



**UNIVERSIDADE
FEDERAL DO CEARÁ**
CAMPUS DE QUIXADÁ

Universidade Federal do Ceará - campus Quixadá

Qualidade de Software

Histórico de Conversas Resolução de Code Smells

Docente:

Carla Ilane Moreira Bezerra

Discentes:

Erick Gabriel Ferreira Gaspar

536261;

Michael Farias de Lima

537835;

Av. José de Freitas Queiroz, 5003, Quixadá - CE, 63902-580
19.05.2023

Long Methods

1)

Aqui está o código ajustado, aplicando a técnica de **Extract Method** para resolver o "Long Method Code Smell". Eu separei as partes longas de métodos em métodos auxiliares menores e mais específicos, o que melhora a legibilidade e a manutenção do código.

Código Refatorado:

```
java Copiar código

package org.example.studycards;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class LeitnerSystem extends StudyMethod {
    List<Box> boxes = null;

    public LeitnerSystem(String methodName) {
        super(methodName);
        initializeBoxes();
    }

    private void initializeBoxes() {
        boxes = new ArrayList<>(Arrays.asList(new Box(), new Box(), new Box(), new
    }

    @Override
    public String getMethodName() {
        return this.methodName;
    }

    @Override
    void setMethodName(String methodName) {
        this.methodName = methodName;
    }

    @Override
    public String toString() {
        return buildBoxesRepresentation();
    }

    private String buildBoxesRepresentation() {
        StringBuilder response = new StringBuilder();
        int index = 0;
        for (Box box : boxes) {
            response.append("Box ").append(index).append(": ").append(box.toString());
            index++;
        }
        return response.toString();
    }

    public void clearBoxes() {
        initializeBoxes();
    }
}
```

```
    initializeBoxes();
}

public List<Box> getBoxes() {
    return boxes;
}

public String getRandomCard(List<Box> otherBoxes) {
    if (otherBoxes == null || otherBoxes.isEmpty()) {
        return "No card found";
    }

    Box combinedBox = combineAllCards(otherBoxes);
    Integer randomCard = combinedBox.getRandomCard();
    return randomCard != null ? buildRandomCardResponse(randomCard) : "No card
}

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

private String buildRandomCardResponse(Integer randomCard) {
    CardManager manager = CardManager.getCardManager();
    Card card = manager.getCard(randomCard);
    return "[" + randomCard + "] The random question was: " + card.getQuestion()
        " | The answer is: " + card.getAnswer();
}

public void addCardToBox(Integer id, Integer boxId) {
    boxes.get(boxId).addCard(id);
}

public void removeCardFromBox(Integer id, Integer boxId) {
    boxes.get(boxId).removeCard(id);
}

public Card takeCardFromBox(Integer boxId) {
    Integer cardId = boxes.get(boxId).getRandomCard();
    return this.cardManager.getCard(cardId);
}

public void boxIdValidation(Integer boxId) throws Exception {
    if (boxId == null || boxId > (boxes.size() - 1) || boxId < 0) {
        throw new Exception("Invalid box ID");
    }
}

public void upgradeCard(Integer cardId, Integer boxId) throws Exception {
    processCardMovement(cardId, ↓ :Id, 1);
}
```

```

        boxes.get(boxId).removeCard(id);
    }

    public Card takeCardFromBox(Integer boxId) {
        Integer cardId = boxes.get(boxId).getRandomCard();
        return this.cardManager.getCard(cardId);
    }

    public void boxIdValidation(Integer boxId) throws Exception {
        if (boxId == null || boxId > (boxes.size() - 1) || boxId < 0) {
            throw new Exception("Invalid box ID");
        }
    }

    public void upgradeCard(Integer cardId, Integer boxId) throws Exception {
        processCardMovement(cardId, boxId, 1);
    }

    public void downgradeCard(Integer cardId, Integer boxId) throws Exception {
        processCardMovement(cardId, boxId, -1);
    }

    private void processCardMovement(Integer cardId, Integer boxId, int direction)
        boxIdValidation(boxId);

        Box refBox = boxes.get(boxId);
        if (!refBox.hasCard(cardId)) {
            throw new Exception("No card Found");
        }

        refBox.removeCard(cardId);
        int newBoxId = Math.min(Math.max(boxId + direction, 0), boxes.size() - 1);
        boxes.get(newBoxId).addCard(cardId);
    }
}

```

Principais mudanças:

- `initializeBoxes`: Método extraído para inicializar as caixas, usado tanto no construtor quanto no método `clearBoxes`.
- `buildBoxesRepresentation`: Extraiu a lógica do `toString` para tornar o método mais curto.
- `combineAllCards` e `buildRandomCardResponse`: Métodos auxiliares criados para encapsular a lógica de combinar cartões e construir a resposta para `getRandomCard`.
- `processCardMovement`: Método genérico para mover o cartão para cima ou para baixo, reduzindo duplicação entre `upgradeCard` e `downgradeCard`.

Essas mudanças tornam o código mais modular, legível e fácil de manter.



