

Refatorações LLM

Descrição	Autor	Data
Depois de clonar o projeto, executei o comando pmd.bat check -f json -R .\regras.xml -d .\ -r .json foram encontrados problemas com as regras: Dataclass, NcssCount e ExcessiveParameterList. Iniciei pelas correções de NcssCount	João Pedro	17/12/2024
Depois de clonar o projeto, executei o comando pmd.bat check -f json -R .\rules.xml -d .\ -r .json foram encontrados problemas com as regras: Dataclass, NcssCount e ExcessiveParameterList. Iniciei pelas correções de DataClass	Francisco Paulino	21/12/2024
Finalizei os code smells DataClass e parti para code smells do tipo ExcessiveParameterList	Francisco Paulino	22/12/2024
Finalizei os code smells NcssCount e parti para code smells do tipo Feature Envy	João Pedro	22/12/2024
Finalizei os code smells DataClass e parti para code smells do tipo ExcessiveParameterList	Francisco Paulino	22/12/2024

Link do repositório

<https://github.com/Joa-Pedro-P-Holanda/LLMCodeSmellRefactor>

Dataclass

.\src\main\java\org\example\studysearch\SearchLog.java

Ele refatorou sugerindo a criação de um novo construtor



```
package org.example.studysearch;
```

```
import java.util.ArrayList;
import java.util.HashMap;...
```

- ◆ You're right, the `SearchLog` class exhibits the "data class" code smell. It primarily holds data and lacks methods that operate on that data. Here are some ways to refactor it:

1. Add methods to manipulate the data:

- Instead of directly modifying `searchHistory` through `addSearchHistory`, consider methods like `logSearch(String searchTerm)` that adds the search term to the history and updates the `searchCount` map if necessary.
- Implement methods to analyze or filter the search history based on specific criteria.
- If applicable, methods for locking/unlocking could involve verification or logging actions.

2. Consider immutability for `searchHistory`:

- Since modifying the search history directly might not be desired, you could create an immutable copy of the list when returning it from `getSearchHistory()`. This encourages creating a new updated list with additional entries rather than modifying the existing one.

3. Explore alternative structures:

- Depending on how you use `searchCount`, it might be more appropriate as a separate class representing "SearchTermCount" with fields like `searchTerm` and `count`.

Here's an example with some refactored methods:

Java



```
public class SearchLog {
    private final List<String> searchHistory; // Make immutable
    private final Map<String, Integer> searchCount;
    private final boolean isLocked;
    private final Integer numUsages;
    private final String logName;

    public SearchLog(String logName) {
        this.searchHistory = new ArrayList<>();
        this.searchCount = new HashMap<>();
        this.logName = logName;
        this.numUsages = 0;
        this.isLocked = false;
    }
}
```

```

// Add methods to manipulate data
public SearchLog logSearch(String searchTerm) {
    if (isLocked) {
        // Handle locked state
        return this;
    }
    List<String> newHistory = new ArrayList<>(searchHistory);
    newHistory.add(searchTerm);
    Map<String, Integer> newCount = updateSearchCount(searchTerm);
    return new SearchLog(logName, newHistory, newCount, numUsages + 1, isLocked);
}

private Map<String, Integer> updateSearchCount(String searchTerm) {
    Map<String, Integer> newCount = new HashMap<>(searchCount);
    newCount.put(searchTerm, newCount.getOrDefault(searchTerm, 0) + 1);
    return newCount;
}

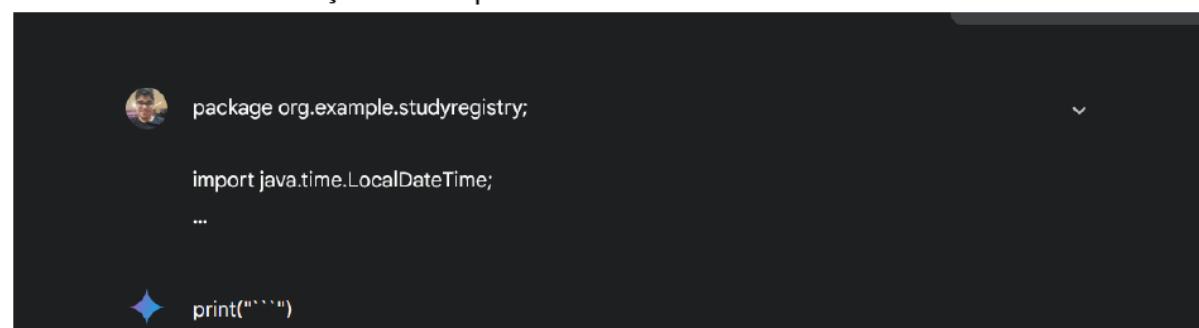
// Getters for immutable data
public List<String> getSearchHistory() {
    return new ArrayList<>(searchHistory); // Return a copy
}

// ... other getters and setters as needed
}

