Impacto dos Padrões de Projeto na Escalabilidade: Uma Análise Quantitativa de Custo-Benefício e Manutenibilidade em Java

Fernando Afonso de Souza Dias , Esdras Altivo Batista Corrêa

Faculdade de Ciências Empresariais - Universidade FUMEC
Rua Cobre, 200 - Cruzeiro, Belo Horizonte - MG, 30310-190 - Brasil

a226866341@fumec.edu.br , a226844158@fumec.edu.br

Resumo. *Este artigo investiga a aplicação prática de padrões de projeto no desenvolvimento de sistemas de processamento de dados. Utiliza-se como estudo de caso o SheetReader, um motor Java para importação de planilhas, para realizar uma análise comparativa entre uma implementação inicial (Baseline) e uma versão reestruturada (Refatorada). Partindo dos desafios recorrentes em sistemas de transformação de dados (como acoplamento elevado e complexidade na extensibilidade), é demonstrado como a implementação estruturada dos padrões Factory Method (para suporte a diferentes planilhas), Template Method (para definir o esqueleto invariante de um algoritmo, delegando a implementação de etapas específicas para subclasses) e Strategy (para encapsular diferentes lógicas de mapeamento, tornando-as intercambiáveis e desacoplando-as do leitor principal) impactou positivamente a qualidade do código. Os resultados validam a tese de que padrões de projeto, quando aplicados contextualmente, otimizam não apenas a estrutura interna, mas também indicadores operacionais em sistemas de processamento de dados. O estudo oferece um modelo replicável para projetos similares.*
Palavras-chave: *design patterns; engenharia de software; coesão; acoplamento; refatoração.*

Abstract. *This article investigates the practical application of design patterns in the development of data processing systems. As a case study, it uses the SheetReader, a Java engine for spreadsheet import, to conduct a comparative analysis between an initial implementation (Baseline) and a restructured version (Refactored). Starting from the recurring challenges in data transformation systems (such as high coupling and extensibility complexity), it demonstrates how the structured implementation of the Factory Method (for supporting different spreadsheets), Template Method (to define the invariant skeleton of an algorithm while delegating the implementation of specific steps to subclasses), and Strategy (to encapsulate different mapping logics, making them interchangeable and decoupling them from the main reader) patterns positively impacted code quality. The results validate the thesis that design patterns, when applied contextually, optimize not only the internal structure but also operational indicators in data processing systems. The study offers a replicable model for similar projects.*
Key-words: *design patterns; software engineer; cohesion; coupling; refactoring.*

1. Introdução

No desenvolvimento de *software*, existem diversos paradigmas de programação. De acordo com Van Roy e Haridi [2004], um paradigma determina os blocos de construção que podem ser usados para compor um programa e as maneiras pelas quais eles podem ser combinados.

Um dos mais influentes na atualidade é o paradigma de Orientação a Objetos, que, segundo Booch [1994], organiza programas como coleções cooperativas de objetos, fundamentando-se em quatro pilares: Abstração, Encapsulamento, Herança e Polimorfismo.

Nesse contexto, a qualidade do projeto de *software* depende da forma como as classes se relacionam, o que nos leva aos conceitos centrais de acoplamento e coesão. Propostos originalmente por Stevens et al. [1974], esses conceitos estabelecem que um bom design deve maximizar a coesão, o grau em que os elementos de um módulo pertencem a uma responsabilidade única e bem definida, e minimizar o acoplamento, que é a medida de dependência entre módulos distintos. Como reforça Pressman [2016], a alta coesão, combinada com baixo acoplamento, conduz a sistemas mais robustos, reutilizáveis e fáceis de evoluir.

Para formalizar a busca por esse design de qualidade, Martin [2002] compilou um conjunto de cinco diretrizes que se tornaram fundamentais: os princípios SOLID, Responsabilidade Única (SRP), Aberto/Fechado (OCP), Substituição de Liskov (LSP), Segregação de Interface (ISP) e Inversão de Dependência (DIP), fornecem um vocabulário e um guia prático para que arquitetos e desenvolvedores estruturem o software de forma a atingir diretamente o baixo acoplamento e a alta coesão. Eles representam a base teórica que viabiliza a construção de sistemas flexíveis e manuteníveis.

Nesse cenário, os padrões de projeto (*design patterns*) surgem como soluções recorrentes para problemas comuns de design de *software*. Segundo Gamma et al. [1995], os padrões não são algoritmos prontos, mas descrições de estruturas e interações que resolvem problemas de maneira eficaz e reutilizável. A utilização de padrões contribui diretamente para o baixo acoplamento e a alta coesão, pois suas estruturas inerentemente promovem a organização modular e a flexibilidade. Além disso, como reforça Freeman et al. [2004], os padrões de projeto funcionam como um vocabulário compartilhado entre desenvolvedores, o que favorece a comunicação e a manutenção de sistemas complexos.

Os padrões de projeto utilizados no desenvolvimento e na refatoração do estudo de caso são detalhados na Tabela 1, juntamente com seus conceitos e aplicação específica no sistema *SheetReader*.

Tabela 1. Padrões de Projeto Utilizados e Seus Conceitos

Padrão	Significado	Aplicação no Projeto (*SheetReader*)
Factory Method	Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar.	Solução utilizada no *SheetReader* para criar instâncias de importadores compatíveis com múltiplos formatos de planilhas.
Strategy	Define uma família de algoritmos, encapsula cada um e os torna intercambiáveis.	Desacopla o leitor de planilhas (*Contexto*) das regras específicas de mapeamento (*Estratégias*), aderindo ao *Princípio Aberto/Fechado*.
Template Method	Define o esqueleto de um algoritmo em um método, mas delega a implementação de algumas etapas para subclasses.	Descrito na seção de Resultados como parte da refatoração do código, padronizando fluxos de processamento; retomado na conclusão como benefício à manutenção.

Fonte: Adaptado de Gamma, Helm, Johnson, and Vlissides (1995).

Nota-se uma relação complementar entre o *Template Method* e o *Strategy*: ambos gerenciam variações de algoritmos. Enquanto o *Template Method* utiliza herança para definir o esqueleto do algoritmo e permitir variações nas subclasses, o *Strategy* utiliza composição para encapsular algoritmos inteiros e torná-los intercambiáveis, alterando o comportamento do objeto independentemente dos clientes que o utilizam.

Diante do exposto, formula-se a seguinte questão de pesquisa: quais as vantagens de utilizar padrões de projeto e de que forma eles contribuem para alcançar alta coesão e baixo acoplamento em sistemas orientados a objetos?

Parte-se da hipótese de que os padrões de projeto constituem instrumentos fundamentais para promover qualidade no desenvolvimento de *software*, uma vez que oferecem soluções reutilizáveis, favorecem a modularidade e contribuem diretamente para a construção de sistemas mais coesos, menos acoplados e, consequentemente, mais flexíveis e manuteníveis.

O presente estudo justifica-se pela crescente complexidade dos sistemas de *software* e pela necessidade de práticas de desenvolvimento que promovam qualidade, reutilização e facilidade de manutenção. Ao investigar a contribuição dos padrões de projeto para alcançar alta coesão e baixo acoplamento, pretende-se fornecer subsídios teóricos e práticos que auxiliem desenvolvedores e estudantes de computação na adoção de soluções arquiteturais mais eficientes. Além disso, a pesquisa contribui para o debate acadêmico sobre a importância dos princípios de design na construção de *software* robusto e sustentável.

Nesse sentido, o objetivo geral deste trabalho consiste em analisar as vantagens da utilização de padrões de projeto e como sua aplicação pode contribuir para alcançar alta coesão e baixo acoplamento em sistemas orientados a objetos. Para atingir esse propósito, busca-se, inicialmente, revisar a literatura sobre paradigmas de programação, orientação a objetos, acoplamento, coesão e padrões de projeto. Em seguida, pretende-se aplicar e comparar implementações de código que utilizem padrões de projeto e outras que não os empreguem, de modo a analisar os resultados obtidos em termos de coesão e acoplamento.

2. Revisão Bibliográfica

O campo da engenharia de software tem, ao longo das décadas, buscado aprimorar a qualidade dos projetos, com a adoção de paradigmas e práticas que promovam sistemas robustos e fáceis de manter. Nesse contexto, os padrões de projeto, inicialmente introduzidos por Alexander et al. [1977] no campo da arquitetura, foram posteriormente adaptados para o desenvolvimento de software por [Gamma et al., 1994], como soluções recorrentes para problemas de *design*. O uso desses padrões, conforme Wedyan e Abufakher [2020] apresenta, traz benefícios como aprimoramento na comunicação entre desenvolvedores, ganho de reusabilidade, sustentabilidade e, de forma mais ampla, eficiência no processo de desenvolvimento.

Paralelamente à catalogação dos padrões, a comunidade de engenharia de software buscou formalizar os princípios que definem um design orientado a objetos de alta qualidade. Desses esforços, destacam-se os princípios SOLID, compilados e popularizados por Martin [2002]. Esses cinco princípios — Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP) e Dependency Inversion Principle (DIP) — formam a base teórica para a criação de software manutenível e flexível, atuando como um guia central para se atingir o baixo acoplamento e a alta coesão.

Os padrões de projeto e os princípios SOLID estão intrinsecamente ligados. Os padrões, em geral, não são invenções arbitrárias, mas sim a materialização ou a implementação prática de um ou mais desses princípios. Por exemplo, o padrão Strategy é uma aplicação canônica do

Princípio Aberto/Fechado (OCP), pois permite a extensão do comportamento sem modificação do código existente. Da mesma forma, o Factory Method é um mecanismo fundamental para se alcançar a Inversão de Dependência (DIP). A literatura, portanto, não apenas investiga os padrões, mas também o grau em que sua aplicação reforça ou viola esses princípios fundamentais, o que pode explicar parte das divergências encontradas.

