


## DataClass Card

(Não há do SearchLog porque quando fui refatorar um do Feature Envy, ele refatorou o do DataClass e saiu a violação do Response.json).

A solução será reformular os métodos e o design para focar em comportamentos úteis e encapsulados que fazem sentido no contexto de uma classe `Card`. Vou refatorar novamente, ajustando para que os métodos reflitam comportamentos específicos e eliminem o problema.

### Refatoração Final:

java

 Copiar código

```
public class Card {
    private String question;
    private String answer;

    public Card(String question, String answer) {
        validateInput(question, "Question");
        validateInput(answer, "Answer");
        this.question = question;
        this.answer = answer;
    }

    public String displayQuestion() {
        // Exibe a pergunta com formatação apropriada
```

```
    public String displayQuestion() {
        // Exibe a pergunta com formatação apropriada
        return "Q: " + question;
    }

    public boolean validateAnswer(String userAnswer) {
        // Compara a resposta dada pelo usuário com a resposta correta
        if (userAnswer == null) {
            return false;
        }
        return this.answer.equalsIgnoreCase(userAnswer.trim());
    }


    public void edit(String newQuestion, String newAnswer) {
        // Atualiza os campos com validação, encapsulando a lógica de edição
        validateInput(newQuestion, "New Question");
        validateInput(newAnswer, "New Answer");
        this.question = newQuestion;
        this.answer = newAnswer;
    }
```

```

private void validateInput(String input, String fieldName) {
    // Valida entradas para evitar valores nulos ou vazios
    if (input == null || input.trim().isEmpty()) {
        throw new IllegalArgumentException(fieldName + " cannot be null or empty.");
    }
}

@Override
public String toString() {
    // Retorna uma representação útil do Card
    return "Question: " + question;
}
}

```

 Copiar código

## Explicação da Refatoração:

### 1. Eliminação de Setters:

- Removemos os métodos `setQuestion` e `setAnswer`. Agora, o estado do objeto só pode ser modificado através do método `edit`, que encapsula a lógica de atualização com validação.


### 2. Métodos Comportamentais:

- `displayQuestion`: Fornece uma maneira específica e formatada de exibir a pergunta.
- `validateAnswer`: Avalia se uma resposta fornecida está correta. Esse método encapsula a comparação de strings com segurança.

### 3. Centralização de Validação:

- A validação foi movida para um método privado `validateInput`, reduzindo duplicação e melhorando a manutenção.

### 4. Coesão:

- Todos os métodos implementados têm um propósito específico, relacionado diretamente ao conceito de "Card" como parte de  sistema de perguntas e respostas.

### 3. Centralização de Validação:

- A validação foi movida para um método privado `validateInput`, reduzindo duplicação e melhorando a manutenção.

### 4. Coesão:

- Todos os métodos implementados têm um propósito específico, relacionado diretamente ao conceito de "Card" como parte de um sistema de perguntas e respostas.

### 5. Imutabilidade Controlada:

- A classe não permite modificações diretas dos campos `question` e `answer`. Todas as alterações devem passar pela lógica do método `edit`.

## Benefícios:

- **Encapsulamento:** O acesso e a modificação dos dados são totalmente controlados.
- **Validação Centralizada:** Garante que sempre haverá valores válidos para `question` e `answer`.

## DataClass Task

eu quero q faça a refatoração do DataClass com Encapsulation Behavior



Entendido! Vamos fazer a refatoração para resolver o **DataClass** com **Encapsulation Behavior**, sem introduzir outros problemas, como o **NcssCount**. A ideia é encapsular comportamento relevante e mover lógica para dentro da classe, enquanto minimizamos linhas de código extras.