```

\src\main\java\org\example\studyregister\Task.java

Usando o método de adição de comportamentos



```

package org.example.studyregistry;

import java.time.LocalDateTime;
...

◆ print("````")

```

```
package org.example.studyregistry;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

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)
    {
        this.title = title;
        this.name = title;
        this.description = description;
        this.author = author;
        this.date = date;
    }

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

```
public void setTitle(String title) {
    this.title = title;
    this.name = title; // Update name field as well when title is changed.
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

public String getAuthor() {
    return author;
}

public void setAuthor(String author) {
    this.author = author;
}

public LocalDateTime getDate() {
    return date;
}
```

```
public void setDate(LocalDateTime date) {
    this.date = date;
}

// Added method to format date in a human-readable way
public String getFormattedDate() {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
    return formatter.format(date);
}

// Added toString method to provide a more informative representation of the Task
@Override
public String toString() {
    return "Task{" +
        "title='" + title + '\'' +
        ", description='" + description + '\'' +
        ", author='" + author + '\'' +
        ", date=" + date +
        '}';
}
```

Aqui ele não entendeu que era para criar um método que usasse os alterasse dois atributos



```
package org.example.studyregistry;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;...
```

Java



```
The Task class already has methods to modify title and description:
public void setTitle(String title) {
    this.title = title;
    this.name = title; // Update name field as well when title is changed.
}
public void setDescription(String description) {
    this.description = description;
}
```



Um método que altere os dois ao mesmo tempo

◆ Entendendo a solicitação

Você está pedindo um método que permita alterar tanto o título quanto a descrição de uma tarefa simultaneamente, dentro da classe `Task`.

Implementando o método

Análise:

- **Métodos existentes:** A classe `Task` já possui métodos individuais para alterar o título e a descrição.
- **Novo método:** Precisamos criar um método que agrupe essas modificações em uma única operação.

Implementação:

- **Novo método:** Precisamos criar um método que agrupe essas modificações em uma única operação.

Implementação:

```
Java
```

```
public class Task extends Registry{
    // ... outros métodos

    public void setTitleAndDescription(String newTitle, String newDescription) {
        this.title = newTitle;
        this.name = newTitle; // Atualiza o campo name (se necessário)
        this.description = newDescription;
    }
}
```

Explicação:

- **Nome do método:** `setTitleAndDescription` é um nome claro e conciso que indica a sua função.
- **Parâmetros:** Recebe dois parâmetros `String` para os novos valores do título e da descrição.
- **Corpo do método:**
 - Atribui o novo título à propriedade `title`.
 - Atualiza a propriedade `name` (se necessário) com o novo título.
 - Atribui a nova descrição à propriedade `description`.

Java



