

Documentação da Refatoração do Projeto

Nome dos Alunos:

Maria Eduarda Medeiro Porto- 824144948
Matheus Alves Santana- 824144952
Matheus Henrique da Costa e Silva – 82410661

2025

RESUMO

Este projeto é uma refatoração incremental do código legado do [GildedRose-Refactoring-Kata](#), um exercício clássico para ilustrar problemas em bases de código antigas. O código original, escrito em Java, gerencia a atualização da qualidade e prazo de venda (**sellIn**) de itens em uma loja fictícia, mas apresenta diversos code smells, como métodos excessivamente longos, duplicação de código e falta de estrutura orientada a objetos (OO). A refatoração transforma esse código procedural em um sistema OO limpo, testável e extensível, aplicando boas práticas de desenvolvimento de software.

Índice

Introdução.....	4
Objetivos e Contexto.....	4
Benefícios e Adequação aos Critérios.....	4
Tecnologias e Patterns Utilizados.....	5
Metodologia.....	6
Principais Deficiências Encontradas.....	6
Métodos com lógica condicional extensa.....	6
Nomes e responsabilidades pouco explícitos.....	6
Coesão de classe e acoplamento elevado.....	6
Histórico de Commits.....	7
Commit 1: Adicionar Constantes para Strings Mágicas	7
Commit 2: Extração de Métodos para Tipos de Itens	7
Commit 4- subida de testes unitários incorretos.....	8
Commit 5- Correção e delete de erros.....	8
Commit 6: Implementar Testes Unitários com JUnit.....	8
Commit 7- Limpeza “final 1” e término da documentação	9
Justificativa para as Mudanças Feitas na Refatoração.....	10
Legibilidade: Código Mais Organizado, Nomes de	10
Variáveis e Funções Claros e Significativos	
. Estrutura: Redução de Repetições,	10
Modularização Adequada e Menor Complexidade	
Comentários e Documentação: Comentários	11
Explicativos Apenas Quando Necessário e Apropriada	
Documentação	
Boas Práticas: Aplicação de Princípios.....	11
SOLID, DRY, KISS e YAGNI Quando	

Aplicável

Testes: Geração de Testes Unitários para	11
Garantir que o Código Continua Funcionando	
Corretamente Após a Refatoração	
Descrição dos Testes Unitários Implementados.....	12
NormalItemStrategyTest.java.....	12
AgedBrieStrategyTest.java.....	12
BackstagePassesStrategyTest.java.....	13
SulfurasStrategyTest.java.....	13
GildedRoseTest.java.....	13
Conclusão sobre a Importância do Clean Code na Manutenção	14
de Software	
Clean Code.....	14
Importância na Manutenção de Software.....	14

1. Introdução

O repositório escolhido (GildedRose-Refactoring-Kata) é um exercício clássico e considerado “simples” de refatoração de código, projetado para ilustrar problemas comuns em bases de código legadas. O código principal está localizado na classe **GildedRose**, que gerencia a atualização da qualidade de itens em uma loja fictícia.

O repositório original está em:

<https://github.com/emilybache/GildedRose-Refactoring-Kata/tree/main/Java>

E o repositório do grupo:

<https://github.com/mariamempor/gilded-rose-clean-code-A3>

Obs: A data da alimentação do repositório , bem como as dos commits pode divergir com atraso por conta de um problema que tivemos no repositório original do trabalho e no projeto.

1.2. Objetivos e Contexto

O projeto foi desenvolvido para atender aos critérios de uma atividade acadêmica, focando em:

- Identificar e corrigir deficiências em código legado.
- Aplicar princípios de Clean Code (e.g., SOLID, DRY, KISS, YAGNI).
- Implementar pelo menos um Design Pattern.
- Garantir manutenibilidade, legibilidade e testabilidade.

