não expõe setters diretamente.

Java

```
public void setQuestion(String newQuestion) {
    validateInput(newQuestion, answer);
    this.question = newQuestion;
    this.answer = answer;
}
```

```
public void edit(String newQuestion, String newAnswer) {
    validateInput(newQuestion, newAnswer);
    this.question = newQuestion;
    this.answer = newAnswer;
}
```

```
private void validateInput(String question, String answer) {
    if (question == null || question.isBlank()) {
        throw new IllegalArgumentException("Question cannot be null or blank.");
    }
    if (answer == null || answer.isBlank()) {
        throw new IllegalArgumentException("Answer cannot be null or blank.");
    }
}
```

```
public boolean isCorrectAnswer(String userAnswer) {
    return Objects.equals(this.answer.trim().toLowerCase(), userAnswer.trim());
}
```

```
@Override
public String toString() {
    return "Question: " + question + "\nAnswer: " + answer;
}
```

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Card card = (Card) o;
    return Objects.equals(question, card.question) && Objects.equals(answer, card.answer);
}
```

```
@Override
public int hashCode() {
    return Objects.hash(question, answer);
}
```

A principal alteração é a criação do método privado `validateInput(String question, String answer)`. Este método agora contém toda a lógica de validação que antes estava duplicada no construtor e no método `edit()`.

Benefícios dessa refatoração:

- Remoção de Duplicação de Código: A lógica de validação agora está em um único lugar, o que torna o código mais limpo, fácil de manter e menos propenso a erros. Se a lógica de validação precisar ser alterada, ela só precisa ser modificada em um único local.

- Melhoria na Legibilidade: O construtor e o método `edit()` agora são mais concisos e fáceis de entender, pois a lógica de validação foi extraída para um método separado.

- Reusabilidade: Se em algum momento for necessário adicionar mais métodos que também precisem realizar essa validação, o método `validateInput()` poderá ser reutilizado facilmente.

- Princípio DRY (Don't Repeat Yourself): A refatoração segue o princípio DRY, evitando a repetição do mesmo código em diferentes partes da classe.

Como funciona:

1. O construtor `Card(String question, String answer)` agora chama o método `validateInput(question, answer)` antes de atribuir os valores aos atributos `question` e `answer`.

2. O método `edit(String newQuestion, String newAnswer)` também chama o método `validateInput(newQuestion, newAnswer)` antes de atualizar os valores dos atributos.

3. O método `validateInput(String question, String answer)` realiza as verificações se a pergunta e a resposta são nulas ou vazias. Se alguma das condições for verdadeira, uma `IllegalArgumentException` é lançada.

Com essa refatoração, o código ficou mais organizado, manutável e adequado aos princípios de boas práticas de programação.

O arquivo de teste tá dando erro nessa parte:

`@Test  
@DisplayName("Set Question Test")`

O erro no seu teste `setQuestion()` ocorre porque você removeu os setters da classe `Card`. A linha `card.setQuestion(newQuestion)` não existe mais, resultando em um erro de compilação ou erro em tempo de execução (dependendo do seu ambiente de desenvolvimento).

Como você removeu os setters e adicionou o método `edit()`, você deve usar esse método para modificar a pergunta e a resposta. No entanto, se você só quer testar a alteração da pergunta, você pode criar um novo método na classe `Card` que permita isso, ou, para manter a consistência com o design atual, criar um novo card com a pergunta desejada.

Aqui estão duas soluções:

Solução 1: Usando o método `edit()` (Recomendado)

Esta é a solução mais enxilhada com o design atual da sua classe `Card`, que não expõe setters diretamente.

Java

```
public void setQuestion(String newQuestion) {
    validateInput(newQuestion, answer);
    this.question = newQuestion;
}
```

```
public void edit(String newQuestion, String newAnswer) {
    validateInput(newQuestion, newAnswer);
    this.question = newQuestion;
    this.answer = newAnswer;
}
```

```
private void validateInput(String question, String answer) {
    if (question == null || question.isBlank()) {
        throw new IllegalArgumentException("Question cannot be null or blank.");
    }
    if (answer == null || answer.isBlank()) {
        throw new IllegalArgumentException("Answer cannot be null or blank.");
    }
}
```

```
public boolean isCorrectAnswer(String userAnswer) {
    return Objects.equals(this.answer.trim().toLowerCase(), userAnswer.trim());
}
```

```
@Override
public String toString() {
    return "Question: " + question + "\nAnswer: " + answer;
}
```

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Card card = (Card) o;
    return Objects.equals(question, card.question) && Objects.equals(answer, card.answer);
}
```

```
@Override
public int hashCode() {
    return Objects.hash(question, answer);
}
```

A principal alteração é a criação do método privado `validateInput(String question, String answer)`. Este método agora contém toda a lógica de validação que antes estava duplicada no construtor e no método `edit()`.