2)

Você está correto: a abordagem anterior aliviou o problema, mas a estrutura ainda pode ser melhorada para eliminar completamente o *Long Method Code Smell*. Vamos fazer uma refatoração mais profunda, extraiendo a lógica de construção das strings relacionadas ao Kanban em métodos auxiliares menores e completamente independentes.

Aqui está a versão aprimorada e mais limpa:

Código Refatorado

```
java Copiar código

public String kanbanView() throws Exception {
    try {
        validateKanbanNotEmpty();

        StringBuilder sb = new StringBuilder();
        sb.append("[ Material ToDo: ").append(System.lineSeparator())
          .append(getFormattedState(State.TODO));

        sb.append("Material in progress:").append(System.lineSeparator())
          .append(getFormattedState(State.DOING));

        sb.append("Material completed:").append(System.lineSeparator())
          .append(getFormattedState(State.DONE));

        sb.append("]");
        return sb.toString();
    } catch (Exception e) {
        throw new Exception(e.getMessage());
    }
}

private void validateKanbanNotEmpty() throws Exception {
    if (kanban.isEmpty()) {
        throw new Exception("No material found");
    }
}

private String getFormattedState(State state) {
    List<PlannerMaterial> materials = kanban.get(state);
    if (materials.isEmpty()) {
        return "No material found" + System.lineSeparator();
    }
    return formatMaterials(materials);
}

private String formatMaterials(List<PlannerMaterial> materials) {
    StringBuilder formattedMaterials = new StringBuilder();
    for (PlannerMaterial material : materials) {
        formattedMaterials.append(", ").append(material.toString());
    }
    formattedMaterials.append(System.lineSeparator());
    return formattedMaterials.toString();
}
```

```
    return formatMaterials(materials);
}

private String formatMaterials(List<PlannerMaterial> materials) {
    StringBuilder formattedMaterials = new StringBuilder();
    for (PlannerMaterial material : materials) {
        formattedMaterials.append(", ").append(material.toString());
    }
    formattedMaterials.append(System.lineSeparator());
    return formattedMaterials.toString();
}
```

Copiar código

O que foi ajustado:

1. Criação de `validateKanbanNotEmpty`:
 - Um método dedicado para validar se o Kanban não está vazio. Isso encapsula a validação e a torna reutilizável.
2. Extração de `getFormattedState`:
 - Este método processa o estado específico e chama outro método (`formatMaterials`) para lidar com a lógica de formatação de uma lista de materiais.
 - Lida com a mensagem de lista vazia e separa a responsabilidade da lógica de formatação.
3. Criação de `formatMaterials`:
 - Este método recebe uma lista de `PlannerMaterial` e retorna a string formatada.
 - Centraliza a lógica de iteração e concatenação.

Por que agora está completamente refatorado?

- **Métodos Pequenos:** Cada método tem uma responsabilidade única, contribuindo para a coesão e reduzindo a complexidade.
- **Separação de Preocupações:** A validação, a formatação por estado e a formatação de materiais são tratadas de forma independente.
- **Facilidade de Testes:** Cada método extraído pode ser testado isoladamente.
- **Eliminação Completa do Long Method:** O método `kanbanView` agora apenas orquestra os métodos auxiliares, ficando compacto e legível.

Se houver mais ajustes ou regras específicas que você deseja implementar, posso continuar refinando!



3)

To address the **Long Method** code smell in the `toString` method of the `TodoTracker` class, we can break it into smaller, more focused methods using the **Extract Method** refactoring technique. Below is the refactored code:

Refactored Code:

```
java Copiar código

@Override
public String toString() {
    if (toDos.isEmpty()) {
        return "No ToDos found";
    }

    StringBuilder str = new StringBuilder();
    for (ToDo todo : toDos) {
        appendToDoDetails(str, todo);
    }
    return str.toString();
}

private void appendToDoDetails(StringBuilder str, ToDo todo) {
    str.append(todo.toString()).append("\n");
    Integer id = todo.getId();
    List<LocalDateTime> todosDate = tracker.get(id);
    if (todosDate == null || todosDate.isEmpty()) {
        str.append("No tracks found\n");
    } else {
        appendExecutionTimes(str, todosDate);
    }
}

private void appendExecutionTimes(StringBuilder str, List<LocalDateTime> todosDate) {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    for (LocalDateTime ldt : todosDate) {
        str.append(formatter.format(ldt)).append("\n");
    }
}
```

Changes Made:

1. Extracted Method `appendToDoDetails`:
 - Handles the details of each `ToDo`, including its `id` and associated execution times.
 - Makes the `toString` method cleaner and more concise.
2. Extracted Method `appendExecutionTimes`:
 - Focuses solely on formatting and appending the execution times.
 - Improves readability and separation concerns.

Você chegou ao limite do plano Free para o GPT-4o.
As respostas serão fornecidas por outro modelo até o seu limite ser redefinido após
18:39.