Um conceito fundamental para a melhoria da qualidade do código é a *refatoração*. Segundo Fowler [1999], *refatoração* é o processo de alteração de um sistema de *software* de modo que não se altere o comportamento externo do código, mas melhore-se a estrutura interna. É uma forma disciplinada de limpar o código que minimiza as chances de introdução de erros, sendo essencial para transformar uma arquitetura *monolítica* ou de baixa coesão em um sistema modular baseado em padrões.

Apesar da ampla aceitação dos padrões de projeto, a literatura sobre seu impacto em atributos de qualidade do software apresenta resultados contraditórios. Estudos conduzidos por Weiss [2008], Zhang e Budgen [2012], Ali e Elish [2013], Ampatzoglou et al. [2013a], Riaz et al. [2015] e Mayvan et al. [2017] indicam que o efeito de um padrão na qualidade não é trivial. Por exemplo, estudos que avaliaram o padrão *Abstract Factory*, que tem como objetivo criar famílias de objetos sem especificar suas classes concretas, apontam para resultados divergentes. Enquanto trabalhos como os de Prechelt et al. [2001]; Aversano et al. [2007b]; Gatrell et al. [2009] o associam a um aumento na extensibilidade, outros trabalhos, como os de Ali e Elish [2013]; Ampatzoglou et al. [2013b], identificaram uma redução. Essa divergência de resultados reflete a complexidade do tema. A inconsistência nos achados sugere a existência de fatores de confusão que obscurecem o real impacto dos padrões, o que justifica a necessidade de novas pesquisas. O presente estudo busca preencher essa lacuna por meio de uma abordagem experimental que controla variáveis e mede os efeitos de forma objetiva, fornecendo maior clareza sobre a contribuição dos padrões de projeto para a qualidade do software.

2.1. Fatores de Confusão e Atributos de Qualidade

A análise da literatura de Gatrell e Counsell [2011]; Gravino et al. [2011]; Scanniello et al. [2015] revela que a manutenibilidade é um dos atributos de qualidade mais avaliados em estudos sobre padrões de projeto. Artigos como de Bieman et al. [2001a] reforçam que os benefícios do seu uso se tornam mais evidentes durante a fase de manutenção e evolução do software. A manutenibilidade, de acordo com a norma ISO/IEC 9126 — posteriormente substituída pela ISO/IEC 25010 (2011) — é uma característica multidimensional composta por atributos como modificabilidade, estabilidade e propensão a mudanças (*change proneness*).

Um fator de confusão recorrente, que afeta essas métricas, é o tamanho da classe, conforme apontado por Pree [1994]; El Emam et al. [2001]. Estudos realizados por Bieman et al. [2001a, 2003] mostram que classes maiores tendem a apresentar maior propensão a mudanças. Essa constatação sugere que o tamanho pode atuar como medida indireta da propensão a mudanças, especialmente quando o esforço de manutenção não é considerado. Contudo, o efeito do uso de padrões no tamanho dos módulos ainda não está claro, como discutido em trabalhos de Khomh e Guéhéneuc [2008]; Gatrell e Counsell [2012], o que reforça a necessidade de mais investigação.

Outros fatores que influenciam a manutenibilidade incluem a documentação de padrões, conforme relatado por Wedyan et al. [2015], e a presença de preocupações transversais (*cross-cutting concerns*), destacada por Garcia et al. [2006]; Aversano et al. [2007b, 2009]. Estes sendo funcionalidades cujas implementação não podem ser encapsuladasem um único módulo e, portanto, espalham-se pelo código, prejudicandoa modularidade e aumentandoo acoplamento. Pes-

quisas de Hannemann e Kiczales [2002]; Garcia et al. [2006]; Aversano et al. [2009]; Cacho et al. [2014] indicam que a documentação de padrões melhora significativamente a compreensão do código e reduz o custo de manutenção em termos de tempo e falhas. No entanto, há uma lacuna no entendimento de como instâncias não intencionais de padrões de projeto impactam o sistema. Por outro lado, conforme evidenciado por Eaddy et al. [2007]; Aversano et al. [2007b]; Eaddy et al. [2008]; Aversano et al. [2009], a implementação de preocupações transversais tende a diminuir a modularidade e a afetar negativamente a qualidade do software.

A modularização de padrões também se apresenta como uma linha de investigação promissora, especialmente por meio da Programação Orientada a Aspectos (*AOP*), segundo Cacho et al. [2014]. Embora essa abordagem apresente resultados iniciais positivos, ainda existem barreiras técnicas para sua adoção em larga escala, como apontam Roo et al. [2008]; Tanter et al. [2014]; Leger e Fukuda [2014]. Além disso, o uso de modelos de qualidade, na avaliação de padrões de projeto tem produzido resultados mistos, conforme relatado em Bieman et al. [2001b]; Aversano et al. [2007a]; Izurieta e Bieman [2013]; Sfetsos et al. [2014]; Hussain et al. [2017]. Em alguns cenários, os padrões contribuem positivamente; em outros, o efeito é neutro ou inexistente. Esses achados reforçam a ideia de que o impacto varia conforme o contexto de aplicação e evidenciam lacunas relevantes que ainda precisam ser exploradas na literatura.

3. Metodologia

A coleta principal dos dados numéricos foi realizada com a ferramenta *CKJM* (*Chidamber & Kemerer Java Metrics*), cuja validade para análise da propensão a falhas e complexidade é um pilar da investigação em engenharia de *software* sobre o trabalho de Subramanyam e Krishnan [2003].

Para responder à questão de pesquisa e validar a hipótese formulada, esta pesquisa adota uma abordagem quantitativa de natureza experimental, configurada como um estudo de caso. O objetivo é aferir objetivamente o impacto da aplicação de padrões de projeto nos atributos de qualidade de coesão e acoplamento. A metodologia foi estruturada em quatro etapas: a definição do objeto de estudo, a criação de duas versões do sistema para comparação, a seleção e coleta de métricas quantitativas e, por fim, a análise dos dados. Os conceitos e as métricas essenciais utilizados neste estudo, juntamente com seu contexto no projeto, estão detalhados na Tabela 2.

Tabela 2. Conceitos e Métricas Essenciais do Estudo

Conceito	Contexto e Definição no Projeto
CKJM	Ferramenta principal (*Chidamber & Kemerer Java Metrics*) utilizada para a coleta primária de métricas quantitativas de Orientação a Objetos.
WMC	Sigla para *Weighted Methods per Class*. É uma métrica de Complexidade usada para contextualizar a complexidade total das classes.
God Classes	Classes que concentram complexidade crítica e muitas responsabilidades, resultando em métricas de coesão falsamente positivas no *LCOM* e violando o *SRP*.
RFC	Sigla para *Response For Class*. É uma métrica de Acoplamento que mede o impacto de uma mudança, contando o total de métodos que podem ser chamados a partir de uma classe.
Grupo Controle	As classes *OLD* referem-se às 10 classes originais, o objeto de refatoração.

Fonte: [a]Adaptado de Chidamber & Kemerer (1994); [b]Adaptado de Fowler (1999). O conceito de "Grupo Controle"é uma definição metodológica dos autores.

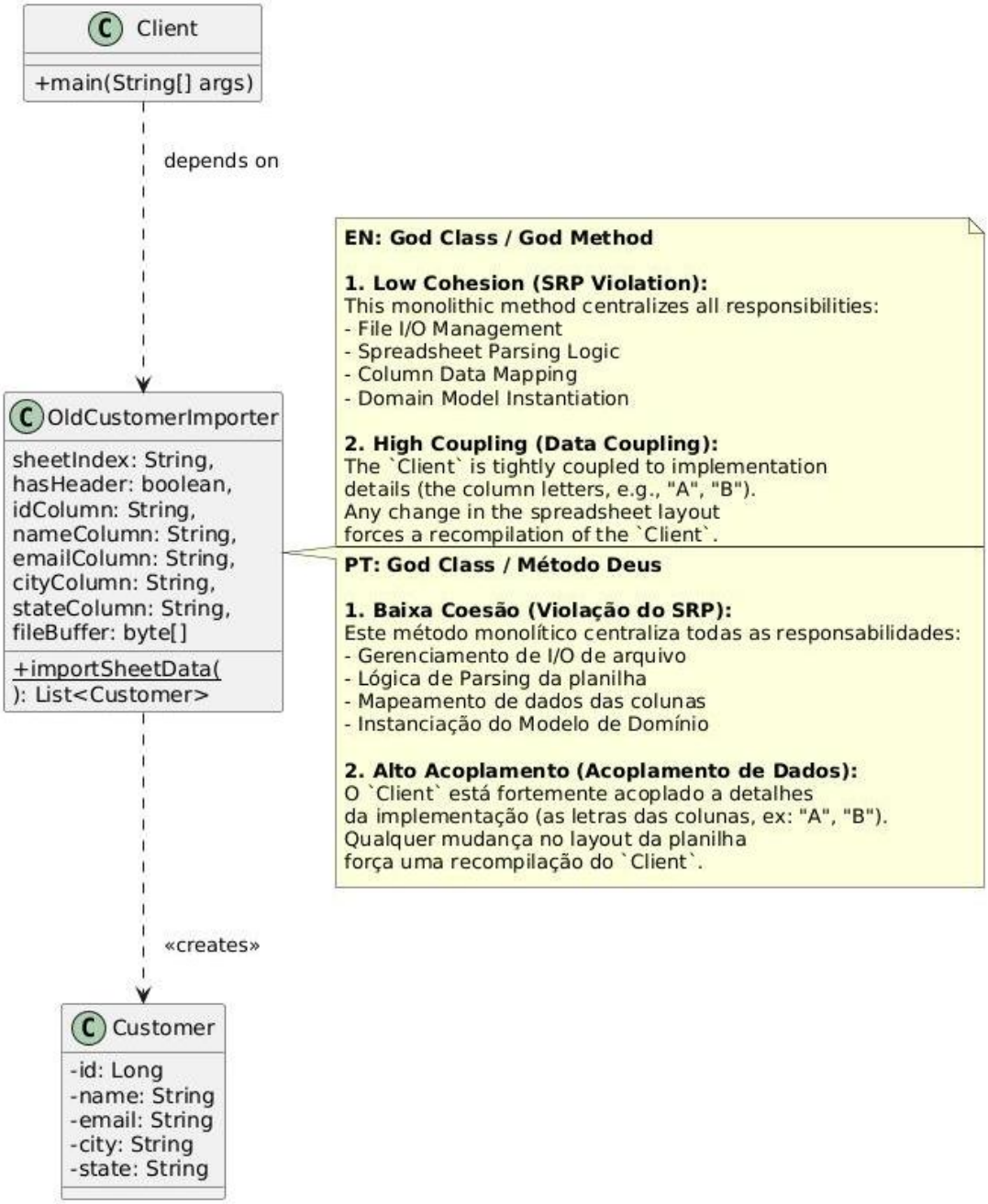
3.1. Delineamento da Pesquisa e Objeto de Estudo