1.3. Benefícios e Adequação aos Critérios

- **Legibilidade:** Nomes claros (e.g., `updateQuality()`) e estrutura organizada facilitam leitura.
- **Estrutura:** Redução de repetições e complexidade via modularização e patterns.
- **Comentários e Documentação:** Javadoc explicativo apenas quando necessário, evitando excesso.
- **Boas Práticas:** Aplicação de SOLID (e.g., SRP, OCP), DRY, KISS e YAGNI, com Strategy Pattern para extensibilidade.

- **Testes:** Testes unitários garantem funcionamento correto pós-refatoração, prevenindo regressões.
- **Versionamento:** Commits incrementais registrados em repositório GitHub público, demonstrando progresso.

Essas mudanças transformam manutenção de uma tarefa complexa em algo eficiente, reduzindo bugs e facilitando evoluções.

1.4. Tecnologias e Patterns Utilizados

- **Linguagem:** Java
- **Frameworks:** JUnit para testes.
- **Patterns:** Strategy (para comportamentos de itens) e Factory (para criação de estratégias).
- **Princípios:** SOLID, DRY, KISS, YAGNI.
- **Ferramentas:** Maven para build e dependências.

2. Metodologia

- A análise foi conduzida via inspeção estática do repositório (estrutura de arquivos, nomes de classes/métodos, organização de pacotes) e leitura de trechos de código, ao longo dos dias 22/10 e 26/10.
- Os critérios de avaliação incluem: legibilidade, modularidade, coesão, acoplamento, clareza de nomes, complexidade condicional, presença de código morto ou redundante.

3. Principais Deficiências Identificadas

3.1. Métodos com lógica condicional extensa

Foi observado que o código da implementação original do exercício contém diversas condicionais do tipo if/else if/else, responsáveis por tratar múltiplos casos de itens (por exemplo: itens “normais”, “Aged Brie”, “Backstage passes”, “Sulfuras”, etc.). Esta estrutura complexa provoca:

- Dificuldade de manutenção (mudança de regras exige alterar muitos ramos).
- Violação do princípio Aberto-Fechado (Open/Closed): para adicionar um novo tipo de item, provavelmente será necessário modificar o método existente.
- Potencial para erros acidentais pela dispersão da lógica.

3.2. Nomes e responsabilidades pouco explícitos

Alguns métodos e variáveis não refletiam de forma clara a intenção ou o comportamento que implementam. Incluindo variáveis genéricas, métodos com múltiplas responsabilidades (“update all items”, “calculate quality and sellIn”).

Isso reduz a legibilidade e torna mais custoso entender “o que o código faz” antes de “como”.

3.3. Coesão de classe e acoplamento elevado

A classe principal que controla a atualização de itens assumia várias tarefas (reduzir sellIn, ajustar quality, tratar casos especiais). Esse tipo de “god class” concentra lógica diversificada, indicando baixa coesão. Além disso, acoplamentos explícitos entre itens e o mecanismo de atualização dificultam extensões ou modificações isoladas.

Em síntese, o repositório escolhido pelo grupo oferece uma boa didática, mas não atende plenamente aos critérios de “clean code” em sua forma inicial. As deficiências mais significativas, portanto, são: condicionais extensas, baixa coesão, nomes pouco explícitos, falta de modularização.

4. Histórico de commits

Descrição dos commits de refatoração do projeto.

4.1. Commit 1: Adicionar Constantes para Strings Mágicas 31/10

Foram eliminadas strings mágicas definindo constantes em uma classe utilitária. Melhorou a legibilidade e reduziu erros de digitação. Atende a DRY e KISS.

Mudanças:

- Criação **src/main/java/com/gildedrose/ItemNames.java** com constantes.
- Atualização **GildedRose.java** para usar essas constantes.

4.2. Commit 2: Extração de Métodos para Tipos de Itens 01/11

Quebra do método longo em métodos menores. Redução da complexidade e aplicação SRP. Legibilidade melhorada.

Modificado: **src/main/java/com/gildedrose/GildedRose.java** (método principal refatorado e métodos extraídos)

4.3. Commit 3: Introdução do Strategy Pattern para Comportamentos de Itens 02/11

Descrição: Aplicação do OO com Strategy Pattern (interface e subclasses). Permitiu extensibilidade (OCP/SOLID). Redução da duplicação (DRY). Estrutura modular.