Obter Plus



```

        if (toDos.isEmpty()) {
            return "No ToDos found";
        }

        StringBuilder str = new StringBuilder();
        for (ToDo ToDo : toDos) {
            appendToDoDetails(str, ToDo);
        }
        return str.toString();
    }

    private void appendToDoDetails(StringBuilder str, ToDo ToDo) {
        str.append(ToDo.toString()).append("\n");
        Integer id = ToDo.getId();
        List<LocalDateTime> todosDate = tracker.get(id);
        if (todosDate == null || todosDate.isEmpty()) {
            str.append("No tracks found\n");
        } else {
            appendExecutionTimes(str, todosDate);
        }
    }

    private void appendExecutionTimes(StringBuilder str, List<LocalDateTime> todosDate) {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        for (LocalDateTime ldt : todosDate) {
            str.append(formatter.format(ldt)).append("\n");
        }
    }
}

```

Changes Made:

1. Extracted Method `appendToDoDetails`:
 - Handles the details of each `ToDo`, including its `id` and associated execution times.
 - Makes the `toString` method cleaner and more concise.
2. Extracted Method `appendExecutionTimes`:
 - Focuses solely on formatting and appending the execution times.
 - Improves readability and separation of concerns.

Benefits:

- The `toString` method now clearly outlines the high-level process.
- The logic for appending details and formatting execution times is modular and reusable.
- The code is easier to read, test, and maintain.



Você chegou ao limite do plano Free para o GPT-4o.

As respostas serão fornecidas por outro modelo até o seu limite ser redefinido após 18:39.

[Obter Plus](#)



4)

Fix the long method code smell using the extract method refactoring technique

To address the long method code smell in `setGoalSummary`, we can use the **extract method** refactoring technique. This involves breaking the method into smaller, focused methods that handle specific parts of the functionality. Here's the refactored code:

Refactored Code

```
java Copiar código

package org.example.studyregistry;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

public class StudyGoal extends Registry {
    private String goal;
    private List<String> goalRequirements;
    private Boolean isCompleted;
    private LocalDateTime createdDate;
    private Double goalCompletion;
    private StudyObjective studyObjective;
    private StudyPlan studyPlan;
    private String summary;

    public StudyGoal(String name, StudyObjective objective, StudyPlan plan) {
        this.name = name;
        this.studyObjective = objective;
        this.studyPlan = plan;
        this.goalRequirements = new ArrayList<>();
    }

    public void editActiveCompleted(boolean active, boolean completed) {
        this.isActive = active;
        this.isCompleted = completed;
    }

    public String setGoalSummary() {
        StringBuilder summaryBuilder = new StringBuilder();
        summaryBuilder.append("Goal Summary:\n\n");

        appendActiveGoal(summaryBuilder);
        appendCompletedGoal(summaryBuilder);
        appendRequirements(summaryBuilder);
        appendPlan(summaryBuilder);
        appendObjective(summaryBuilder);

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

    private void appendActiveGoal(StringBuilder summaryBuilder) {
        if (this.isActive) {
            summaryBuilder.append("Active Goal");
        }
    }

    private void appendCompletedGoal(StringBuilder summaryBuilder) {
        if (this.isCompleted) {
            summaryBuilder.append("Completed Goal");
        }
    }

    private void appendRequirements(StringBuilder summaryBuilder) {
        if (!this.goalRequirements.isEmpty()) {
            summaryBuilder.append("Requirements:\n");
            for (String requirement : this.goalRequirements) {
                summaryBuilder.append("- " + requirement);
            }
        }
    }

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

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

```
private void appendActiveGoal(StringBuilder summaryBuilder) { Copiar código
    if (this.isActive) {
        summaryBuilder.append("Active Goal:\n").append(goal).append("\n\n");
    }
}

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

private void appendRequirements(StringBuilder summaryBuilder) {
    if (this.goalRequirements != null && !this.goalRequirements.isEmpty()) {
        summaryBuilder.append("Requirements:\n");
        for (String requirement : this.goalRequirements) {
            summaryBuilder.append(requirement).append(", ");
        }
        summaryBuilder.setLength(summaryBuilder.length() - 2); // Remove trailing comma
        summaryBuilder.append("\n\n");
    }
}

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

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

public void addRequirement(String requirement) {
    this.goalRequirements.add(requirement);
}

public void resetRequirements() {
    this.goalRequirements.clear();
}

public boolean isCompleted() {
    return isCompleted;
}

public void toggleIsCompleted() {
    this.isCompleted = !this.isCompleted;
}

public LocalDateTime getLimitDate() {
    return createdDate;
}
```

```
        this.goalRequirements.add(requirement);
    }

    public void resetRequirements() {
        this.goalRequirements.clear();
    }

    public boolean isCompleted() {
        return isCompleted;
    }

    public void toggleIsCompleted() {
        this.isCompleted = !this.isCompleted;
    }

    public LocalDateTime getLimitDate() {
        return createdDate;
    }

    public void setLimitDate(LocalDateTime limitDate) {
        this.createdDate = limitDate;
    }