Benefícios dessa refatoração:

- Este approach maintains the immutability of the `Card` class, ensuring data consistency.

- It provides a dedicated method for modifying the question, improving code clarity.

- Tests remain unmodified, focusing on the expected behavior.

With this change in the `Card` class, your test case should now pass without modifying the test file itself.

mantenha o método `setQuestion` do arquivo original `card.java`

O arquivo de teste não deve ser modificado, faça as modificações no arquivo `card.java`:

`package org.example.studycards;`

`import java.util.Objects;`

`public class Card {`

`// ... existing code ...`

`public void updateQuestion(String newQuestion) {
 validateInput(newQuestion, this.answer); // Keep existing answer
 this.question = newQuestion;
 }`

`}`

`2. Update the test case:`

The test case remains unchanged. It continues to call the `getCard` method to retrieve the existing card and then uses the newly created `updateQuestion` method to modify the question.

Java

```
public void setQuestion(String newQuestion) {
    validateInput(newQuestion, answer);
    this.question = newQuestion;
}
```

```
public void updateQuestion(String newQuestion) {
    validateInput(newQuestion, this.answer); // Keep existing answer
    this.question = newQuestion;
}
```

```
private void validateInput(String question, String answer) {
    if (question == null || question.isBlank()) {
        throw new IllegalArgumentException("Question cannot be null or blank.");
    }
    if (answer == null || answer.isBlank()) {
        throw new IllegalArgumentException("Answer cannot be null or blank.");
    }
}
```

```
public boolean isCorrectAnswer(String userAnswer) {
    return Objects.equals(this.answer.trim().toLowerCase(), userAnswer.trim());
}
```

```
@Override
public String toString() {
    return "Question: " + question + "\nAnswer: " + answer;
}
```

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Card card = (Card) o;
    return Objects.equals(question, card.question) && Objects.equals(answer, card.answer);
}
```

```
@Override
public int hashCode() {
    return Objects.hash(question, answer);
}
```

Mudanças e explicações:

- Reintrodução de `setQuestion()` e `setAnswer()`: Os métodos foram readicionados à classe.

- Validação em `setQuestion()` e `setAnswer()`: A parte crucial é que agora o método `validateInput()` também é chamado dentro de `setQuestion()` e `setAnswer()`. Isto garante que mesmo usando os setters diretamente, a integridade dos dados seja mantida e as mesmas regras de validação aplicadas no construtor e no método `edit()` sejam respeitadas. Isso evita que a classe entre em um estado inválido. O método `validateInput` recebe a pergunta e a resposta para validar os dois parâmetros.

Por que isso é importante?

Sem a validação dentro dos setters, a classe poderia entrar em um estado inconsistente. Por exemplo, seria possível definir uma pergunta como nula ou vazia usando `setQuestion()`, mas a lógica de validação dentro do construtor ou no método `edit()` exigiria ambos os parâmetros. Isso violaria a regra definida anteriormente. Ao adicionar a validação, garantimos que todas as formas de modificar os atributos da classe (`setQuestion()` e `setAnswer()`) sigam as mesmas regras.

Com essa alteração, o teste original `setQuestion()` deve funcionar corretamente, e a classe manteém a integridade dos dados.

Sim, reintroduzir os setters (`setQuestion` e `setAnswer`) com validação dentro deles, embora resolva o problema do teste e manteña a exigência de tê-los, reintroduz um potencial code smell, embora mais suíl que o que um Data Class puro. Vamos analisar:

Problema Original (Data Class):

Um Data Class puro é um código smell porque expõe diretamente seus dados através de getters e setters sem nenhum comportamento. Isso viola o princípio do encapsulamento e torna a classe frágil e suscetível a mudanças, pois qualquer parte do código pode modificar os dados diretamente, sem nenhuma validação ou lógica de negócio.

Solução Anterior (Reintrodução dos Setters e Adição de `edit()`):

A solução anterior, que reintroduziu os setters com validação, é um compromisso. Ela atende à exigência de manter os setters, mas introduz um novo tipo de code smell: `Comportamento Anônimo com Setters`.

Comportamento Anônimo com Setters (Novo Code Smell):

Este código smell ocorre quando uma classe possui comportamento (como a validação), mas também expõe setters públicos. Isso cria uma certa ambiguidade e pode levar a um uso inconsistente da classe.

- Duplicação de Lógica (Potencial): Embora neste caso a validação esteja centralizada no método `validateInput()`, existe o risco de, no futuro, a lógica de validação se tornar mais complexa e acabar sendo duplicada entre os setters e outros métodos.

- Encapsulamento Fragilizado: Mesmo com a validação nos setters, o encapsulamento é menos forte do que a que o código puro oferece. O ideal é que a lógica de negócio (como a validação e a formatação dos dados) esteja concentrada em métodos específicos, e não espalhada pelos setters.

- Usa Inconsistente (Potencial): Os desenvolvedores podem ser tentados a usar os setters diretamente em vez do método `edit()`, o que pode dificultar o rastreamento de como os dados estão sendo modificados e dificultar a manutenção a longo prazo.

Em resumo:

A solução atual não é tão ruim quanto um Data Class puro, pois pelo menos a validação está sendo realizada. No entanto, ela introduz um novo code smell (`Comportamento Anônimo com Setters`) que indica uma possível melhoria no design.

Alternativas (Casos seja possível reconsiderar a exigência):

- Manter apenas o `edit()`: Esta seria a solução mais limpa e alinhada com os princípios do Data Class ao adicionar comportamento (validação) e encapsular a modificação dos dados.

- Criar métodos com nomes mais expressivos: Em vez de `setQuestion()`, criar um método como `changeQuestion(String newQuestion)` ou `updateQuestion(String newQuestion)` indica a intenção e seria menos propenso a erros.

Solução Atual (Reintrodução dos Setters com Validação):

A solução atual, que reintroduziu os setters com validação dentro deles, embora resolva o problema do código smell, não é a melhor opção. Vamos analisar:

Entendo. Se a exigência é manter o método `setQuestion` no `Card.java`, mesmo após a refatoração inicial que o removeu para adicionar comportamento e evitar o Data Class, podemos readicioná-lo, mas com as devidas considerações para garantir a consistência e a integridade dos dados.

Aqui está a versão modificada do `Card.java` com o `setQuestion` de volta, mas agora com validação:

Java