O objeto de estudo é o *SheetReader*, um motor de importação de planilhas desenvolvido em *Java* com *Maven*. Para viabilizar a análise comparativa, o projeto foi estruturado em duas versões distintas, representando os grupos de controle e experimental:

1. Implementação *Baseline* (Grupo de Controle): A versão inicial do sistema, desenvolvida com uma abordagem direta, sem a aplicação explícita dos padrões de projeto. Esta versão é caracterizada pelo uso de uma *"God Class"* que centraliza a lógica de importação.
2. Implementação Refatorada (Grupo Experimental): A versão reestruturada a partir da *baseline*, com a aplicação deliberada dos padrões *Factory Method*, *Template Method* e *Strategy* para solucionar pontos específicos de *design*.

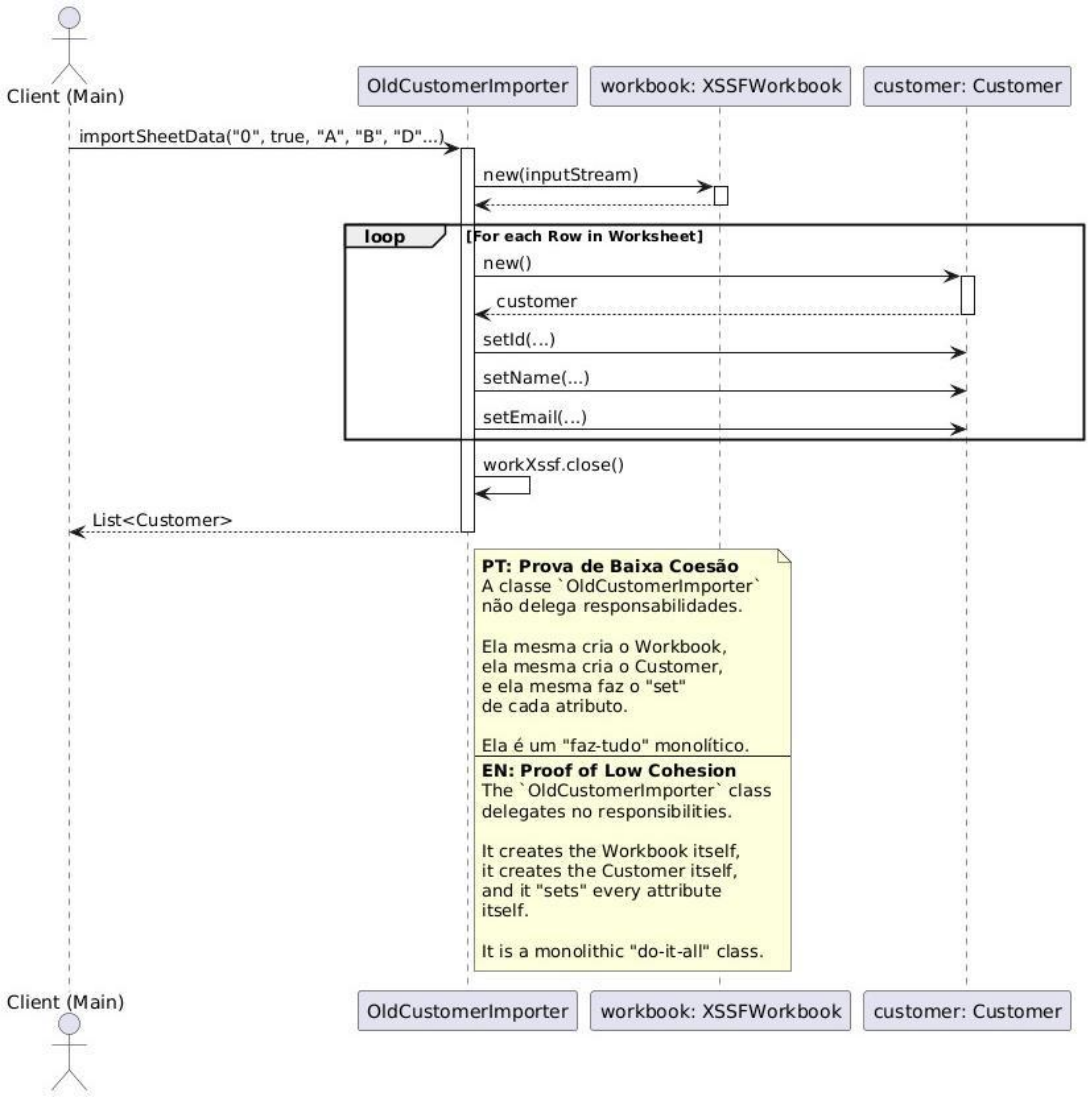
A *Implementação Baseline* é detalhada nos diagramas das Figura 1 que ilustra a arquitetura com *God Class* e Alto Acoplamento, enquanto a Figura 2 expõe a Baixa Coesão na sequência de execução monolítica.

Figura 1. Arquitetura Original (*God Class* / Alto Acoplamento).



Fonte: Dados coletados pelos autores (2025). UML Gerado na ferramenta plantUML.

Figura 2. Diagrama de Sequência da *God Class* (Baixa Coesão).



Fonte: Dados coletados pelos autores (2025). UML Gerado na ferramenta plantUML.

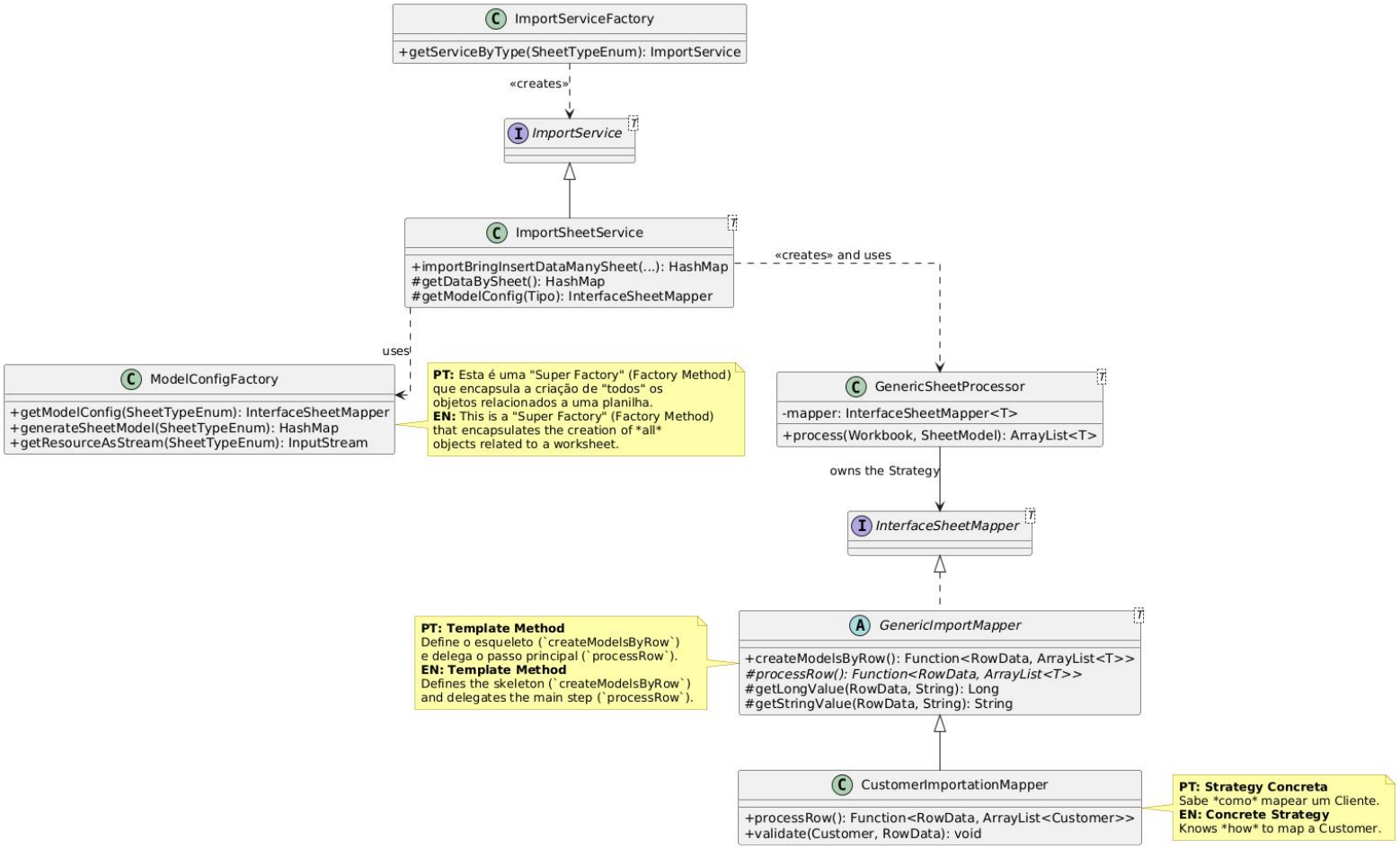
A variável independente deste estudo é a aplicação dos padrões de projeto. As variáveis dependentes são as métricas de coesão e acoplamento.