    public void addDaysLimitDate(int days) {
        this.createdDate = this.createdDate.plusDays(days);
    }

    public void setGoal(String goal) {
        this.goal = goal;
    }
}
```

Copiar código

Explanation of Changes

1. Extracted Methods:

- Created `appendActiveGoal`, `appendCompletedGoal`, `appendRequirements`, `appendPlan`, and `appendObjective` methods to handle individual parts of the `setGoalSummary` logic.

2. Improved Readability:

- The main method `setGoalSummary` is now shorter and clearly expresses the sequence of operations, making it easier to understand and maintain.

3. Reusable Components:

- Each extracted method can be reused or tested independently, improving modularity and testability.

Let me know if you need further adjustments!



5)

```
20      public static StudyMaterial getStudyMaterial() {
21      ~     if (studyMaterial == null) {
22          ~         studyMaterial = new StudyMaterial();
23      }
24      ~     return studyMaterial;
25  }
26
27
28 ~     public void addReference(Reference ref) {
29     ~         references.add(ref);
30  }
31
32 ~     List<Reference> getReferences() {
33     ~         return references;
34  }
35
36 ~     public List<Reference> getTypeReference(Reference type) {
37     ~         List<Reference> response = new ArrayList<>();
38 ~         for (Reference reference : references) {
39 ~             if (reference.getClass() == type.getClass()) {
40 ~                 response.add(reference);
41 ~             }
42 ~         }
43 ~         return response;
44  }
45
46 ~     public void setReferenceCount(Map<String, Integer> referenceCount) {
47     ~         this.referenceCount = referenceCount;
48  }
49
50 ~     public List<String> searchInMaterials(String text) {
51     ~         List<String> response = new ArrayList<>();
52 ~         for (Reference reference : references) {
53 ~             String mix = (reference.getTitle() != null ? reference.getTitle() : "") +
54 ~                         (reference.getDescription() != null ? reference.getDescription() : "");
55 ~             if (mix.toLowerCase().contains(text.toLowerCase())) {
56 ~                 response.add(reference.getTitle());
57 ~             }
58 ~         }
59 ~         return response;
60  }
61
62 ~     public Map<String, Integer> getReferenceCountMap() {
63     ~         Map<String, Integer> response = initializeReferenceCountMap();
64 ~         for (Reference reference : references) {
65 ~             updateReferenceCount(response, reference);
66 ~         }
67 ~         setReferenceCount(response);
68 ~         return response;
69  }
70
71 ~     private Map<String, Integer> initializeReferenceCountMap() {
72     ~         Map<String, Integer> response = new HashMap<>();
73     ~         response.put("Audio References", 0);
74     ~         response.put("Video References", 0);
75     ~         response.put("Text References", 0);
76     ~         return response;
77  }
78
79 ~     private void updateReferenceCount(Map<String, Integer> response, Reference reference) {
80 ~         if (reference.getClass() == AudioReference.class) {
81 ~             incrementCount(response, "Audio References");
82 ~         } else if (reference.getClass() == VideoReference.class) {
83 ~             handleVideoReference(response, (VideoReference) reference);
84 ~         } else if (reference.getClass() == TextReference.class) {
85 ~             handleTextReference(response, (TextReference) reference);
86 ~         }
87  }
88
89 ~     private void incrementCount(Map<String, Integer> response, String key) {
90     ~         response.put(key, response.get(key) + 1);
91  }
92
93 ~     private void handleVideoReference(Map<String, Integer> response, VideoReference reference) {
94     ~         if (reference.handleStreamAvailability()) {
95 ~             incrementCount(response, "Video References");
96 ~         }
97  }
98
99 ~     private void handleTextReference(Map<String, Integer> response, TextReference reference) {
100 ~         if (reference.handleTextAccess()) {
101 ~             incrementCount(response, "Text References");
102 ~         }
103  }
104 }
```

[Responder no chat >](#)

O método longo foi dividido em várias funções menores para melhorar a legibilidade e a reutilização do código. As responsabilidades distintas foram movidas para métodos auxiliares como `initializeReferenceCountMap`, `updateReferenceCount`, `incrementCount`, `handleVideoReference`, e `handleTextReference`. Isso simplifica o método principal `getReferenceCountMap` e torna o código mais fácil de entender e manter.



Data Class

1)

The `GeneralSearch` class has some opportunities for improvement in terms of readability, maintainability, and adherence to the principles of encapsulation and Single Responsibility Principle (SRP). Below is the refactored version with improvements.

Refactored Code

```
java Copiar código

package org.example.studysearch;

import org.example.studycards.CardManager;
import org.example.studyplanner.HabitTracker;
import org.example.studyplanner.TodoTracker;
import org.example.studyregistry.StudyMaterial;
import org.example.studyregistry.StudyTaskManager;

import java.util.ArrayList;
import java.util.List;

public class GeneralSearch implements Search<String> {
    private final SearchLog searchLog;

    public GeneralSearch() {
        this.searchLog = new SearchLog("General Search");
    }