```
Task task = new Task("Tarefa 1", "Descrição inicial", "Usuário", LocalDateTime.now()  
    // Alterando título e descrição  
    task.setTitleAndDescription("Nova Tarefa", "Descrição atualizada");
```

Considerações adicionais

- **Atomicidade:** Esse novo método garante que as alterações no título e na descrição sejam atômicas, ou seja, ocorrem como uma única operação. Isso é importante em cenários onde a consistência dos dados é crucial.
- **Validação:** Você pode adicionar validação aos novos valores para garantir que eles sejam válidos (por exemplo, não nulos ou vazios).
- **Logs:** Considerar adicionar logs para rastrear as modificações realizadas nas tarefas.
- **Eventos:** Em sistemas mais complexos, você pode disparar eventos quando o título e a descrição de uma tarefa são alterados, permitindo que outros componentes sejam notificados

Benefícios

- **Conveniência:** Permite alterar múltiplos atributos em uma única chamada.
- **Melhora a legibilidade do código:** Torna o código mais conciso e fácil de entender.
- **Maior controle:** Permite realizar validações e outras operações antes de aplicar as alterações.

Com este novo método, você pode simplificar o código que manipula as tarefas e garantir a consistência dos dados.



Não use o campo name



Entendendo a solicitação e a solução

Análise da solicitação:

Você deseja remover a dependência do campo `name` da classe `Task` e criar um método que altere simultaneamente o título e a descrição sem afetar outros campos.

Solução:

A solução proposta anteriormente atualizava o campo `name` sempre que o título era alterado. Para evitar essa dependência, podemos simplesmente remover a linha que atualiza o `name` no método `setTitleAndDescription`.

Código modificado:

```
Java □  
  
public class Task extends Registry {  
    // ... outros métodos  
  
    public void setTitleAndDescription(String newTitle, String newDescription) {  
        this.title = newTitle;  
        this.description = newDescription;  
    }  
}
```

Explicação da mudança:

- **Remoção da atualização de `name`:** A linha `this.name = newTitle;` foi removida, pois não há mais a necessidade de manter o `name` sincronizado com o título.
- **Foco no objetivo:** O método agora se concentra exclusivamente em alterar os atributos `title` e `description` da tarefa.

Considerações adicionais:

- **Propósito do campo `name`:** Se o campo `name` tinha algum propósito específico na sua aplicação, você pode:
 - **Removê-lo:** Se não for mais utilizado, pode ser removido da classe.
 - **Atribuir um valor padrão:** Se tiver um valor padrão relevante, atribua-o no construtor ou em outro local apropriado.
 - **Criar um método separado para acessá-lo:** Se precisar acessar o valor de `name` em algum momento, crie um método específico para isso.

Exemplo de uso com o novo método:

Java

```
Task task = new Task("Tarefa 1", "Descrição inicial", "Usuário", LocalDateTime.now()

// Alterando título e descrição
task.setTitleAndDescription("Nova Tarefa", "Descrição atualizada");

// Acessando o título (se necessário)
String novoTitulo = task.getTitle();
```

Benefícios da mudança

- **Simplicidade:** A classe `Task` fica mais simples e fácil de entender, pois há menos dependências entre os atributos.
- **Flexibilidade:** Permite que você utilize o campo `name` de forma mais independente, se necessário.
- **Evita erros:** Ao remover a atualização automática de `name`, você diminui o risco de introduzir

◆ Testar o Gemini Advanced

- **Evita erros:** Ao remover a atualização automática de `name`, você diminui o risco de introduzir erros em outras partes do código que dependem desse campo.

Com essa modificação, você terá um método `setTitleAndDescription` mais clean e eficiente para alterar os atributos da sua classe `Task`.



```
package org.example.studyregistry;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;...
```

Java

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

```

Java

package org.example.studyregistry;

import java.time.LocalDateTime;

public class Task extends Registry {
    // ... other methods

    public boolean isOverdue() {
        LocalDateTime now = LocalDateTime.now();
        return this.date.isBefore(now);
    }

    // ... other methods
}

```

This code adds a new method `isOverdue` to the `Task` class. Here's a breakdown of the code:

1. `isOverdue` **method**: This method checks if the task's due date (`date` attribute) has already passed.
2. `LocalDateTime.now()` : This gets the current date and time.

1. `isOverdue` **method**: This method checks if the task's due date (`date` attribute) has already passed.
2. `LocalDateTime.now()` : This gets the current date and time.
3. `date.isBefore(now)` : This compares the task's due date (`date`) with the current time (`now`). It returns `true` if the due date is before the current time (meaning the task is overdue).
4. **Return value**: The method returns `true` if the task is overdue and `false` otherwise.

By calling this method on a `Task` object, you can easily determine if the task is behind schedule.

\src\main\java\org\example\studycards\Card.java

De início ele sugeriu a mudança dos atributos para que tivessem a propriedade final