Para analisar o impacto dos padrões na complexidade e escalabilidade, a comparação foi realizada em dois cenários de carga de trabalho para ambas as versões:

- Cenário 1 (Baixa Complexidade): Importação de 2 classes de domínio distintas.
- Cenário 2 (Complexidade Elevada): Importação de 10 classes de domínio distintas.

Isso permite analisar não apenas a diferença estática entre as versões, mas como a arquitetura com a refatoração exposta na Figura 3 responde ao aumento de responsabilidades, testando diretamente o *Princípio Aberto/Fechado* (*OCP*).

Figura 3. Diagrama de Classes: Arquitetura com Padrões (*Factory Method*, *Strategy*, *Template Method*).



Fonte: Dados coletados pelos autores (2025). UML Gerado na ferramenta plantUML.

3.2. Procedimentos e Ferramentas de Coleta de Dados

Para garantir a robustez da análise, foi adotada uma abordagem com múltiplas ferramentas e técnicas: a ferramenta de linha de comando *CKJM* para a extração primária de dados quantitativos, a plataforma *SonarQube* para análise visual de *code smells* (indicadores de baixa qualidade ou má concepção do código) e, por fim, a análise de diagramas UML para avaliação qualitativa da estrutura.

3.2.1. Extração de Dados Quantitativos com CKJM

A coleta principal dos dados numéricos foi realizada com a ferramenta *CKJM*(*Chidamber & Kemerer Java Metrics*), um utilitário de linha de comando focado em métricas de *software* orientado a objetos. O processo seguiu os seguintes passos para cada cenário do projeto (*Baseline*-2, Refatorada-2, *Baseline*-10, Refatorada-10):

1. Compilação do Projeto: O código-fonte de cada versão foi compilado utilizando o *Apache Maven* para gerar os arquivos `.class` necessários para a análise estática.
2. Execução da Ferramenta: O *CKJM* foi executado via linha de comando, tendo como entrada o diretório das classes compiladas (`target/classes`).

3. Geração dos Dados: A ferramenta produziu um arquivo de saída no formato `.txt` para cada versão, contendo os valores das métricas calculadas para cada classe do projeto. Este arquivo serviu como fonte primária para a análise estatística comparativa.

3.2.2. Análise Visual e Qualitativa com SonarQube

Para enriquecer a análise e fornecer um contexto visual e qualitativo, foi utilizada a plataforma de inspeção contínua de qualidade de código *SonarQube* (*Community Edition*).

1. Configuração do Ambiente: Uma instância local do *SonarQube* foi instalada e executada na máquina de desenvolvimento, garantindo um ambiente de análise limpo e isolado.
2. Análise das Versões: Cada um dos cenários foi analisado separadamente através do *plugin SonarScanner* para *Maven*. Foram criados projetos distintos no painel do *SonarQube* (ex: `SheetReader-Baseline-10`, `SheetReader-Refactored-10`) para permitir a comparação direta.
3. Inspeção dos Resultados: O *dashboard web* do *SonarQube* foi utilizado para inspecionar visualmente as métricas, identificar *code smells* e validar os dados coletados pelo CKJM.

3.3. Métricas de Análise

A análise foi conduzida por uma abordagem mista, combinando métricas quantitativas de ferramenta com análise qualitativa de diagramas.

Métricas Quantitativas (Coletadas pelo CKJM e SonarQube)

- Acoplamento(CKJM) - *CBO (Coupling Between Objects)*: Conta o número de outras classes às quais uma classe está acoplada. Um valor alto de *CBO* indica alta dependência, dificultando a manutenção e o reuso. O objetivo é obter um valor baixo.
- Métrica de Controle(CKJM) - *LOC (Lines of Code)*: O número de linhas de código foi coletado para contextualizar os resultados, verificando se a aplicação dos padrões impactou o tamanho do sistema, um fator de confusão conhecido em estudos de manutenibilidade.
- Complexidade Ciclomática(SonarQube): Métrica proposta por (McCabe [1976]) que mede o número de caminhos linearmente independentes no código. Representa a complexidade estrutural e o esforço necessário para testes (*coverage*). Um alto valor indica que o fluxo de controle é ramificado, exigindo mais casos de teste unitário.
- Complexidade Cognitiva(SonarQube): Métrica introduzida pela SonarSource (Campbell [2017]) para medir o esforço mental necessário para entender o fluxo de controle. Diferentemente da ciclomática, ela penaliza estruturas que dificultam a leitura humana (como aninhamentos profundos e quebras de fluxo) e ignora estruturas verbosas, mas simples (como *getters/setters*). É um indicador direto de *manutenibilidade*.

Métricas Qualitativas (Análise Manual e UML)

- Coesão - *LCOM (Lack of Cohesion in Methods)*: Mede a falta de coesão em uma classe. Embora o CKJM calcule o *LCOM*, foi identificado durante a coleta que a métrica apresentou distorções significativas na versão *Baseline*. O cálculo do *LCOM* pelo CKJM, que se baseia em atributos compartilhados por métodos, não captura adequadamente a falta de coesão em uma "God Class"que centraliza lógicas proceduralmente.

- Análise Estrutural via UML: Devido à limitação identificada no *LCOM*, a análise de coesão e acoplamento foi suplementada por uma avaliação qualitativa. Diagramas de classes UML de ambas as versões (Baseline e Refatorada) foram gerados e analisados visualmente. Esta análise focou em:
  - *Coesão (Visual):* Verificando a aderência ao Princípio da Responsabilidade Única (SRP), comparando a "God Class"da *Baseline* com a distribuição de responsabilidades nas classes de Strategy e Template na *Refatorada*.
  - *Acoplamento (Visual):* Inspecionando visualmente as linhas de dependência, validando o Princípio da Inversão de Dependência (DIP) na versão refatorada e o alto acoplamento direto na *Baseline*.

3.4. Análise dos Resultados

A etapa final da metodologia consiste na análise comparativa dos dados coletados. Os dados brutos da ferramenta *CKJM* serão utilizados para criar tabelas comparativas e realizar análises estatísticas (como cálculo de soma, média e desvio padrão) dos valores de *CBO* e *LOC*.

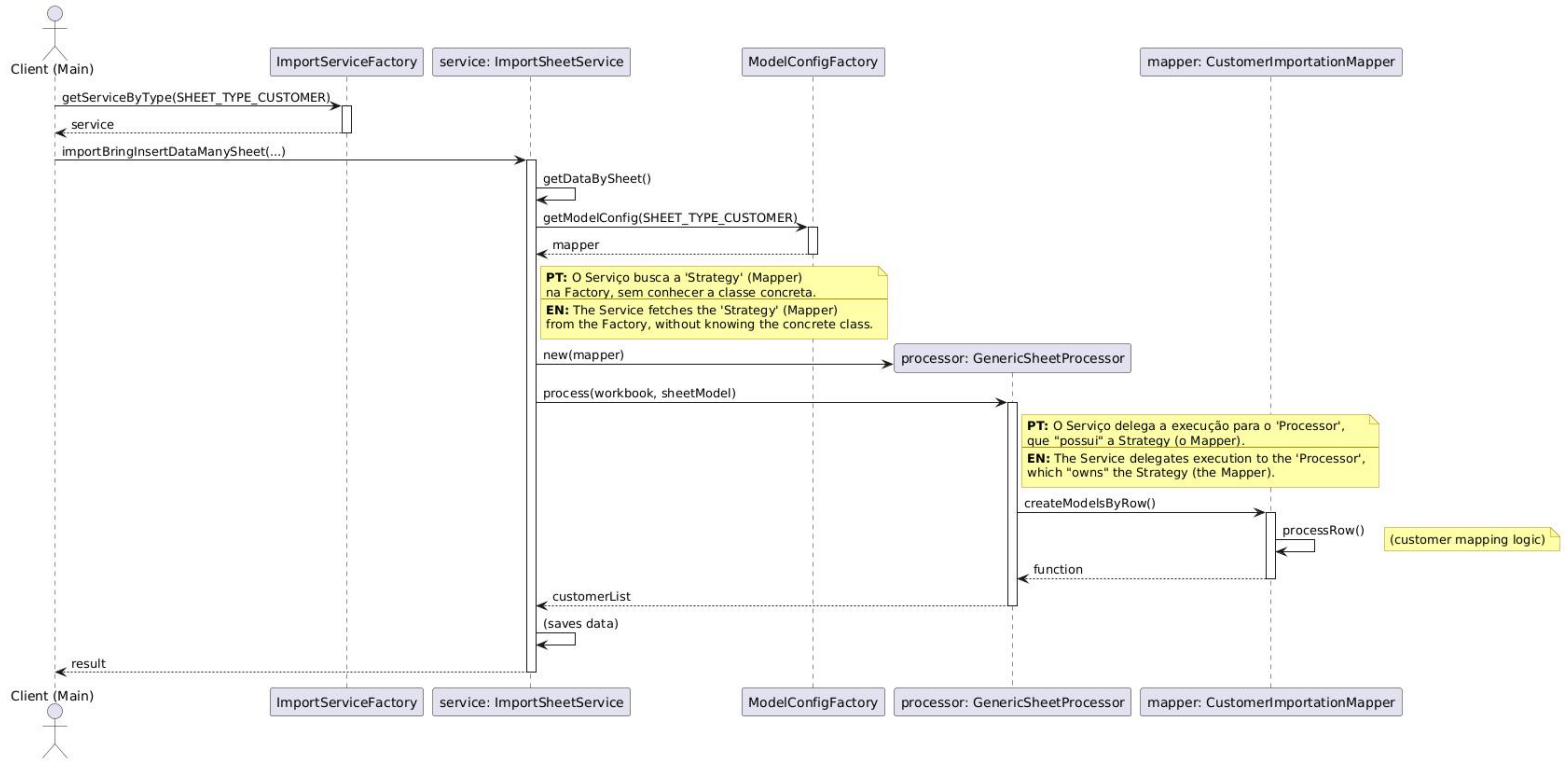
Esta análise será realizada comparando os quatro cenários definidos: *Baseline-2* vs. *Refatorada-2* e *Baseline-10* vs. *Refatorada-10*.

