

NcssCount LeitnerSystem

```
response += "The answer is: " + card.getAnswer();  
return response;  
} fix the NcssCount code smell using the extract method  
refactoring technique
```



To address the NcssCount code smell using the extract method refactoring technique, we can break down the method into smaller, more focused methods. This approach reduces the complexity of the `getRandomCard` method, improves readability, and adheres to the **Single Responsibility Principle**.

Here's the refactored code:

java

Copiar Editor

```
public String getRandomCard(List<Box> otherBoxes) {  
    if (isEmpty(otherBoxes)) {  
        return null;  
    }  
}
```



```
Box allBoxes = mergeAllBoxes(otherBoxes);
```

```
Box allBoxes = mergeAllBoxes(otherBoxes);  
Integer randomCard = allBoxes.getRandomCard();
```

```
if (randomCard == null) {  
    return "No card found";  
}
```

```
return generateResponse(randomCard);  
}
```

```
private boolean isEmpty(List<Box> otherBoxes) {  
    return otherBoxes == null || otherBoxes.isEmpty();  
}
```

```
private Box mergeAllBoxes(List<Box> otherBoxes) {  
    Box allBoxes = new Box();  
    for (Box box : otherBoxes) {  
        allBoxes.addCards(box.getCards());  
    }  
    return allBoxes;  
}
```




```
private String generateResponse(Integer randomCard) {
    CardManager manager = CardManager.getCardManager();
    Card card = manager.getCard(randomCard);

    String response = "[" + randomCard + "] ";
    response += "The random question was: " + card.getQuestion() + " | ";
    response += "The answer is: " + card.getAnswer();

    return response;
}
```

Explanation of Changes:

1. Extracted Method:

- `isEmpty` checks whether the list is null or empty. This makes the null/empty check reusable and descriptive.
- `mergeAllBoxes` handles merging cards  from all boxes into a single `Box` object.

Explanation of Changes:


1. Extracted Method:

- `isEmpty` checks whether the list is null or empty. This makes the null/empty check reusable and descriptive.
- `mergeAllBoxes` handles merging cards from all boxes into a single `Box` object.
- `generateResponse` is responsible for creating the final response string using the random card details.

2. Simpler Main Method:

- The `getRandomCard` method is now concise and delegates specific tasks to helper methods.
- It clearly shows the sequence of operations: validation, merging, card selection, and response generation.

3. Improved Readability:

- Each helper method has a single responsibility  and a descriptive name, making the code easier to understand and maintain.

NcssCount KanbanView

1. Redução de Código Redundante

- Alteração:** Nos métodos `addHabitToKanban` e `addToDoToKanban`, a lógica para verificar se o objeto é nulo e lançar uma exceção foi simplificada, removendo o bloco `try-catch` desnecessário.
- Motivação:** Evitar o uso desnecessário de blocos `try-catch` para operações que já possuem validação explícita. Isso reduz o número de linhas e melhora a clareza.

2. Extração de Lógica Repetitiva

- Alteração:** No método `kanbanView`, foi criado um método auxiliar privado chamado `appendStateMaterials` para evitar duplicação de código ao construir as partes do estado do Kanban (`TODO`, `DOING`, `DONE`).
- Motivação:** Promove a reutilização de código e facilita futuras alterações, já que a lógica para exibir os materiais de um estado específico está centralizada.

3. Simplificação de Condições

- Alteração:** No método `kanbanView`, a verificação `if (kanban.isEmpty())` foi mantida, mas agora o código trata os estados individualmente de forma mais clara com o método auxiliar.
- Motivação:** A estrutura ficou mais limpa e fácil de seguir.

4. Uso de Boa Prática para StringBuilder

- Alteração:** O uso do `StringBuilder` foi reorganizado para construir a string do Kanban de maneira mais concisa e evitar manipulação direta dentro do método principal.
- Motivação:** Melhor organização e separação de responsabilidades.