Arquivos modificados/criados:

- **Novo arquivo:** **src/main/java/com/gildedrose/UpdateStrategy.java** (interface, para usar **ItemNames**)
- **Novo arquivo:** **src/main/java/com/gildedrose/NormalItemStrategy.java**
- **Novo arquivo:** **src/main/java/com/gildedrose/AgedBrieStrategy.java**
- **Novo arquivo:** **src/main/java/com/gildedrose/BackstagePassesStrategy.java**

- **Novo arquivo:** `src/main/java/com/gildedrose/SulfurasStrategy.java`
- **Novo arquivo:** `src/main/java/com/gildedrose/ItemStrategyFactory.java`
- **Modificado:** `src/main/java/com/gildedrose/GildedRose.java`

4.4. Commit 4- subida de testes unitários incorretos 07/11

Descrição: Foram implementados Testes Unitários que depois acabaram se mostrando ineficazes para o propósito do projeto, necessitando a correção posterior.

4.5. Commit 5- Correção e delete de erros 10/11

Descrição: Correção e delete dos arquivos não úteis do projeto. Correção dos Testes Unitários

4.6. Commit 6: Implementar Testes Unitários com JUnit 13/11

Descrição: Substituição dos testes approval por unitários isolados. Garantindo o funcionamento (testes) e adiciona validação. Atende a critérios de testes e boas práticas.

Arquivos modificados/criados:

Modificado: `src/main/java/com/gildedrose/UpdateStrategy.java` (adição de validação genérica, corrigido para **Item**)

Novos arquivos: `src/test/java/com/gildedrose/NormalItemStrategyTest.java`

`src/test/java/com/gildedrose/AgedBrieStrategyTest.java`

`src/test/java/com/gildedrose/BackstagePassesStrategyTest.java`

`src/test/java/com/gildedrose/SulfurasStrategyTest.java`

Modificado: `src/test/java/com/gildedrose/GildedRoseTest.java` (corrigido para usar **Item** e manter o approval test original)

4.7. Commit 7- Limpeza “final 1” e término da documentação 14/11

Descrição: Limpeza final 1, adição de comentários, Garante YAGNI. Código OO completo com patterns e testes.

Modificado: `src/main/java/com/gildedrose/GildedRose.java`

Modificado: `src/main/java/com/gildedrose/UpdateStrategy.java`

5. Justificativa para as Mudanças Feitas na Refatoração

A refatoração do código original foi realizada de forma incremental, com commits no GitHub entre setembro e novembro, para transformar um código procedural legado em um código orientado a objetos (OO) limpo e sustentável. O foco foi em aplicar Design Patterns (Strategy e Factory), reduzir code smells e garantir manutenibilidade, sem adicionar funcionalidades desnecessárias (YAGNI).

Link do Código Refatorado do projeto, Branch refatoracaoDoCodigo:

<https://github.com/mariamempor/gilded-rose-clean-code-A3/tree/refatoracaoDoCodigo>

Pasta de códigos: src/main/java/com/gildedrose

5.1. Legibilidade: Código Mais Organizado, Nomes de Variáveis e Funções Claros e Significativos

Implementação: Substituimos as strings mágicas por constantes em **ItemNames.java**. Nomes de métodos e classes foram escolhidos para serem descritivos (ex: **updateQuality()** em vez de algo genérico, **NormalItemStrategy** para itens normais). Variáveis como **item.quality** e **item.sellIn** mantiveram nomes claros do original.

Por quê: Torna o código autoexplicativo, facilitando leitura e manutenção. Evita ambiguidades, atendendo diretamente ao critério sem necessidade de comentários excessivos.

5.2. Estrutura: Redução de Repetições, Modularização Adequada e Menor Complexidade

Implementação: Quebramos o método **updateQuality()** original (longo e complexo) em métodos menores (**updateNormalItem()**) e, depois, em classes separadas via Strategy Pattern. Usamos Factory Pattern em **ItemStrategyFactory** para centralizar criação de estratégias.