Esses dados quantitativos serão, então, triangulados com as evidências qualitativas descritas na seção anterior:

- Os relatórios do *SonarQube* serão usados para discutir a redução de *code smells* e fornecer um contexto visual para os problemas de design.
- A análise dos diagramas UML será usada para discutir a melhoria na coesão estrutural (SRP) e no acoplamento (DIP), suprindo a lacuna da métrica *LCOM* do CKJM, que se mostrou insuficiente para o caso da "God Class". O baixo acoplamento em tempo de execução e a aplicação dos padrões *Factory*, *Strategy* e *Template Method* são visualmente comprovados na Figura 4.

Essa abordagem mista, combinando dados quantitativos brutos com plataformas de análise visual e diagramas UML, visa fornecer uma conclusão robusta e bem fundamentada sobre a contribuição dos padrões de projeto para a alta coesão e o baixo acoplamento, especialmente em um contexto de crescimento da complexidade do software.

Figura 4. Diagrama de Sequência de Execução (Baixo Acoplamento).



Fonte: Dados coletados pelos autores (2025). UML gerado na ferramenta plantUML.

4. Resultados e Discussão

Para alcançar a arquitetura proposta, foram aplicadas técnicas de *refatoração* catalogadas por Fowler (Fowler [1999]), destacando-se: (i) *Extract Class*, para mover responsabilidades da *God Class* para novas classes coesas; (ii) *Extract Method*, para quebrar o método monolítico em etapas menores do *Template Method*; e (iii) *Replace Conditional with Polymorphism*, eliminando as checagens de tipo (*if/else*) e substituindo-as pela injeção dinâmica das estratégias (*Strategy*).

Para avaliar o impacto da aplicação de padrões de projeto na qualidade do *software*, duas versões do sistema *SheetReader* foram analisadas: uma "Versão Original" (com lógica procedural) e uma "Versão Refatorada" (utilizando padrões como *Template Method*, *Factory* e *Strategy*). A avaliação foi realizada por meio de duas ferramentas de análise estática: *CKJM* (para métricas de nível de classe) e *SonarQube* (para uma avaliação holística da qualidade). As métricas de qualidade de *software* selecionadas para esta avaliação estão detalhadas na Tabela 3.

Tabela 3. Significado e Objetivo das Métricas de Qualidade de Software (CKJM)

Métrica (Sigla)	O que Significa?	Qual é o Objetivo?
CBO (*Acoplamento*)	"Acoplamento (*Coupling Between Objects*) Conta quantas classes esta classe depende diretamente (seja usando, herdando, etc.). Mede Acoplamento."	Baixo (Reduzir a dependência externa)
LCOM (*Coesão*)	"Falta de Coesão (*Lack of Cohesion in Methods*) Mede o quão 'focada' a classe é. Um valor alto significa que os métodos da classe não têm muito em comum (provavelmente faz coisas demais, violando o SRP). Mede Coesão (interna)."	Baixo (Manter a classe focada)
RFC (*Responsabilidade*)	"Acoplamento (*Response For Class*) Mede o 'impacto' de uma mudança. Conta o número total de métodos que podem ser chamados a partir de uma classe (seus próprios métodos + métodos de outras classes que ela usa). Mede Responsabilidade."	Baixo (Limitar o impacto de mudanças)
WMC (*Complexidade*)	"Complexidade (*Weighted Methods per Class*) Mede a complexidade total da classe. É a soma da complexidade (quantos *if*, *for*, *while*) de todos os seus métodos. Mede Complexidade interna."	Baixo (Tornar o código mais simples)

Fonte: [a]Adaptado de Chidamber & Kemerer (1994).

4.1. Análise de Métricas de Nível de Classe (*CKJM*)

A análise quantitativa, resumida na Tabela 4, avaliou o impacto dos padrões em múltiplas dimensões: camadas da arquitetura, escalabilidade e qualidade geral do código. Esta análise comparou as duas versões do projeto em um cenário escalado para 10 importadores distintos. Os resultados demonstram melhorias inequívocas em todos os aspectos avaliados.

Nas camadas de *Orquestradores* e *Implementadores*, a refatoração resultou em uma redução drástica de Acoplamento (CBO -35% e -33%, respectivamente) e de Complexidade (RFC -31%). A eficiência de código também foi notável, com reduções de 59% e 51% em Linhas de Código (*LOC*) nessas mesmas camadas, indicando implementações mais enxutas e focadas.

O maior impacto, contudo, é observado na *Escalabilidade* e *Qualidade*. A arquitetura refatorada eliminou 100% da duplicação de código, reduziu o custo de manutenção por nova entidade em 53% e, crucialmente, alterou a curva de crescimento de complexidade de Linear (O(n)) para Sub-linear (O(log n)). Isso se traduz diretamente em um sistema mais fácil de manter, testar (cuja *Testabilidade* evoluiu de "Quase impossível"para "Fácil") e estender (com *Extensibilidade* baseada em "Configurar"ao invés de "Duplicar código").

Para fins de análise na Tabela 4, definem-se os conceitos observados: "Orquestradores" referem-se às classes responsáveis pelo fluxo de controle e delegação (como os *Services* e *Processors*); já os "Implementadores" são as classes concretas que contêm a lógica de negócio específica de cada formato de planilha (as classes *Strategy*).

Tabela 4. Tabela Comparativa Resumida (Cenário com 10 importadores)

Métrica (Foco)	Versão Original (Média)	Versão Refatorada (Média)	Mudança	Impacto na Qualidade
ORQUESTRADORES				
Acoplamento (CBO)	20	13	-35%	*Melhor*
Código (LOC)	598	244	-59%	*Melhor*
IMPLEMENTADORES				
Acoplamento (CBO)	6	4	-33%	*Melhor*
Complexidade (RFC)	29	20	-31%	*Melhor*
Código (LOC)	163	80	-51%	*Melhor*
ARQUITETURA COMPLETA				
Código Total (LOC)	3,369	3,196	-5%	*Melhor*
Acoplamento Total (CBO)	127	121	-5%	*Melhor*
ESCALABILIDADE				
Custo por Nova Entidade	+163 LOC	+77 LOC	-53%	*Melhor*
Complexidade de Manutenção	17 pontos	3 pontos	-82%	*Melhor*
Curva de Crescimento	Linear O(n)	Sub-linear O(log n)	*Ordem melhor*	*Melhor*
QUALIDADE				
Duplicação de Código	85%	0%	-100%	*Melhor*
Testabilidade	Quase impossível	Fácil	*Melhorada*	*Melhor*
Extensibilidade	Duplicar código	Configurar	*Flexível*	*Melhor*

Fonte: Dados coletados pelos autores através da ferramenta CKJM

4.2. Análise de Qualidade Holística (*SonarQube*)

Esta interpretação é validada pela análise da plataforma *SonarQube*, que avalia a manutenibilidade do código de forma holística. Os resultados comparativos entre as versões estão detalhados na Tabela 5.

Tabela 5. Métricas de Qualidade do SonarQube por Versão (Cenário com 2 Importadores)

Métrica	Versão Original (299 LOC)	Versão Refatorada (1.4k LOC)
Security (Risco)	A (0 Vulnerabilidades)	A (0 Vulnerabilidades)
Maintainability (Manutenibilidade)	E (Crítica)	A (Excelente)
Dívida Técnica (Esforço)	23 Minutos	74 Minutos
Complexidade Ciclomática	75	306
Complexidade Cognitiva	34	186

Fonte: Dados coletados pelos autores através da ferramenta SonarQube.

Esta interpretação é validada pela análise da plataforma SonarQube, realizada em um cenário inicial com 2 importadores (Tabela 5), que avalia a manutenibilidade do código de

forma holística, atribuindo notas de 'A' (Excelente) até 'E' (Crítica). Os resultados expõem a armadilha de usar Linhas de Código (LOC) como uma métrica de qualidade isolada. A "Versão Original"(299 LOC), embora significativamente menor, recebeu a pior nota de manutenibilidade possível ("E- Crítica). Isso prova que o código não era "enxuto", mas sim perigosamente denso, concentrando "Code Smells"e complexidade crítica (como o WMC de 72.5) em "God Classes"estruturalmente pobres.

Em contraste, a "Versão Refatorada"(1.4k LOC) recebeu a nota máxima ("A- Excelente). O aumento expressivo no LOC não representa uma piora; pelo contrário, é a evidência física da distribuição da complexidade, conforme o SRP. Este "custo"inicial de LOC refere-se à criação do framework de importação (Factories, Services, Mappers). Como demonstrado na simulação de escalabilidade (6), este framework permite um crescimento linear e controlado. A "Versão Original", por outro lado, cresce exponencialmente (via "Copiar e Colar"), o que prova que ela rapidamente superaria o LOC da versão refatorada após a adição de poucos novos importadores, tornando-se insustentável em todos os aspectos.