5. Melhor Nomeação e Organização

- Alteração:** O método `appendStateMaterials` tem um nome claro e descreve exatamente o que faz. Isso aumenta a legibilidade do código.
- Motivação:** Nomear métodos e variáveis de forma descritiva facilita a compreensão do código por outros desenvolvedores ou pelo próprio autor no futuro.



6. Remoção de Blocos de Exceção Duplicados

- Alteração:** Blocos de `try-catch` nos métodos de adição e remoção (`addHabitToKanban`, `removeHabitFromKanban`, etc.) foram eliminados porque não eram necessários para tratar exceções já verificadas manualmente.
- Motivação:** Evitar tratamentos redundantes e foco na lógica essencial.

NcssCount TodoTracker

Código:

java

 Copiar  Editar

```
public List<String> handleSearch(String text) {  
    List<String> results = new ArrayList<>();  
    for (ToDo toDo : toDos) {  
        if (toDo.toString().toLowerCase().contains(text.toLowerCase())) {  
            results.add(toDo.toString());  
        }  
    }  
    return results;  
}
```

Benefícios:

- Facilita a busca de `ToDo` diretamente pelo texto fornecido.
- Retorna strings prontas para exibição, simplificando o uso do método.





Alterações:

1. O método principal delega a formatação de dados para outros submétodos.
2. Foi criado o método `getExecutionTimesAsString` para lidar com os horários de execução (`tracker`).

Código:

java

 Copiar  Editar

```
@Override  
public String toString() {  
    StringBuilder str = new StringBuilder();  
    for (ToDo toDo : toDos) {  
        str.append(toDo.toString()).append("\n");  
        str.append(getExecutionTimesAsString(toDo.getId())).append("\n");  
    }  
    String response = str.toString();  
    return response.isEmpty() ? "No Todos found" : response;  
}
```



```

    }
    String response = str.toString();
    return response.isEmpty() ? "No Todos found" : response;
}

private String getExecutionTimesAsString(Integer id) {
    List<LocalDateTime> todosDate = this.tracker.get(id);
    if (todosDate == null) {
        return "No tracks found";
    }
    StringBuilder str = new StringBuilder();
    for (LocalDateTime ldt : todosDate) {
        String pattern = "yyyy-MM-dd HH:mm:ss";
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern);
        str.append(formatter.format(ldt)).append("\n");
    }
    return str.toString();
}

```

Benefícios:

Benefícios:

- Reduz a complexidade do método `toString` principal.
- Facilita a manutenção do código, pois cada submétodo tem uma responsabilidade única.

Código:

```

java
public List<ToDo> sortTodosByPriority() {
    List<ToDo> sortedTodos = new ArrayList<>(todos);
    sortedTodos.sort(Comparator.comparingInt(ToDo::getPriority));
    return sortedTodos;
}

```

Benefícios:

- Permite recuperar uma lista ordenada por prioridade sem alterar a lista original.
- Usa a API de comparadores do Java para simplificar a lógica.

Resumo das Melhorias

1. **Modularidade:** A lógica foi dividida em métodos menores, cada um com uma responsabilidade clara.
2. **Legibilidade:** O código ficou mais fácil de entender e manter.
3. **Reutilização:** Métodos como `handleSearch` e `getExecutionTimesAsString` podem ser usados em outros contextos.
4. **Flexibilidade:** O método `sortTodosByPriority` adiciona uma funcionalidade útil sem impactar outros métodos.

NcssCount StudyGoal

`return summary.toString();`
} fix the NcssCount code smell using the extract method refactoring technique