```

package org.example.studycards;

public class Card {
    private String question;...

```



```

Java

package org.example.studycards;

import java.util.Objects;

```

```
public class Card {  
  
    private final String question;  
    private final String answer;  
  
    public Card(String question, String answer) {  
        this.question = Objects.requireNonNull(question, "Question cannot be null");  
        this.answer = Objects.requireNonNull(answer, "Answer cannot be null");  
    }  
  
    public String getQuestion() {  
        return question;  
    }  
  
    public String getAnswer() {  
        return answer;  
    }  
  
    public Card withQuestion(String question) {  
        return new Card(question, this.answer);  
    }  

```

```
public String formatResponse(Integer cardId) {  
    return "[" + cardId + "] " +  
        "The random question was: " + this.getQuestion() + " | " +  
        "The answer is: " + this.getAnswer();  
}  
  
@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);  
}  
}
```

Explanation of Changes:

1. Immutability:

- The `question` and `answer` fields are now declared as `final`, making the `Card` class immutable. This means that once a `Card` object is created, its state cannot be changed.
- Instead of mutator methods (`setQuestion`, `setAnswer`, `edit`), we introduce "copy-and-update" methods:
 - `withQuestion(String question)` : Creates a new `Card` object with the given `question` and the original `answer`.
 - `withAnswer(String answer)` : Creates a new `Card` object with the original `question` and the given `answer`.

2. Null Checks:

- The constructor now includes null checks for both `question` and `answer` using `Objects.requireNonNull`.

3. Improved Equals and HashCode:

- The `equals` and `hashCode` methods are overridden to correctly compare and hash `Card` objects based on their `question` and `answer` fields.

Benefits of Immutability:

- **Thread Safety:** Immutable objects are inherently thread-safe, as their state cannot be modified concurrently.
- **Predictability:** The behavior of immutable objects is more predictable, as their state remains constant throughout their lifetime.
- **Easier to Reason About:** Immutable objects are easier to reason about and debug, as their state does not change unexpectedly.
- **Suitable for Caching:** Immutable objects can be safely cached without concerns about concurrent modifications.

By applying the "move method" technique to create immutable objects with copy-and-update methods, we have addressed the data class code smell and improved the overall design and maintainability of the `Card` class.

Mas como os atributos precisavam ser mutáveis ele gerou o código com os gets e o método de edit agora sem o final, apenas com a verificação se dos atributos no construtor



Não altere a mutabilidade dos atributos eles precisam ser alteráveis, remova o final de devolva os sets

◆ Analisando a Solicitação e Propondo uma Solução

Entendendo a Solicitação:

A solicitação pede para manter a mutabilidade dos atributos da classe `Card` e remover a parte do código que retorna os sets.

Solução Proposta:

A solução abaixo mantém a mutabilidade dos atributos, removendo os métodos `withQuestion` e `withAnswer`, que eram responsáveis pela criação de novos objetos com valores atualizados.

Além disso, a parte do código que retorna os sets foi removida, pois não está claro qual a finalidade desses sets.

Java

```
package org.example.studycards;

import java.util.Objects;

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

    public Card(String question, String answer) {
        this.question = Objects.requireNonNull(question, "Question cannot be null");
        this.answer = Objects.requireNonNull(answer, "Answer cannot be null");
    }

    public String getQuestion() {
        return question;
    }

    public void setQuestion(String question) {
        this.question = question;
    }

    public void setAnswer(String answer) {
        this.answer = answer;
    }

    public void edit(String question, String answer) {
        setQuestion(question);
        setAnswer(answer);
    }

    public String formatResponse(Integer cardId) {
        return "[" + cardId + "] " +
            "The random question was: " + this.getQuestion() + " | " +
            "The answer is: " + this.answer;
    }