Contudo, é importante notar que para sistemas muito simples e sem perspectiva de escalabilidade, a "Versão Original", apesar da nota crítica, pode ser considerada aceitável. Nesses casos, o custo inicial de 1.4k LOC da arquitetura refatorada poderia não justificar o esforço de implementação, caracterizando um caso de *overengineering* (ou seja, uma solução mais complexa do que o problema exige) para um escopo tão limitado.

4.3. Análise de Escalabilidade (Simulação de 2 para 10 Importadores)

A análise final avalia o benefício mais crítico da nova arquitetura: a escalabilidade. O custo de adicionar 8 novos tipos de importação a ambas as versões foi simulado, e os resultados estão detalhados na Tabela 6.

Na "Versão Original", a única forma de adicionar funcionalidade é "Copiar e Colar" as "*God Classes*" (*OldCustomerImporter*), duplicando o código complexo e a dívida técnica a cada novo importador.

Na "Versão Refatorada", o *framework* de processamento (*GenericSheetProcessor*) é amplamente reutilizado. O custo de extensão para uma nova entidade é mínimo: não é necessário criar novas classes de serviço, pois a classe genérica *ImportSheetService<T>* é reutilizada, sendo apenas instanciada com o tipo de modelo de domínio desejado (ex: *Product*, *Supplier*). O custo real se limita a duas modificações centrais e de baixo esforço: a adição de um novo valor ao *SheetTypeEnum* e a adição de um novo *case* na *ImportServiceFactory*. Esta abordagem centraliza a lógica de criação em um único local, em vez de duplicar centenas de linhas de código, como na "Versão Original".

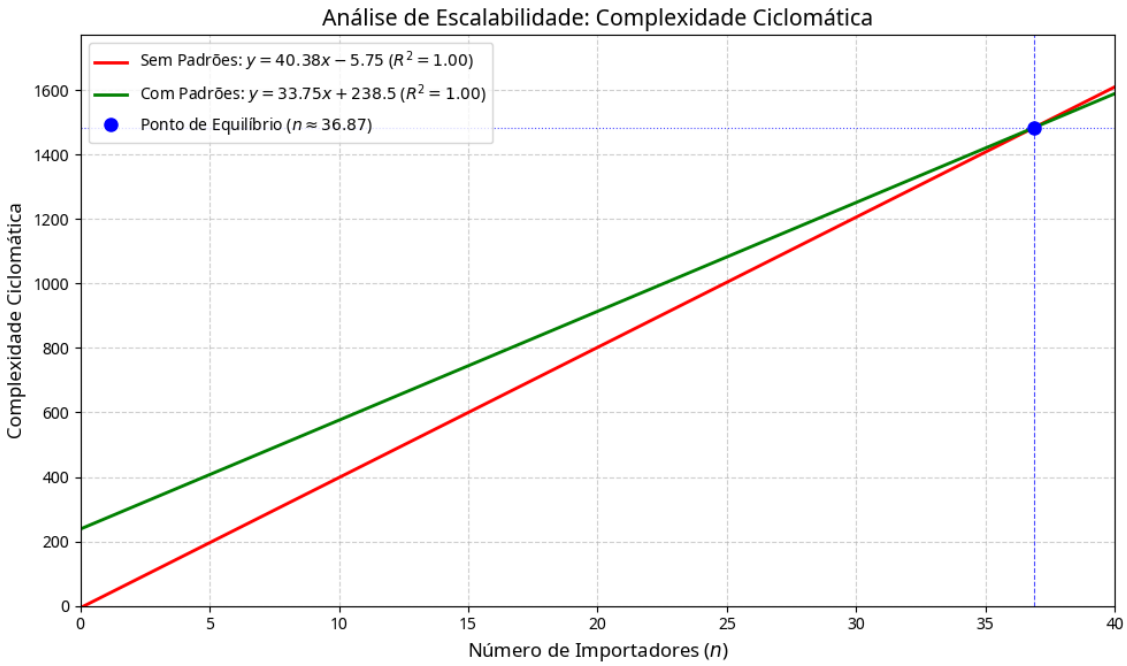
Tabela 6. Simulação de Custo de Escalabilidade (de 2 para 10 Importadores)

Métrica (Total)	V. Orig. (2 Imp)	V. Orig. (10 Imp)	V. Refat. (2 Imp)	V. Refat. (10 Imp)
Complexidade Total (WMC)	∼153	∼765	∼285	∼365
Manutenibilidade (Sonar)	E (Crítica)	E (Crítica)	A (Excelente)	A (Excelente)
Dívida Técnica Total	23 min	∼115 min	74 min	∼98 min
Complexidade Ciclomática	75	∼398	306	∼576
Complexidade Cognitiva	34	∼414	186	∼239

Fonte: Dados coletados pelos autores através das ferramentas CKJM e SonarQube.

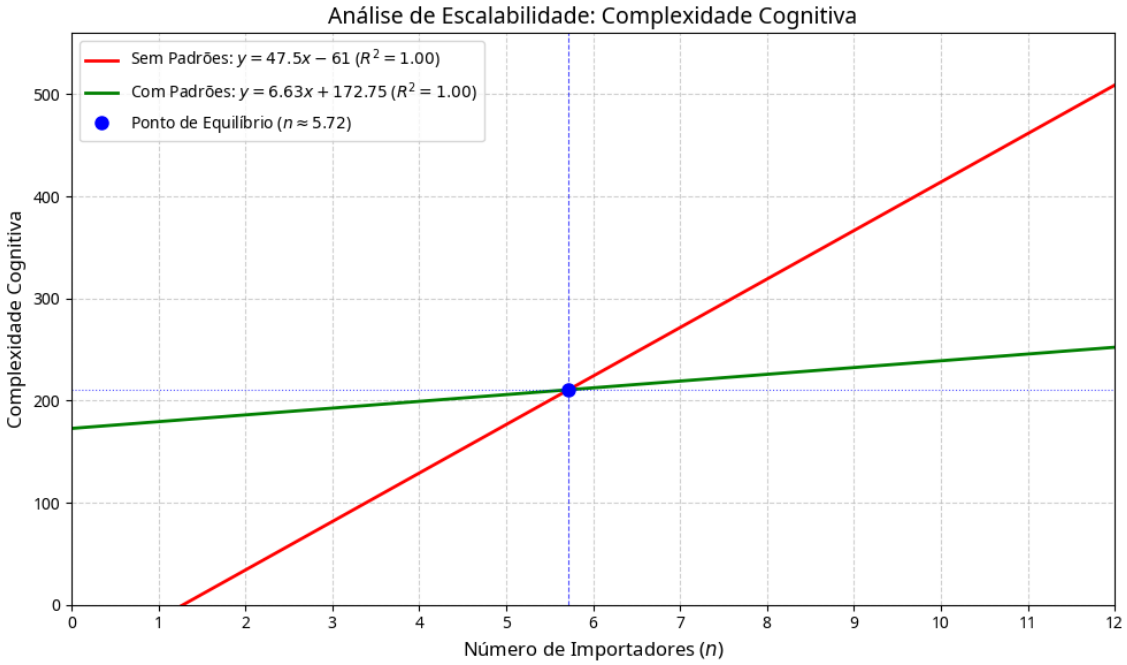
A simulação expõe a falha crítica do *design* original: ele não escala. O custo de manutenção cresce exponencialmente, como visto no salto do *WMC* total de ∼153 para ∼765, tornando o projeto rapidamente inviável. A "Versão Refatorada" exige um "investimento" inicial maior em arquitetura (*Dívida Técnica* inicial de 74 min), mas escala de forma linear e sustentável (*WMC* de ∼285 para ∼365), mantendo a nota 'A'. O resultado mais impactante da simulação é o ponto de inflexão (*break-even point*) da *Dívida Técnica*. Como mostra a Tabela 6, com 10 importadores, o projeto refatorado (∼98 min) passa a ter menos dívida técnica total que o projeto original (∼115 min). Isso prova quantitativamente que o investimento inicial na arquitetura se paga rapidamente à medida que o projeto cresce. Além do ganho quantitativo, a análise expõe o benefício qualitativo na manutenção, que é uma consequência direta da aplicação do *Princípio Aberto-Fechado* (*OCP*) e do padrão *Template Method*: Na Versão Original, uma correção de *bug* na lógica de leitura do *Excel* exigiria a modificação manual de todas as 10 "*God Classes*", multiplicando o esforço e o risco de introdução de novos defeitos. Na Versão Refatorada, essa mesma correção é aplicada uma única vez na classe base (o *template*). A correção é automaticamente propagada para todos os 10 modelos de importação, garantindo consistência e reduzindo o custo de manutenção de forma drástica.

Figura 5. Complexidade Ciclomática (Estrutural).



Fonte: Dados coletados pelos autores (2025).

Figura 6. Complexidade Cognitiva (Compreensão Humana).



Fonte: Dados coletados pelos autores (2025).

A análise detalhada das complexidades Ciclomática e Cognitiva no cenário de escalabilidade (10 importadores) revela um fenômeno importante sobre a aplicação de padrões de projeto, conforme ilustrado na Figura 5 e na Figura 6.

Observa-se um aparente paradoxo: a Complexidade Ciclomática da versão refatorada (576) apresenta-se superior à da versão original (398), como visto na Figura 5. Isso ocorre porque a aplicação de padrões como *Factory* e *Strategy* aumenta a quantidade de classes, interfaces e métodos delegados, multiplicando os caminhos estruturais que precisam ser testados pela máquina. O gráfico indica um ponto de equilíbrio tardio ($n \approx 37$), sugerindo que, estritamente sob a ótica de caminhos de teste, a arquitetura modular é mais densa.