### Refatoração com Encapsulation Behavior

Aqui está a implementação com foco em encapsular o comportamento, eliminando o problema de **DataClass**:

java

Copiar código

```
package org.example.studyregistry;
```

```
import java.time.LocalDateTime;
```



```
package org.example.studyregistry;

import java.time.LocalDateTime;

public class Task extends Registry {
    private String title;
    private String description;
    private String author;
    private LocalDateTime date;

    public Task(String title, String description, String author, LocalDateTime date) {
        validate(title, "Title cannot be null or empty");
        validate(author, "Author cannot be null or empty");
        validate(description, "Description cannot be null or empty");
        validateDate(date);

        this.title = title;
        this.description = description;
        this.author = author;
        this.date = date;
        this.name = title; // Registry inheritance
    }
}
```

```
// Encapsula comportamento: Atualiza título
public void updateTitle(String newTitle) {
    validate(newTitle, "Title cannot be null or empty");
    this.title = newTitle;
    this.name = newTitle;
}

// Encapsula comportamento: Atualiza descrição
public void updateDescription(String newDescription) {
    validate(newDescription, "Description cannot be null or empty");
    this.description = newDescription;
}

// Encapsula comportamento: Atualiza autor
public void updateAuthor(String newAuthor) {
    validate(newAuthor, "Author cannot be null or empty");
    this.author = newAuthor;
}


// Encapsula comportamento: Atualiza data
public void reschedule(LocalDateTime newDate) {
}
```

```
// Encapsula comportamento: Atualiza data
public void reschedule(LocalDateTime newDate) {
    validateDate(newDate);
    this.date = newDate;
}

// Método de resumo encapsulado
public String getSummary() {
    return String.format("Task: %s\nAuthor: %s\nDate: %s\nDescription: %s",
        title, author, date.toString(), description);
}

// Métodos de acesso (limitados para leitura)
public String getTitle() {
    return title;
}

public String getDescription() {
    return description;
}
```

 Copiar código




```
public String getAuthor() {
    return author;
}

public LocalDateTime getDate() {
    return date;
}

// Validações privadas para reutilização
private void validate(String value, String errorMessage) {
    if (value == null || value.isEmpty()) {
        throw new IllegalArgumentException(errorMessage);
    }
}

private void validateDate(LocalDateTime date) {
    if (date == null || date.isAfter(LocalDateTime.now())) {
        throw new IllegalArgumentException("Date must be in the past or present");
    }
}
}
```

 Copiar código



## O que foi feito?

### 1. Encapsulação de Comportamento

- Adicionamos métodos que representam ações específicas da classe:
  - `updateTitle`, `updateDescription`, `updateAuthor`, e `reschedule` encapsulam lógica de atualização.
  - Cada método aplica validações e atualiza o estado, reduzindo o risco de inconsistência ao alterar diretamente os atributos.
- Adicionamos o método `getSummary` para encapsular a lógica de exibição.

### 2. Validações Centralizadas

- Criamos métodos privados reutilizáveis para validar entradas:
  - `validate(String value, String errorMessage)` simplifica a validação de strings.
  - `validateDate(LocalDate date)` valida datas.

### 3. Removemos Getters e Setters Simples



- Mantivemos apenas `getters` para leitura controlada, eliminando `setters` para impedir alterações

### 3. Removemos Getters e Setters Simples

- Mantivemos apenas `getters` para leitura controlada, eliminando `setters` para impedir alterações diretas.

### 4. Redução de Linhas

- Compactamos as validações ao movê-las para métodos privados reutilizáveis.
- Mantivemos o design simples, evitando criar classes auxiliares desnecessárias que poderiam inflar o NCSS.

---

## Por que essa refatoração resolve o `DataClass`?

- A lógica e o comportamento relacionados à tarefa agora estão dentro da própria classe, como atualização de campos e validação.
- A classe deixou de ser um mero contêiner de dados e passou a ser responsável por seu estado e consistência.



# DataClass ToDo



Let's integrate this additional code into the **Encapsulation Behavior refactoring** approach. We will ensure proper validation, encapsulation, and meaningful behavior while keeping the `toString` method updated with the format specified.