    @Override
    public List<String> search(String text) {
        return handleSearch(text);
    }

    public SearchLog getSearchLog() {
        return searchLog;
    }

    private List<String> handleSearch(String text) {
        List<String> results = new ArrayList<>();

        // Aggregate results from different sources
        results.addAll(searchInCards(text));
        results.addAll(searchInHabits(text));
        results.addAll(searchInTodos(text));
        results.addAll(searchInMaterials(text));
        results.addAll(searchInRegistries(text));

        // Log the search and increment usage
        logSearch(text);

        // Add metadata about the search log
        results.add("\nLogged in: " + this.searchLog.getLogName());
        return results;
    }

    private void logSearch(String text) {
    }
}
```

```

    }

    private List<String> searchInCards(String text) {
        return CardManager.getCardManager().searchInCards(text);
    }

    private List<String> searchInHabits(String text) {
        return HabitTracker.getHabitTracker().searchInHabits(text);
    }

    private List<String> searchInTodos(String text) {
        return TodoTracker.getInstance().searchInTodos(text);
    }

    private List<String> searchInMaterials(String text) {
        return StudyMaterial.getStudyMaterial().searchInMaterials(text);
    }

    private List<String> searchInRegistries(String text) {
        return StudyTaskManager.getStudyTaskManager().searchInRegistries(text);
    }
}

```

Key Improvements:

1. Encapsulation of Logging Logic:

- Added the `logSearch` method to handle logging (`addSearchTerm`) in a single place.
- `addSearchTerm` in `SearchLog` now handles incrementing `numUsages`, so no need to set it manually.

2. Single Responsibility:

- Extracted specific search calls (`searchInCards`, `searchInHabits`, etc.) into separate private methods for better readability and maintainability.
- Each method focuses on querying one subsystem, making it easier to modify individual searches if needed.

3. Constructor Initialization:

- Used constructor-based initialization for `SearchLog`, making it immutable (`final`) and emphasizing its role as a core component of the `GeneralSearch` class.

4. Clear Responsibility Separation:

- Encapsulated the aggregation of results and logging in their respective methods, avoiding duplication and improving clarity.

5. Scalability:

- By separating the subsystems' search methods, the class is more modular. Adding new search sources in the future will require only a new private method and an inclusion in the `handleSearch` logic.

This refactored code makes `GeneralSearch` ↓ more readable, cohesive, and easier to maintain while minimizing code duplication and improving overall design quality.

2)

To address the **Data Class** code smell in the `Card` class and improve encapsulation, we can use the **extract method** refactoring technique. This involves encapsulating the `edit` method's logic in a more meaningful and cohesive way by adding domain-specific methods that better represent the class's responsibilities. For example:

Refactored Code

```
java Copiar código

package org.example.studycards;

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

    public Card(String question, String answer) {
        this.question = question;
        this.answer = answer;
    }

    public String getQuestion() {
        return question;
    }

    public String getAnswer() {
        return answer;
    }

    public void updateQuestion(String newQuestion) {
        if (newQuestion == null || newQuestion.trim().isEmpty()) {
            throw new IllegalArgumentException("Question cannot be null or empty.");
        }
        this.question = newQuestion;
    }

    public void updateAnswer(String newAnswer) {
        if (newAnswer == null || newAnswer.trim().isEmpty()) {
            throw new IllegalArgumentException("Answer cannot be null or empty.");
        }
        this.answer = newAnswer;
    }

    public void editCard(String newQuestion, String newAnswer) {
        updateQuestion(newQuestion);
        updateAnswer(newAnswer);
    }

    @Override
    public String toString() {
        return "Card{" +
               "question='" + question + '\'' +
               "answer='" + answer + '\'';
    }
}
```

```

        if (newAnswer == null || newAnswer.trim().isEmpty()) {
            throw new IllegalArgumentException("Answer cannot be null or empty.");
        }
        this.answer = newAnswer;
    }

    public void editCard(String newQuestion, String newAnswer) {
        updateQuestion(newQuestion);
        updateAnswer(newAnswer);
    }