To fix the `NcssCount` code smell in your method using the extract method refactoring technique, you can break down the method into smaller, more manageable methods, each handling a specific part of the goal summary generation. Here's a refactored version of your code:

java

Copiar Editar

```
public String setGoalSummary() {  
    StringBuilder summary = new StringBuilder();  
    summary.append("Goal Summary:\n").append("\n\n");  
  
    addActiveGoal(summary);  
    addCompletedGoal(summary);  
    addGoalRequirements(summary);  
    addStudyPlan(summary);  
    addStudyObjective(summary);  
}
```

```

        this.summary = summary.toString();
        return summary.toString();
    }

    private void addActiveGoal(StringBuilder summary) {
        if (this.isActive) {
            summary.append("Active Goal:\n").append(goal).append("\n\n");
        }
    }

    private void addCompletedGoal(StringBuilder summary) {
        if (this.isCompleted) {
            summary.append("Completed Goal:\n").append(goal).append("\n\n");
        }
    }

    private void addGoalRequirements(StringBuilder summary) {
        if (this.goalRequirements != null) {
            summary.append("Requirements:\n");
            for (String requirement : this.goalRequirements) {
                summary.append(requirement).append(", ");
            }
        }
    }

```

```

    private void addGoalRequirements(StringBuilder summary) {
        if (this.goalRequirements != null) {
            summary.append("Requirements:\n");
            for (String requirement : this.goalRequirements) {
                summary.append(requirement).append(", ");
            }
            // Remove trailing comma and space
            int length = summary.length();
            if (length > 2 && summary.substring(length - 2).equals(", ")) {
                summary.setLength(length - 2);
            }
            summary.append("\n\n");
        }
    }

    private void addStudyPlan(StringBuilder summary) {
        if (this.studyPlan != null) {
            summary.append("Plan:\n").append(this.studyPlan.toString()).append("\n\n");
        }
    }

```

```
private void addStudyPlan(StringBuilder summary) {
    if (this.studyPlan != null) {
        summary.append("Plan:\n").append(this.studyPlan.toString()).append("\n\n");
    }
}

private void addStudyObjective(StringBuilder summary) {
    if (this.studyObjective != null) {
        summary.append("Objective:\n").append(this.studyObjective.toString()).append("\n\n");
    }
}
```

Key Improvements:

Key Improvements:


1. **Extracted Methods:** Each section of the summary generation now has its own dedicated method (`addActiveGoal` , `addCompletedGoal` , etc.), making the code more modular and easier to read.
2. **Code Reuse:** The main `setGoalSummary` method delegates tasks to the extracted methods, improving clarity.
3. **Trailing Comma Removal:** In the `addGoalRequirements` method, the trailing comma is removed if necessary, ensuring a clean output.

This approach adheres to the Single Responsibility Principle, making each method focused on a single task and easier to maintain.

1. Refatoração do método `getReferenceCountMap`

- **Sem refatoração:** O método continha toda a lógica de contagem de referências diretamente no mesmo método. Ele era menos legível e tinha múltiplas responsabilidades.
- **Refatorado:** A lógica foi dividida em métodos auxiliares:
 - `initializeReferenceCount(Map<String, Integer> response)` : Inicializa o mapa com os contadores de referências.
 - `countAudioReferences()` : Conta referências de áudio e retorna um mapa com o resultado.
 - `countVideoReferences()` : Conta referências de vídeo disponíveis e retorna um mapa com o resultado.
 - `countTextReferences()` : Conta referências de texto acessíveis e retorna um mapa com o resultado.

2. Criação de métodos auxiliares

- A lógica de contagem específica para cada tipo de referência foi isolada em métodos independentes:
 - Cada método (`countAudioReferences` , `countVideoReferences` , `countTextReferences`) faz a contagem apenas para seu tipo específico.
 - Isso melhora a legibilidade e facilita a manutenção, caso a lógica para contagem de um tipo específico mude no futuro. 

3. Modularidade e reutilização

- A separação em métodos auxiliares permite reutilização em outras partes do código, caso seja necessário realizar contagens individuais fora do escopo de `getReferenceCountMap`.
- A modularidade melhora a testabilidade, pois cada método pode ser testado separadamente.

4. Melhoria na legibilidade

- A lógica agora é mais clara, com nomes de métodos que indicam exatamente o que eles fazem.
- Reduziu-se a complexidade dentro do método principal, tornando o código mais limpo e fácil de entender.

Essas mudanças não alteram a funcionalidade, mas tornam o código mais organizado, escalável e alinhado a boas práticas de programação.