    // ... (outros métodos, se houver)
}
```

Explicação das Alterações:

- **Remoção de Métodos:** Os métodos `withQuestion` e `withAnswer` foram removidos, pois eles eram responsáveis pela criação de novos objetos imutáveis.
- **Manutenção da Mutabilidade:** Os atributos `question` e `answer` continuam sendo modificáveis através dos métodos `setQuestion` e `setAnswer`.
- **Remoção da Parte sobre Sets:** A parte do código que retorna os sets foi removida, pois não há informações suficientes para entender sua função.

Considerações:

- **Mutabilidade vs. Immutabilidade:** A escolha entre mutabilidade e imutabilidade depende do contexto da aplicação. A mutabilidade pode ser útil em situações onde é necessário modificar objetos existentes, mas pode levar a erros mais difíceis de depurar. A imutabilidade, por outro lado, pode tornar o código mais seguro e previsível.

- **Sets:** Se a intenção era retornar conjuntos de cartas, seria necessário criar uma classe separada para representar esses conjuntos e adicionar métodos à classe `Card` para adicionar ou remover cartas desses conjuntos.

Observações:

- **Null Checks:** Os null checks no construtor garantem que os atributos `question` e `answer` nunca sejam nulos.
- **Método `edit`:** O método `edit` é uma conveniência para modificar ambos os atributos de uma vez.

Próximos Passos:

- **Analizar a Necessidade de Sets:** Se você precisa de alguma funcionalidade relacionada a conjuntos de cartas, podemos adicionar essa funcionalidade à classe `Card` ou criar uma nova classe para representá-los.
- **Considerar a Immutabilidade:** Se você deseja tornar a classe `Card` imutável no futuro, pode reintroduzir os métodos `withQuestion` e `withAnswer` e tornar os atributos `final`.

reintroduzir os métodos `withQuestion` e `withAnswer` e tornar os atributos `final`.

- **Adicionar Validação:** Você pode adicionar validação aos métodos `setQuestion` e `setAnswer` para garantir que os valores atribuídos sejam válidos.

Lembre-se: A escolha da melhor abordagem depende das suas necessidades específicas e do contexto da sua aplicação.

Gostaria de adicionar **alguma outra funcionalidade** à classe `Card`?



\src\main\java\org\example\studymaterial\Reference.java

Inicialmente ele queria usar o padrão Builder e tornar os atributos imutáveis

<https://g.co/gemini/share/654e8d0b9419>

```
// Add methods for common actions
public void incrementViewCount() {
    this.viewCount++;
}

public void incrementDownloadCount() {
    this.downloadCount++;
}

public void incrementShareCount() {
    this.shareCount++;
}

public abstract void incrementCount(Map<String, Integer> referenceCount);
}
```

Enhancements:

- **Added "add" methods for all attributes:** This provides a complete set of methods for modifying the `Reference` object.

Example Usage:

Java

```
Reference reference = new Reference("My Title", "My Description", "mylink.com",
                                    "Public", "CC BY-SA", true, 4, "English",
                                    0, 0, 0);