Por quê: Eliminamos a duplicação (DRY) em lógica de atualização (ex: decremento de **sellIn** repetido). Modularizamos as responsabilidades (SRP), tornando o código mais coeso e fácil de navegar. Isso reduziu bugs e facilitou extensões.

5.3. Comentários e Documentação: Comentários Explicativos Apenas Quando Necessário e Apropriada Documentação

Implementação: Adicionamos Javadoc mínimo em classes principais (**GildedRose.java** e **UpdateStrategy.java**) para descrever propósito geral. Comentários inline foram usados apenas em lógica não óbvia (ex: "/* Sulfuras não muda*/" em **SulfurasStrategy.java**). Evitamos o uso de comentários redundantes, pois o código é expressivo.

Por quê: Segue YAGNI, adicionando documentação apenas onde esclarece ou comportamentos complexos. Mantemos o código limpo, sem poluir com explicações desnecessárias, atendendo ao critério de "apenas quando necessário".

5.4. Boas Práticas: Aplicação de Princípios SOLID, DRY, KISS e YAGNI Quando Aplicável

Implementação: Aplicamos o SOLID (SRP via métodos/classes separadas, OCP via Strategy Pattern para extensibilidade sem modificações, DIP via interface **UpdateStrategy**). DRY foi usado para eliminar repetições (lógica de validação centralizada em **validate()**). KISS simplificou estruturas (Factory em vez de switch gigante). YAGNI evitou adições extras (apenas validação básica de limites).

Por quê: Garantiu que o código ficasse sustentável e extensível. Por exemplo, adicionar um novo item requer apenas uma nova estratégia e linha no Factory, sem alterar código existente. Isso reduz acoplamento e aumenta reutilização.

5.5. Testes: Geração de Testes Unitários para Garantir que o Código Continua Funcionando Corretamente Após a Refatoração

Implementação: Criamos testes unitários isolados com JUnit para cada estratégia (**NormalItemStrategyTest.java**), cobrindo casos normais e edge (antes/depois de **sellIn**). Mantivemos o teste approval original em **GildedRoseTest.java** para compatibilidade. Adicionamos a validação de limites (qualidade 0-50) para prevenir bugs.

Por quê: Garantiu que refatorações não quebrem funcionalidade. Testes isolados facilitam debugging e CI/CD, atendendo ao critério de funcionamento correto pós-refatoração.

6. Descrição dos Testes Unitários Implementados

Os testes unitários foram implementados com JUnit, focando em isolamento (cada teste verifica uma estratégia específica). Eles garantem funcionamento correto pós-refatoração, substituindo parcialmente o teste approval para maior precisão. Seguem a lógica de ser específicos, objetivos e claros, alinhando-se aos critérios de testes.

Link do Repositório, Branch de testes-unitarios:

<https://github.com/mariamempor/gilded-rose-clean-code-A3/tree/teste-unitarios>

Pasta de testes: src/test/java/com/gildedrose

6.1. NormalItemStrategyTest.java

- **Descrição:** Testa a estratégia para itens normais, que decrementam qualidade e sellIn.
- **Casos:** **testNormalItemBeforeSellIn()** (qualidade decrementa 1, sellIn decrementa 1); **testNormalItemAfterSellIn()** (qualidade decrementa 2 após sellIn expirar).
- **Por quê:** Cobre lógica básica e edge (sellIn negativo), garantindo DRY e funcionamento.

6.2. AgedBrieStrategyTest.java

- **Descrição:** Testa itens que aumentam qualidade com o tempo.
- **Casos:** **testAgedBrieBeforeSellIn()** (qualidade incrementa 1); **testAgedBrieAfterSellIn()** (incrementa 2 após sellIn expirar).
- **Por quê:** Verifica regras especiais (aumento em vez de decremento), prevenindo bugs. Alinha a KISS com testes simples.