Here is the full refactored version of the `ToDo` class:

## Refactored `ToDo` Class:

java

Copiar código

```
import java.text.MessageFormat;

public class ToDo implements PlannerMaterial {
    private Integer id;
    private String title;
    private String description;
    private int priority;

    // Constructor with validation
    public ToDo(Integer id, String title, String description, int priority) {
        setId(id);
    }
```

```
        // Constructor with validation
        public ToDo(Integer id, String title, String description, int priority) {
            setId(id);
            setTitle(title);
            setDescription(description);
            setPriority(priority);
        }

        // Getter and Setter for id with validation
        public int getId() {
            return id;
        }

        public void setId(Integer id) {
            if (id == null || id <= 0) {
                throw new IllegalArgumentException("ID must be a positive integer.");
            }
            this.id = id;
        }

        // Getter and Setter for title with validation
```

```
public String getTitle() {
    return title;
}

public void setTitle(String title) {
    if (title == null || title.trim().isEmpty()) {
        throw new IllegalArgumentException("Title cannot be null or empty.");
    }
    this.title = title.trim();
}

// Getter and Setter for description
public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = (description != null) ? description.trim() : "";
}

↓

// Getter and Setter for priority with validation
```

```
// Getter and Setter for priority with validation
public int getPriority() {
    return priority;
}

public void setPriority(int priority) {
    if (priority < 1 || priority > 5) { // Assuming priority range is between 1 and 5
        throw new IllegalArgumentException("Priority must be between 1 and 5.");
    }
    this.priority = priority;
}

// Behavior method to check if the task is high-priority
public boolean isHighPriority() {
    return this.priority == 5;
}

// Behavior method to check if the task has a description
public boolean hasDescription() {
    return !this.description.isEmpty() ↓
}

↓
```




```

    }

    // Behavior method to check if the task has a description
    public boolean hasDescription() {
        return !this.description.isEmpty();
    }

    // Overridden toString method with a custom message format
    @Override
    public String toString() {
        return MessageFormat.format(
            "[{0}({1})] {2} {3}",
            id, title, description, priority
        );
    }
}

```

 Copiar código

## Key Changes:

1. **Encapsulation:** All fields are private and accessible only through controlled getters and setters.
2. **Validation:**
  - `setId` ensures the ID is positive and non-null.
  - `setTitle` ensures the title is non-null and non-empty.
  - `setPriority` restricts the priority to a valid range (1–5).
3. **Behavior Methods:**
  - `isHighPriority`: Determines if the task is of the highest priority.
  - `hasDescription`: Checks if the description is not empty.
4. **toString:** Uses `MessageFormat` to generate a readable and well-formatted string representation of the `ToDo` object.

## Why These Changes Matter:

- Encapsulation protects the internal state of the object.
- Validation prevents invalid states and ensures consistent data.

# DataClass Reference

## 1. Validação em Setters

- Os setters agora realizam validações, como verificar se os valores fornecidos são `null`, vazios ou inválidos. Isso ajuda a prevenir dados inconsistentes.

- Exemplo:**

```
java Copiar código

public void setTitle(String title) {
    if (title == null || title.isEmpty()) {
        throw new IllegalArgumentException("Title cannot be null or empty.");
    }
    this.title = title;
}
```

- Essa validação garante que a lógica de integridade do dado esteja diretamente na classe, facilitando a reutilização e manutenção.

## 2. Métodos adicionais

- Foram adicionados comportamentos que encapsulam operações comuns, tornando o uso da classe mais intuitivo e eliminando a necessidade de manipular diretamente os dados.

- Exemplo de métodos adicionados:**

- `calculatePopularity`: Calcula a popularidade combinando visualizações, downloads e compartilhamentos.
- `isAccessible`: Verifica se o recurso é acessível com base nos direitos de acesso.
- `formatForDisplay`: Formata os dados principais da classe para exibição.

## 3. Incrementadores

- Métodos como `incrementViewCount`, `incrementDownloadCount` e `incrementShareCount` foram adicionados para modificar contadores de maneira controlada, evitando erros de manipulação direta.

- Exemplo:**

```
java Copiar código

public void incrementViewCount() {
    this.viewCount++;
}
```