referenceWithTitle("Updated Title")
    .withDescription("Updated Description")
    .incrementViewCount()
    .incrementDownloadCount();
```

This enhanced version of the `Reference` class demonstrates the power of the "add method" technique by providing a more fluent and expressive way to create and modify `Reference` objects.

I hope this improved version is more helpful!

\src\main\java\org\example\studyplanner\ToDo.java

<https://g.co/gemini/share/d900dec36f59>

NcssCount

.\src\main\java\org\example\studycards\LeitnerSystem.java

<https://g.co/gemini/share/62472b1ad798>

Na primeira sugestão o LLM alterou os métodos errados, e criou um novo método com uma chamada a um método que não existe na classe alvo. Então enviei um novo prompt. Após especificar qual era o método com o tamanho excessivo o prompt de IA seguinte removeu o code smell, e os testes automatizadas continuaram executando corretamente

.\src\main\java\org\example\studyplanner\KanbanView.java

<https://g.co/gemini/share/23ca8a44e1d0>

No primeiro prompt a IA não retornou o arquivo inteiro então os testes falharam, após pedir pelo arquivo inteiro tive que mandar um terceiro prompt pedindo para extrair parte do comportamento no método para um novo método, em um quarto prompt sugeri um novo parâmetro para o método extraído. Depois extrair a separação de linhas para o outro método também e por fim criei um novo método intermediário com partes do StringBuilder usadas.

.\src\main\java\org\example\studyplanner\TodoTracker.java

<https://g.co/gemini/share/5b0ce3a3d66a>

No primeiro prompt o modelo teve uma alucinação e não respondeu adequadamente, no segundo, retornou os métodos modificados e o smell foi removido adequadamente

.\src\main\java\org\example\studyregistry\StudyGoal.java

<https://g.co/gemini/share/0295b45fcda8>

No primeiro prompt a IA interpretou erroneamente e moveu a responsabilidade para um método privado em vez de extrair o comportamento das declarações do método, no segundo prompt a IA resolveu o problema adequadamente.

.\src\main\java\org\example\studyregistry\StudyMaterial.java

O primeiro prompt removeu o code smell da função original, mas ao extrair para uma função auxiliar acabou por apenas mover o smell para o método auxiliar criado



fix the NcssCount code smell using the extract method refactoring technique

▼

```
package org.example.studyregistry;  
import org.example.studymaterial.AudioReference;...
```



Certainly, applying the Extract Method refactoring technique can address the NcssCount code smell in the provided `getReferenceCountMap` method of the `StudyMaterial` class. Here's how we can refactor the code:

Java



```
public Map<String, Integer> getReferenceCountMap() {  
    Map<String, Integer> response = new HashMap<>();  
    response.put("Audio References", 0);  
    response.put("Video References", 0);  
    response.put("Text References", 0);  
  
    for (Reference reference : references) {
```

```
        for (Reference reference : references) {
            incrementCount(reference, response);
        }

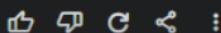
        return response;
    }

private void incrementCount(Reference reference, Map<String, Integer> referenceCount) {
    if (reference.getClass() == AudioReference.class) {
        Integer audioCount = referenceCount.get("Audio References");
        referenceCount.put("Audio References", audioCount + 1);
    } else if (reference.getClass() == VideoReference.class) {
        if (((VideoReference) reference).handleStreamAvailability()) {
            Integer videoCount = referenceCount.get("Video References");
            referenceCount.put("Video References", videoCount + 1);
        }
    } else if (reference.getClass() == TextReference.class) {
        if (((TextReference) reference).handleTextAccess()) {
            Integer textCount = referenceCount.get("Text References");
            referenceCount.put("Text References", textCount + 1);
        }
    }
}
```

Explanation:

1. We've extracted the logic for incrementing the reference count based on the reference type into a separate method named `incrementCount`.
2. The `getRefCountMap` method now iterates through the references and calls the `incrementCount` method for each reference, passing the reference and the reference count map as arguments.
3. The `incrementCount` method checks the reference type and increments the corresponding count in the reference count map.

This refactoring improves the readability and maintainability of the code. The `getRefCountMap` method is now more concise and focused on its primary responsibility of iterating through references. The logic for incrementing the count is encapsulated in a separate method, making it easier to understand and reuse.



No prompt seguinte tentei refatorar com uma técnica diferente e o code smell foi solucionado corretamente, mas o método que foi extraído na primeira tentativa não foi mais utilizado



use the move method refactoring technique to implement the incrementCount on each subclass of reference



- 💡 Certainly, let's apply the Move Method refactoring technique to move the `incrementCount` logic to the respective subclasses of `Reference`.

Here's how we can refactor the code:

1. Modify Reference Class:

Java

```
public abstract class Reference {  
    // ... other methods ...  
  
    public abstract void incrementCount(Map<String, Integer> referenceCount);  
}
```



2. Modify AudioReference Class:

Java

```
public class AudioReference extends Reference {  
    // ... other methods ...  
  