Em contrapartida, a Complexidade Cognitiva — que mede o esforço humano para compreensão — conta uma história diferente na Figura 6. A versão refatorada mantém-se estável e baixa (239), enquanto a versão original dispara (414). O ponto de equilíbrio ocorre rapidamente, em $n \approx 6$ importadores.

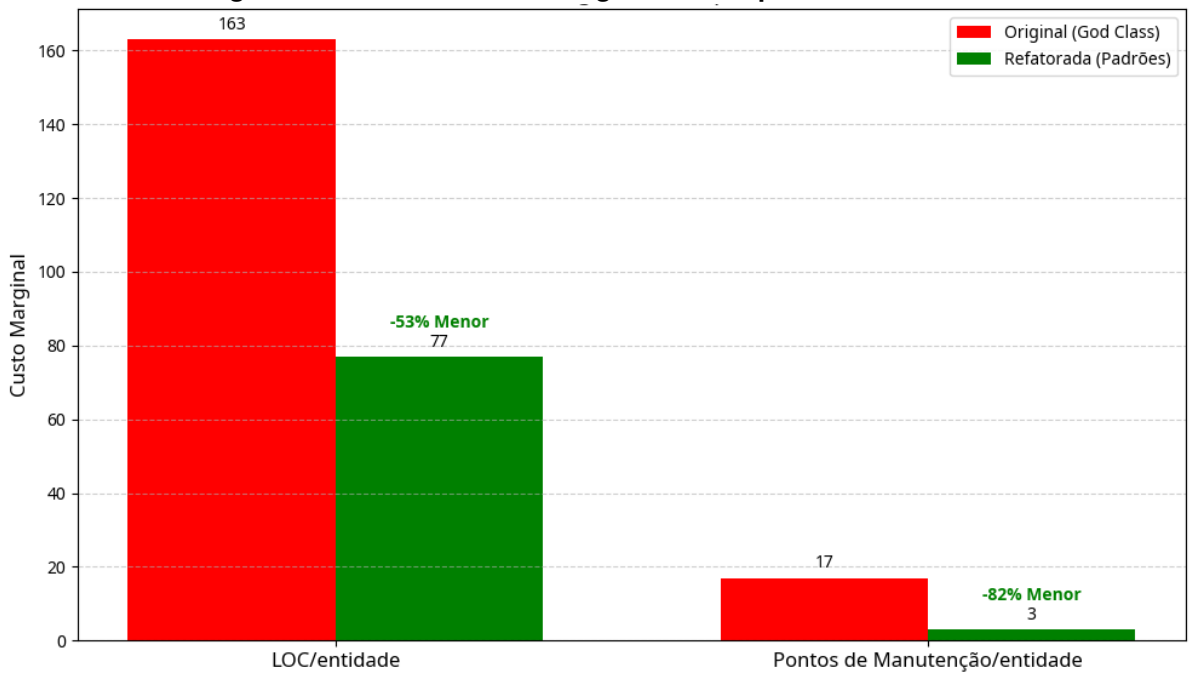
Esse resultado, cruzado com os dados da Tabela 6, valida a distinção qualitativa: enquanto a versão original concentra complexidade em God Classes com aninhamentos profundos (difíceis de ler, alto custo cognitivo), a versão refatorada distribui a complexidade entre muitos componentes pequenos e coesos. Embora haja mais peças para a máquina gerenciar (Ciclomática), cada peça é trivialmente simples para o desenvolvedor entender (Cognitiva).

4.4. Análise Gráfica de Custo-Benefício e Escalabilidade

Para visualizar os dados das Tabelas 4 e 6 e ilustrar o *trade-off* de escalabilidade, foram geradas funções lineares para representar o crescimento do custo de cada arquitetura, onde n é o número de importadores.

A Figura 7 (baseada na Tabela 4) compara o custo marginal de adicionar uma nova entidade. Fica evidente que a arquitetura com padrões é 53% mais eficiente em LOC e 82% mais eficiente em pontos de manutenção a cada nova funcionalidade.

Figura 7. Análise de Custo Marginal: Custo por Nova Entidade

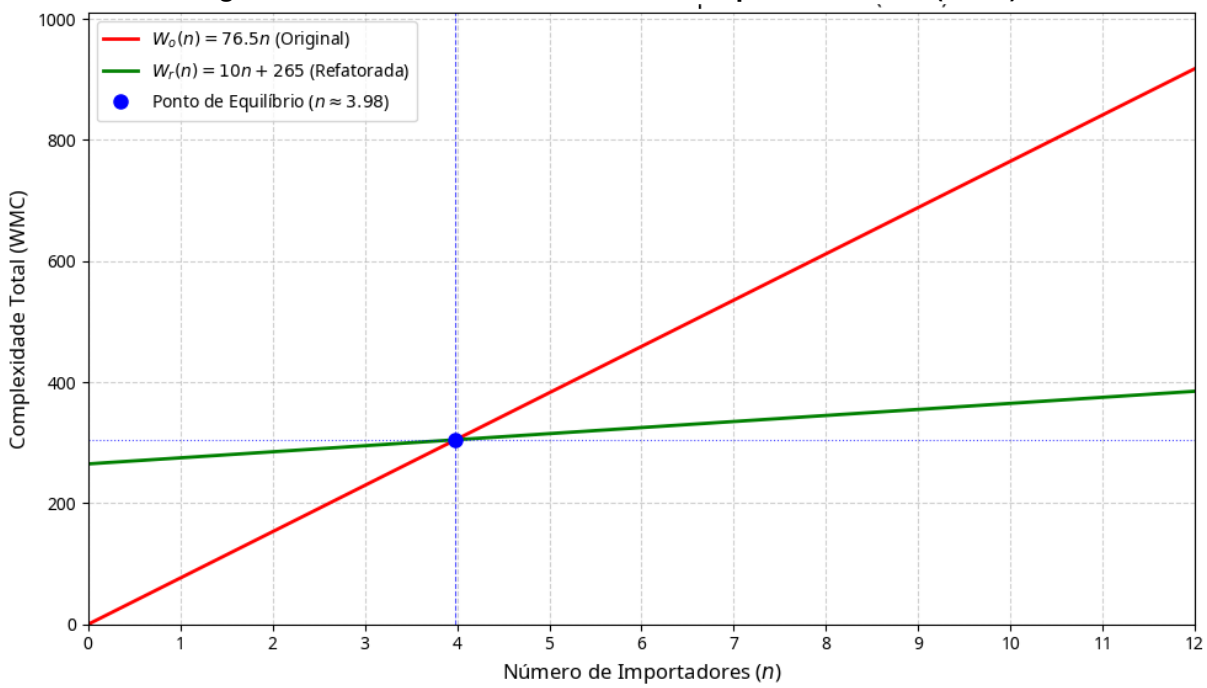


Fonte: Dados coletados pelos autores (2025)

A Figura 8 e Figura 9 (baseadas na Tabela 6) modelam o custo total. A "Versão Original" (vermelho) começa com custo quase zero, mas cresce a uma taxa linear acentuada. A "Versão

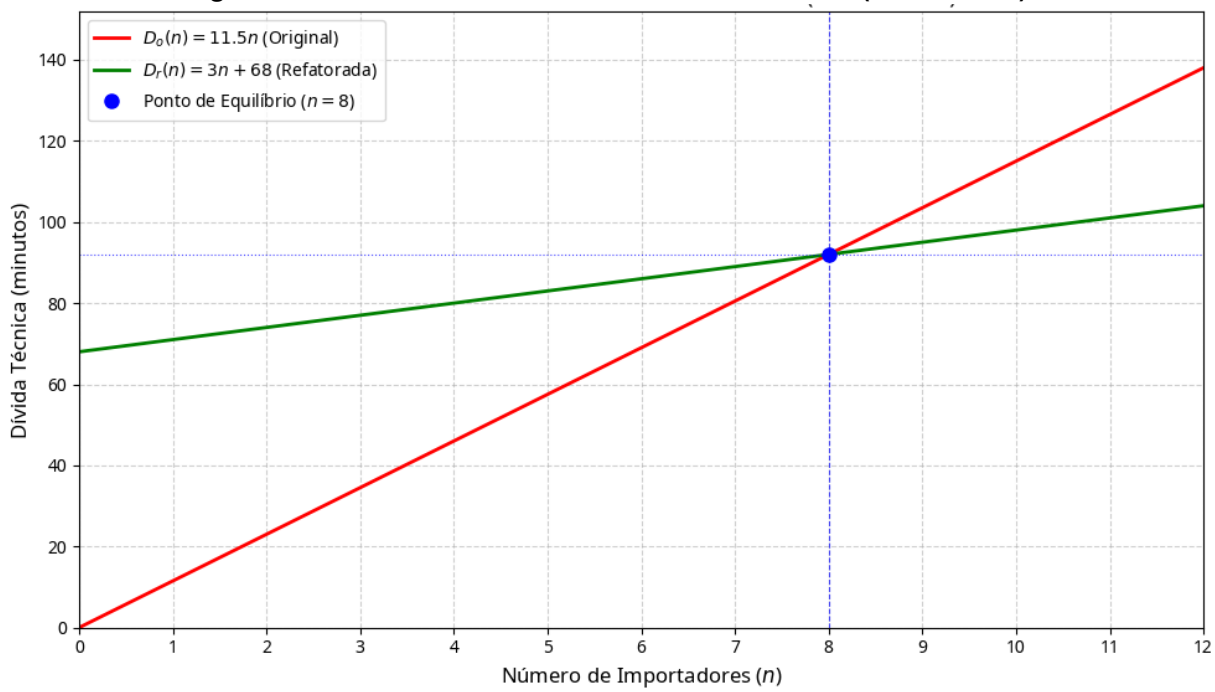
Refatorada" (verde) tem um custo inicial maior (o "investimento" no *framework*), mas uma taxa de crescimento muito menor.