```
public void setQuestion(String newQuestion) {
    validateInput(newQuestion, answer);
    this.question = newQuestion;
}
```

```
public void updateQuestion(String newQuestion) {
    validateInput(newQuestion, this.answer); // Keep existing answer
    this.question = newQuestion;
}
```

```
private void validateInput(String question, String answer) {
    if (question == null || question.isBlank()) {
        throw new IllegalArgumentException("Question cannot be null or blank.");
    }
    if (answer == null || answer.isBlank()) {
        throw new IllegalArgumentException("Answer cannot be null or blank.");
    }
}
```

```
public boolean isCorrectAnswer(String userAnswer) {
    return Objects.equals(this.answer.trim().toLowerCase(), userAnswer.trim());
}
```

```
@Override
public String toString() {
    return "Question: " + question + "\nAnswer: " + answer;
}
```

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Card card = (Card) o;
    return Objects.equals(question, card.question) && Objects.equals(answer, card.answer);
}
```

```
@Override
public int hashCode() {
    return Objects.hash(question, answer);
}
```

Mudanças e explicações:

- Reintrodução de `setQuestion()` e `setAnswer()`: Os métodos foram readicionados à classe.

- Validação em `setQuestion()` e `setAnswer()`: A parte crucial é que agora o método `validateInput()` também é chamado dentro de `setQuestion()` e `setAnswer()`. Isto garante que mesmo usando os setters diretamente, a integridade dos dados seja mantida e as mesmas regras de validação aplicadas no construtor e no método `edit()` sejam respeitadas. Isso evita que a classe entre em um estado inválido. O método `validateInput` recebe a pergunta e a resposta para validar os dois parâmetros.

Por que isso é importante?

Sem a validação dentro dos setters, a classe poderia entrar em um estado inconsistente. Por exemplo, seria possível definir uma pergunta como nula ou vazia usando `setQuestion()`, mas a lógica de validação dentro do construtor ou no método `edit()` exigiria ambos os parâmetros.

Com essa alteração, o teste original `setQuestion()` deve funcionar corretamente, e a classe manteém a integridade dos dados.

Sim, reintroduzir os setters (`setQuestion` e `setAnswer`) com validação dentro deles, embora resolva o problema do teste e manteña a exigência de tê-los, reintroduz um potencial code smell, embora mais suíl que o que um Data Class puro. Vamos analisar:

Problema Original (Data Class):

Um Data Class puro é um código smell porque expõe diretamente seus dados através de getters e setters sem nenhum comportamento. Isso viola o princípio do encapsulamento e torna a classe frágil e