    @Override  
    public void incrementCount(Map<String, Integer> referenceCount) {  
        Integer audioCount = referenceCount.get("Audio References");  
        referenceCount.put("Audio References", audioCount + 1);  
    }  
}
```



3. Modify VideoReference Class:

Java

```
public class VideoReference extends Reference {
    // ... other methods ...

    @Override
    public void incrementCount(Map<String, Integer> referenceCount) {
        if (handleStreamAvailability()) {
            Integer videoCount = referenceCount.get("Video References");
            referenceCount.put("Video References", videoCount + 1);
        }
    }
}
```

4. Modify TextReference Class:

Java

```
public class TextReference extends Reference {
    // ... other methods ...

    @Override
    public void incrementCount(Map<String, Integer> referenceCount) {
        if (handleTextAccess()) {
            Integer textCount = referenceCount.get("Text References");
            referenceCount.put("Text References", textCount + 1);
        }
    }
}
```

5. Modify StudyMaterial Class:

Java

```
public class StudyMaterial {  
    // ... other methods ...  
  
    public Map<String, Integer> getReferenceCountMap() {  
        Map<String, Integer> response = new HashMap<>();  
        response.put("Audio References", 0);  
        response.put("Video References", 0);  
        response.put("Text References", 0);  
  
        for (Reference reference : references) {  
            reference.incrementCount(response);  
        }  
  
        return response;  
    }  
}
```

Feature Envy

<.\src\main\java\org\example\controllers\StudyCardsController.java>

No primeiro prompt o code smell foi removido adequadamente, mas o método causador continuou na classe, mesmo não sendo usado. Além disso, o método `getRandomCardFromBox` era privado na classe LeitnerSystem e não era utilizado, o Gemini alterou a implementação dele e converteu para público.

<https://g.co/gemini/share/0b50857e2528>

<.\src\main\java\org\example\controllers\TimelineView.java>

No primeiro prompt a tipagem do id foi mudada erroneamente para String, no segundo prompt foi corrigida, mas o smell permaneceu e no terceiro prompt o smell foi removido, mas adicionado novamente no método privado `getHabitViewString` criado, utilizando a técnica *Preserve whole object* ao final solucionou o code smell inicial e o novo code smell intermediário.

<https://g.co/gemini/share/0921b95b2f58>

<.\src\main\java\org\example\studysearch\GeneralSearch.java>

<https://g.co/gemini/share/56d132dc51e2>

No primeiro prompt o método move method não foi aplicado para remover o code smell, então no segundo prompt solicitei para que a lógica de adição ao histórico no final do método *handleSearch* fosse extraída para um outro método, mas esse método foi extraído para a mesma classe, no terceiro prompt o método foi corretamente movido para outra classe

.\src\main\java\org\example\studysearch\MaterialSearch.java

<https://g.co/gemini/share/53e67d7a3247>

No primeiro prompt o método auxiliar *handleMaterialSearch* foi removido e o code smell também

.\src\main\java\org\example\studysearch\RegistrySearch.java

<https://g.co/gemini/share/04e985cbad4c>

No primeiro prompt, um dos métodos criados na classe SearchLog já existia então não inseri a duplicata, mas o code smell foi removido com sucesso

ExcessiveParameterList(OBS: Pela mudança de alguns parâmetros alguns testes acabaram quebrando)

.\src\main\java\org\example\studyplanner\HabitTracker.java

Usei a técnica “ParameterObjective”

<https://g.co/gemini/share/0ac4cd50e612>

.\src\main\java\org\example\studyregister\StudyObjective.java

Usei principalmente o DTO, mas posteriormente também usei o record e add methods

<https://g.co/gemini/share/503c8ad10201>

.\src\main\java\org\example\studyregistry\StudyPlan.java

<https://g.co/gemini/share/961fcb5ea5eb>

`\src\main\java\org\example\studyregistry\StudyTaskManage.java`

O principal foi o DTO, mas também foi usado classe record.

<https://g.co/gemini/share/1063c1a872fc>

`\src\main\java\org\example\studymaterial\AudioReference.java`

Essa solução foi uma mistura de métodos, como extract method, criação de uma nova classe e padrão builder

<https://g.co/gemini/share/1d67e3adb179>