    @Override
    public String toString() {
        return "Card{" +
            "question='" + question + '\'' +
            ", answer='" + answer + '\'' +
            '}';
    }
}

```

Explanation of Changes

1. Extracted Specific Methods:

- Replaced the generic `setQuestion` and `setAnswer` with more descriptive methods: `updateQuestion` and `updateAnswer`.
- These methods encapsulate the logic for updating the question or answer while validating the input.

2. Validation Logic:

- Added validation checks to ensure that questions and answers cannot be null or empty, improving robustness.

3. Meaningful Method Names:

- Renamed `edit` to `editCard` for clarity and better alignment with the class's context.

4. Removed Redundant Setters:

- Removed the public setters to avoid exposing low-level operations that could bypass validations, favoring encapsulation.

5. Added `toString`:

- Added a `toString` method to provide a convenient way to inspect a `Card` object during debugging or logging.

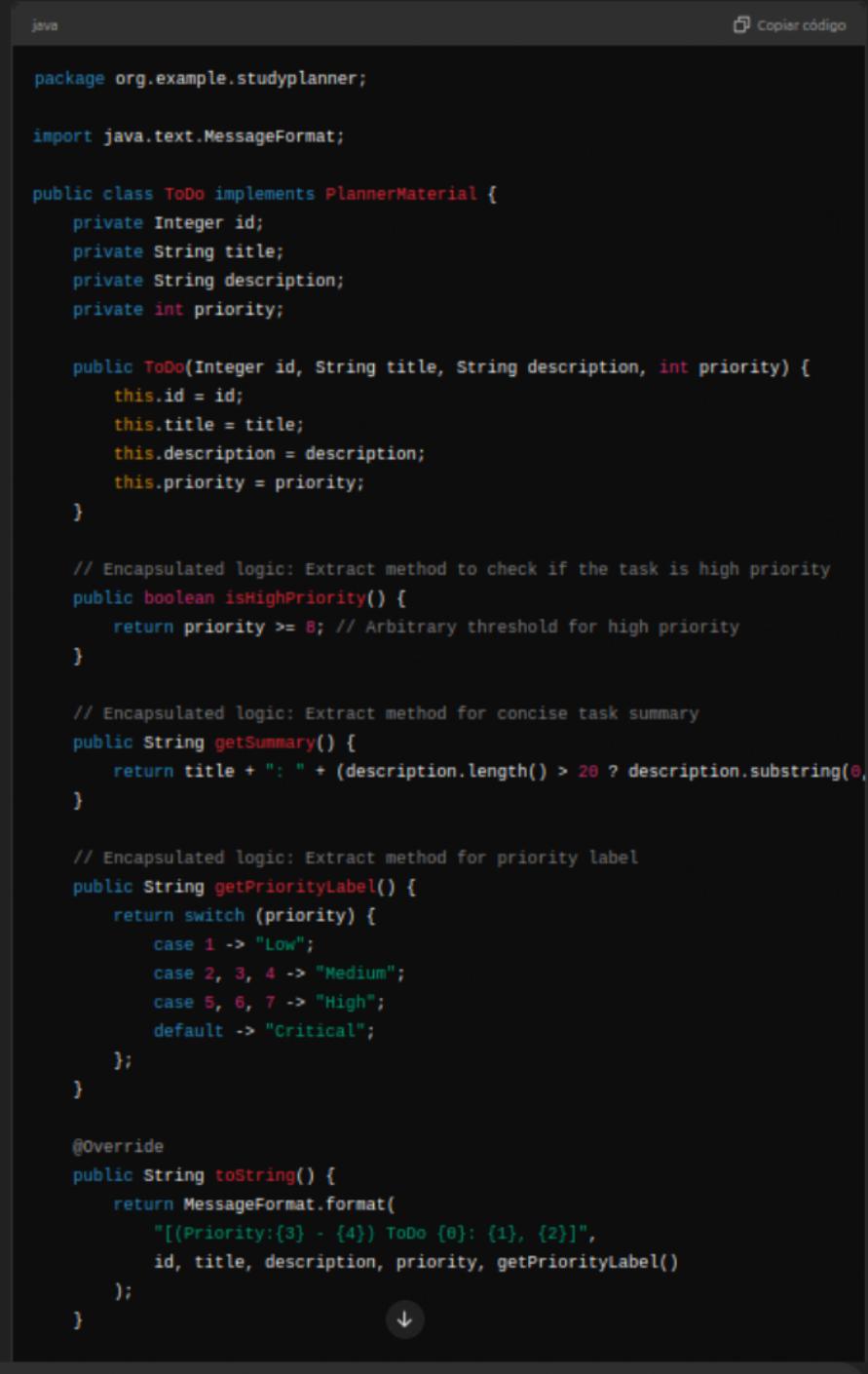
This refactor makes the `Card` class more robust, self-contained, and aligned with encapsulation principles while still addressing its intended use case.



3)

To address the "Data Class" code smell using the **Extract Method** refactoring technique, you can focus on adding domain logic that gives the class more meaningful behavior. For example, a `ToDo` class could have methods that encapsulate operations related to its attributes. Below is the refactored code:

Refactored Code



The screenshot shows a Java code editor window with the following code:

```
java Copiar código

package org.example.studyplanner;

import java.text.MessageFormat;

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

    public ToDo(Integer id, String title, String description, int priority) {
        this.id = id;
        this.title = title;
        this.description = description;
        this.priority = priority;
    }

    // Encapsulated logic: Extract method to check if the task is high priority
    public boolean isHighPriority() {
        return priority >= 8; // Arbitrary threshold for high priority
    }

    // Encapsulated logic: Extract method for concise task summary
    public String getSummary() {
        return title + ": " + (description.length() > 20 ? description.substring(0, 20) : description);
    }