6.3. BackstagePassesStrategyTest.java

- **Descrição:** Testa passes que aumentam qualidade baseado em sellIn restante.
- **Casos:** **testBackstagePassesMoreThan10Days()** (incremento básico); **testBackstagePassesAfterSellIn()** (qualidade zera após expiração).
- **Por quê:** Cobre lógica complexa (incrementos variáveis), garantindo validação de limites.

6.4. SulfurasStrategyTest.java

- **Descrição:** Testa itens imutáveis.
- **Casos:** `testSulfurasNoChange()` (qualidade e sellIn permanecem iguais).
- **Por quê:** Confirma ausência de mudanças, aplicando YAGNI (teste mínimo para caso especial).

6.5. GildedRoseTest.java (Mantido com Approval)

- **Descrição:** Teste approval original para compatibilidade, com referências aos unitários.
- **Casos:** `testUpdateQuality()` (verifica nome após atualização).
- **Por quê:** Mantém legado, mas unitários isolados garantem cobertura real. Adequado para integração.

7. Conclusão sobre a Importância do Clean Code na Manutenção de Software

7.1. Clean Code

Clean Code refere-se a práticas de desenvolvimento que priorizam código legível, simples, testável e sustentável. Baseado em princípios como SOLID (ex: Responsabilidade Única, Aberto-Fechado), DRY (Don't Repeat Yourself), KISS (Keep It Simple, Stupid) e YAGNI (You Aren't Gonna Need It), ele evita code smells como métodos longos, duplicação e acoplamento excessivo. No contexto da refatoração do GildedRose, transformamos um código procedural legado em um orientado a objetos, aplicando patterns como Strategy e Factory, resultando em um código mais expressivo e modular.

7.2. Importância na Manutenção de Software

A manutenção de software envolve correções, evoluções e adaptações contínuas, e o Clean Code é essencial para torná-la eficiente e econômica. Sem ele, o código se torna um "déficit técnico" que aumenta custos e riscos. Os principais benefícios, ilustrados pela refatoração realizada:

- **Redução de Bugs e Facilitação de Debugging:** Código limpo, com nomes claros e testes unitários (como os implementados para cada estratégia no GildedRose), permite identificar e corrigir problemas rapidamente. No código original, if-else aninhados e duplicação tornavam bugs difíceis de rastrear, após refatoração, validações centralizadas erros comuns.
- **Melhoria na Extensibilidade e Evolução:** Princípios como OCP (Aberto-Fechado) permitem adicionar funcionalidades (ex: novos tipos de itens) sem alterar código existente. No GildedRose, o Strategy Pattern tornou trivial adicionar "Conjured" items, evitando regressões. Isso reduz tempo de manutenção em projetos de longo prazo, onde mudanças frequentes são inevitáveis.
- **Aumento da Produtividade e Colaboração:** Código legível e bem estruturado acelera onboarding de novos desenvolvedores e revisões. A refatoração aplicou modularização (métodos curtos, classes coesas), atendendo aos critérios de legibilidade e estrutura do professor, facilitando trabalho em equipe e reduzindo conflitos em merges.
- **Minimização de Débito Técnico:** Code smells como duplicação e complexidade alta acumulam "dívidas" que crescem com o tempo. A refatoração eliminou isso via DRY e KISS, resultando em código sustentável. Testes unitários garantem que mudanças não quebrem

funcionalidades, alinhando-se ao critério de testes e prevenindo retrabalho custoso.

No GildedRose, o código original tinha smells como "Long Method" e "Duplicated Code", dificultando manutenção. Após aplicar Clean Code , o código ficou modular, testável e extensível. Isso demonstrou que Clean Code não é opcional: transforma manutenção de uma tarefa dolorosa em uma eficiente, provando seu valor em projetos reais.

Clean Code é fundamental para a manutenção sustentável de software, transformando desafios em oportunidades. Ele promove qualidade, reduz riscos e aumenta eficiência, como evidenciado na refatoração incremental do GildedRose. Desenvolvedores devem priorizá-lo desde o início, aplicando princípios e patterns, para garantir software que evolua com o tempo sem acumular dívidas. Em resumo, Clean Code não é luxo, mas necessidade para equipes e projetos de sucesso.