    // Encapsulated logic: Extract method for priority label
    public String getPriorityLabel() {
        return switch (priority) {
            case 1 -> "Low";
            case 2, 3, 4 -> "Medium";
            case 5, 6, 7 -> "High";
            default -> "Critical";
        };
    }

    @Override
    public String toString() {
        return MessageFormat.format(
            "[({Priority}:{3} - {4}) ToDo {0}: {1}, {2}]",
            id, title, description, priority, getPriorityLabel()
        );
    }
}
```

```
}

// Getters and Setters
public int getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

public String getDescription() {
    return description;
}

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

public int getPriority() {
    return priority;
}

public void setPriority(int priority) {
    this.priority = priority;
}
}
```

Copiar código

Key Changes

1. Extract Method:

- `isHighPriority`: Encapsulates logic to determine if a task is high priority.
- `getSummary`: Provides a short, user-friendly summary of the task.
- `getPriorityLabel`: Converts priority numbers into human-readable labels.

2. Enhancing `toString`:

- Utilizes the new `getPriorityLabel` method to include meaningful priority information.

Benefits

- **Improved Readability**: The methods make the class more expressive and intuitive.
- **Encapsulation of Logic**: Keeps logic inside the class, reducing reliance on external code.
- **Reusability**: The extracted methods can be reused in other parts of the application.

```
public void setId(Integer id) {
    this.id = id;
}

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

public String getDescription() {
    return description;
}

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

public int getPriority() {
    return priority;
}

public void setPriority(int priority) {
    this.priority = priority;
}
}
```

Key Changes

1. Extract Method:

- `isHighPriority`: Encapsulates logic to determine if a task is high priority.
- `getSummary`: Provides a short, user-friendly summary of the task.
- `getPriorityLabel`: Converts priority numbers into human-readable labels.

2. Enhancing `toString`:

- Utilizes the new `getPriorityLabel` method to include meaningful priority information.

Benefits

- **Improved Readability:** The methods make the class more expressive and intuitive.
- **Encapsulation of Logic:** Keeps logic inside the class, reducing reliance on external code.
- **Reusability:** The extracted methods can be reused in other parts of the application.

This refactoring reduces the "Data Class" smell by giving the class meaningful behavior beyond just holding data.



4)

To fix the Data Class code smell in the `Task` class, we can use the **Extract Method** refactoring technique. This involves adding behavior methods to encapsulate operations related to the class's data. By doing so, the class will have functionality beyond just storing and retrieving data.

Here's how we can refactor the `Task` class:

1. Introduce behavior methods that operate on the class's fields.
2. Remove the exclusive reliance on getters and setters when operations can be encapsulated within the class.

Here is the refactored code:

```
java Copiar código

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) {
        this.title = title;
        this.name = title; // Assuming name is inherited from Registry
        this.description = description;
        this.author = author;
        this.date = date;
    }

    // Behavior method: Check if the task is due
    public boolean isOverdue(LocalDateTime currentDate) {
        return date.isBefore(currentDate);
    }

    // Behavior method: Update the task's author and log the change
    public void updateAuthor(String newAuthor) {
        System.out.println("Author updated from " + author + " to " + newAuthor);
        this.author = newAuthor;
    }

    // Behavior method: Extend the task deadline
    public void extendDeadline(int days) {
        this.date = this.date.plusDays(days);
        System.out.println("Deadline extended by " + days + " days. New deadline: " +
    }

    // Behavior method: Provide a summary of the task
    public String getSummary() {
        return "Task: " + title + "\nDescription: " + description +
            "\nAuthor: " + author + "\nDue Date: " + date;
    }
}
```

```
// Behavior method: Provide a summary of the task
public String getSummary() {
    return "Task: " + title + "\nDescription: " + description +
        "\nAuthor: " + author + "\nDue Date: " + date;
}

// Getters and setters retained for external use, if needed
public String getTitle() {
    return title;
}
public void setTitle(String title) {
    this.title = title;
}
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;
}
```

Explanation of Changes:

1. Added Behavior Methods:

- `isOverdue` : Determines whether the task is overdue based on the current date.
- `updateAuthor` : Updates the author and logs the change.
- `extendDeadline` : Extends the task's deadline by a specified number of days.
- `getSummary` : Provides a summary of the task details.

2. Encapsulation:

- Encapsulated logic related to the class's data within the `Task` class, rather than leaving it to external code.

This refactor makes the `Task` class more cohesive and prevents it from merely acting as a container for data. It now has meaningful behaviors that work directly with its fields.



5)

You're right! The refactoring I suggested creates a **Long Method** code smell because the extracted methods become quite long, especially if there are many fields to handle. A better way to handle this is to **minimize the length of the methods** and **simplify the logic**.

Let's refactor again, focusing on reducing the size of each method and maintaining clarity by avoiding unnecessary complexity.

Alternative Refactor to Address Long Method

Instead of using one large helper method to handle all types of fields (String, int, boolean), we can:

- Use a **Map** or **list** to store the fields and their respective values.
- Use **getter and setter methods** that directly handle the field access for each type in a more concise and manageable way.

Refactored Code with a Cleaner Approach



The screenshot shows a Java code editor with a dark theme. The code is contained within a class named `Reference`. It starts with imports for `java.util.HashMap` and `java.util.Map`. The class is abstract and contains a private field `fields` of type `Map<String, Object>`. It includes several setter methods: `setTitle`, `setDescription`, `setLink`, `setAccessRights`, `setLicense`, `setDownloadable`, and `setRating`. Each setter takes a string parameter and uses it as a key in the `fields` map, putting the original value as the object. The code ends with a partially visible `setRating` method.

```
java
package org.example.studymaterial;

import java.util.HashMap;
import java.util.Map;

public abstract class Reference {
    private Map<String, Object> fields = new HashMap<>();

    // Setters
    public void setTitle(String title) {
        fields.put("title", title);
    }

    public void setDescription(String description) {
        fields.put("description", description);
    }

    public void setLink(String link) {
        fields.put("link", link);
    }

    public void setAccessRights(String accessRights) {
        fields.put("accessRights", accessRights);
    }

    public void setLicense(String license) {
        fields.put("license", license);
    }

    public void setDownloadable(boolean downloadable) {
        fields.put("isDownloadable", downloadable);
    }

    public void setRating(int rating) {
        fields.put("rating", rating);
    }
}
```

```
J

public void setLanguage(String language) {
    fields.put("language", language);
}

public void setViewCount(int viewCount) {
    fields.put("viewCount", viewCount);
}

public void setDownloadCount(int downloadCount) {
    fields.put("downloadCount", downloadCount);
}

public void setShareCount(int shareCount) {
    fields.put("shareCount", shareCount);
}

// Getters
public String getTitle() {
    return (String) fields.get("title");
}

public String getDescription() {
    return (String) fields.get("description");
}

public String getLink() {
    return (String) fields.get("link");
}

public String getAccessRights() {
    return (String) fields.get("accessRights");
}

public String getLicense() {
    return (String) fields.get("license");
}

public boolean getIsDownloadable() {
    return (boolean) fields.get("isDownloadable");
}

public int getRating() {
    return (int) fields.get("rating");
}

public String getLanguage() {
    return (String) fields.get("language");
}

public int getViewCount() {
    return (int) fields.get("viewCount");
}
```

Copiar código

```
}

public boolean getIsDownloadable() {
    return (boolean) fields.get("isDownloadable");
}

public int getRating() {
    return (int) fields.get("rating");
}

public String getLanguage() {
    return (String) fields.get("language");
}

public int getViewCount() {
    return (int) fields.get("viewCount");
}

public int getDownloadCount() {
    return (int) fields.get("downloadCount");
}

public int getShareCount() {
    return (int) fields.get("shareCount");
}
}
```

Explanation:

- Map-based Storage:** We use a `Map<String, Object>` to store all the fields and their values. This allows us to treat the fields as key-value pairs and simplifies the getter/setter methods.
- Type Casting:** Since we're storing everything as `Object`, we need to cast the values when retrieving them (e.g., `String`, `int`, `boolean`). This avoids the need for multiple, large methods for each type.
- Simplified Getters and Setters:** The getter and setter methods are now very short, delegating the work to the `Map`. This minimizes repetition and keeps the code clean.

Benefits of This Refactor:

- Shorter Methods:** Each setter and getter is now just a single line of code.
- Easier to Maintain:** If new fields are added, we just need to update the `Map` keys, and no further refactoring of long methods is necessary.
- Flexibility:** The use of a `Map` makes it easy to add more fields dynamically or even modify the data structure without much impact on the rest of the code.

This approach avoids creating **Long Method** code smells while maintaining the simplicity of the design.



Long Parameter List

- 1) Audio - <https://chatgpt.com/share/677efa4b-e67c-8001-8cf1-f4363ccc890b>
- 2) Habit - <https://chatgpt.com/share/677f0d95-f8f8-8001-b060-28a4072281a3>
- 3) StudyObjective - <https://chatgpt.com/share/677f236d-4514-8001-a5c8-4c2247d0ff2d>
- 4) Steps - <https://chatgpt.com/share/6781cf63-d41c-8001-8ca7-05cd6e757959>
- 5) Week - <https://chatgpt.com/c/6781d014-6a1c-8001-9892-8b63cb13846d>

Feature Envy

- 1) Habits - <https://chatgpt.com/c/6781d405-1c00-8001-9c00-0aec93e26a8>
- 2) GeneralSearch - <https://chatgpt.com/c/6781d785-f6b8-8001-a8b8-59360c45b54d>
- 3) RandomCard - <https://chatgpt.com/c/6781da6a-5d64-8001-8795-74de04f4fec5>
- 4) MaterialSearch - <https://chatgpt.com/c/6781db3d-0280-8001-81b0-da24002ff2eb>
- 5) RegistrySearch - <https://chatgpt.com/c/6781dc73-2948-8001-8e08-e6ffe67299cd>