Figura 8. Análise de Custo-Benefício: Complexidade Total (WMC)



Fonte: Dados coletados pelos autores (2025)

Figura 9. Análise de Custo-Benefício: Dívida Técnica (em minutos)



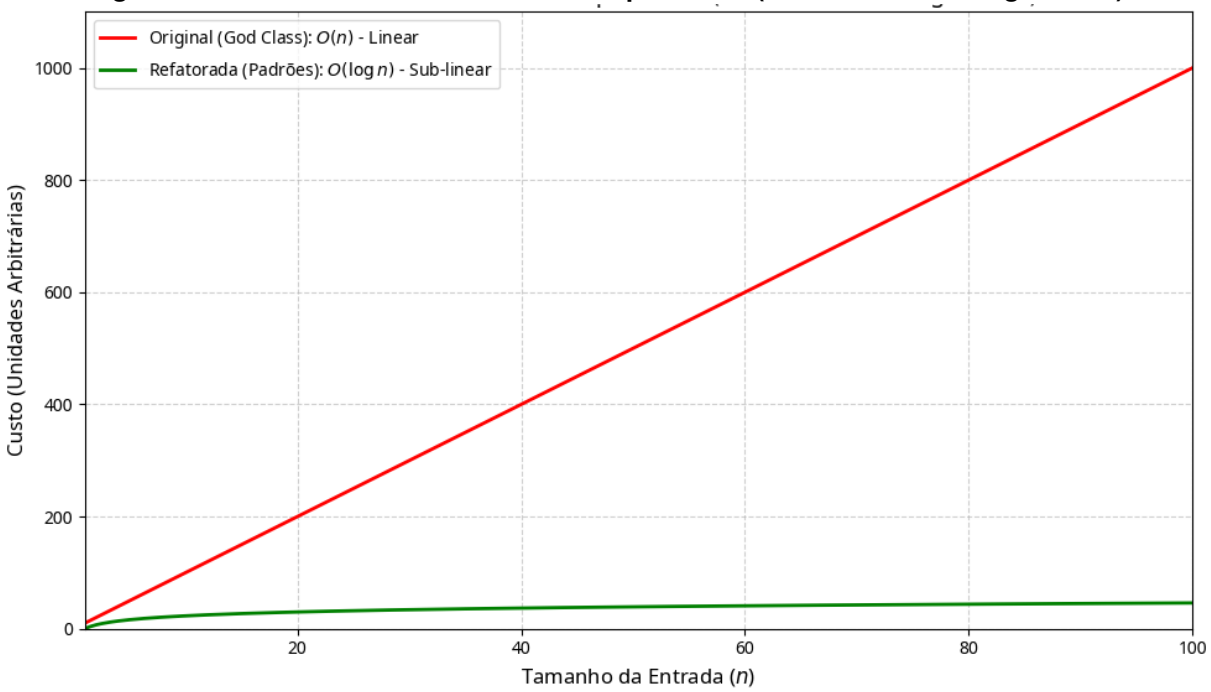
Fonte: Dados coletados pelos autores (2025)

Essa análise permite identificar os pontos de equilíbrio (*break-even points*): a complexidade da versão refatorada se torna menor a partir de $n \approx 4$ importadores (Figura 8), e a dívida

técnica se torna menor a partir de $n = 8$ importadores (Figura 9).

Finalmente, a Figura 10 projeta essa diferença a longo prazo, contrastando a ordem de complexidade $O(n)$ da "Versão Original" com a ordem $O(\log n)$ da arquitetura com padrões, provando a insustentabilidade do primeiro método.

Figura 10. Análise Teórica: Ordem de Complexidade (Crescimento a Longo Prazo)



Fonte: Dados coletados pelos autores (2025)

5. Conclusão

Este trabalho respondeu à questão de pesquisa ao demonstrar que os padrões de projeto são instrumentos essenciais para traduzir os conceitos teóricos de alta coesão e baixo acoplamento em uma arquitetura de software prática e mensurável. A hipótese foi confirmada: a aplicação de padrões (*Template Method*, *Factory*, *Strategy*) resultou em um sistema comprovadamente mais modular, extensível e sustentável.

A análise comparativa validou esta tese em três frentes:

- Métricas de Acoplamento (CKJM): Houve uma redução drástica e inequívoca do acoplamento, refletida nos índices das camadas de *Orquestradores* (CBO -35%) e *Implementadores* (CBO -33% e RFC -31%).
- Qualidade Holística (SonarQube): A manutenibilidade evoluiu da nota 'E' (Crítica) para 'A' (Excelente).
- Escalabilidade e Custo (Simulação): A simulação quantificou o *trade-off* de escalabilidade. A "Versão Original" (God Class) cresce a uma taxa linear acentuada de ≈11.5 minutos de Dívida Técnica e ≈76.5 WMC por novo importador (Funções $D_o(n) \approx 11.5n$ e $W_o(n) \approx 76.5n$). Em contraste, a "Versão Refatorada" possui um custo inicial (intercepto b) de 68 minutos e 265 WMC, mas sua taxa de crescimento é drasticamente menor: ≈3 minutos e ≈10 WMC por importador (Funções $D_r(n) \approx 3n + 68$ e $W_r(n) \approx 10n + 265$).

O investimento inicial na aplicação de padrões de projeto não é apenas uma melhoria teórica; ele se paga rapidamente em termos mensuráveis. A análise das funções de custo estabelece pontos de equilíbrio (*break-even points*) claros: a complexidade total da arquitetura refatorada se torna menor a partir de 4 importadores, e a dívida técnica total se torna menor a partir de 8 importadores. Isso prova que, para sistemas com qualquer expectativa de crescimento, a arquitetura com padrões é a escolha técnica e economicamente superior, resultando em um sistema comprovadamente mais modular, extensível, coeso e, acima de tudo, sustentável a longo prazo.

É importante ressaltar que, por se tratar de um estudo de caso único, os resultados desta pesquisa são contextuais. A análise limitou-se ao domínio de um único sistema (o *SheetReader*) e a um conjunto específico de padrões GoF.

Para uma maior validação e generalização dos achados, são necessários trabalhos futuros que repliquem esta metodologia em outros domínios de software e com diferentes combinações de padrões de projeto. Sugere-se, ainda, a aplicação de estudos que avaliem o impacto dos padrões em outras métricas de qualidade, como desempenho ou consumo de energia, áreas que a própria revisão bibliográfica deste estudo identificou como carentes de investigação.

Referências

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

Ali, M. and Elish, M. (2013). A comparative literature survey of design patterns impact on software quality. In *Int. Conf. on Information Science and Applications (ICISA)*, pages 1–7.

Ampatzoglou, A., Charalampidou, S., and Stamelos, I. (2013a). Research state of the art on gof design patterns: a mapping study. *J. Syst. Softw.*, 86(7):1945–1964.

Ampatzoglou, A., Charalampidou, S., and Stamelos, I. (2013b). Research state of the art on gof design patterns: a mapping study. *J. Syst. Softw.*, 86(7):1945–1964.

Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C., and Di Penta, M. (2007a). An empirical study on the evolution of design patterns. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 385–394.

Aversano, L., Cerulo, L., and Di Penta, M. (2007b). Relating the evolution of design patterns and crosscutting concerns. In *Seventh IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM 2007)*, pages 180–192.

Aversano, L., Cerulo, L., and Di Penta, M. (2009). Relationship between design patterns defects and crosscutting concern scattering degree: an empirical study. *IET Software*, 3(5):395–409.

Bieman, J., Jain, D., and Yang, H. (2001a). OO design patterns, design structure, and program changes: an industrial case study. *Proc. IEEE Int. Conf. on Software Maintenance*, 24(3):580–589.

Bieman, J., Jain, D., and Yang, H. (2001b). OO design patterns, design structure, and program changes: an industrial case study. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 580–589.

Bieman, J., Straw, G., Wang, H., Munger, P., and Alexander, R. (2003). Design patterns and change proneness: an examination of five evolving systems. In *Proceedings of the Ninth International Software Metrics Symposium*, pages 40–49.

Booch, G. (1994). *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA.

Cacho, N., Sant'Anna, C., Figueiredo, E., Dantas, F., Garcia, A., and Batista, T. (2014). Blending design patterns with aspects: a quantitative study. *Journal of Systems and Software*, 98:117–139.

Campbell, G. A. (2017). Cognitive complexity: The new way of measuring understandability. In *Proceedings of the International Conference on Technical Debt 2017*. SonarSource SA.

Eaddy, M., Aho, A., and Murphy, G. (2007). Identifying, assigning, and quantifying crosscutting concerns. In *Proc. First Int. Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07)*, page 2.

Eaddy, M., Zimmermann, T., and Sherwood, K. e. a. (2008). Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515.

El Emam, K., Benlarbi, S., Goel, N., and et al. (2001). The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng.*, 27(7):630–650.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional, Reading, MA.

Freeman, E., Freeman, E., Sierra, K., and Bates, B. (2004). *Head First Design Patterns*. O'Reilly.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., and von Staa, A. (2006). Modularizing design patterns with aspects: a quantitative study. *Transactions on Aspect-Oriented Software Development I*, pages 36–74.

Gatrell, M. and Counsell, S. (2011). Design patterns and fault-proneness a study of commercial c software. *Fifth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–8.

Gatrell, M. and Counsell, S. (2012). Faults and their relationship to implemented patterns, coupling and cohesion in commercial C software. *International Journal of Information System Modeling and Design (IJISMD)*, 3(2):69–88.

Gatrell, M., Counsell, S., and Hall, T. (2009). Design patterns and change proneness: a replication using proprietary c software. In *16th Working Conf. on Reverse Engineering (WCRE'09)*, pages 160–164.

Gravino, C., Risi, M., Scanniello, G., and Tortora, G. (2011). Does the documentation of design pattern instances impact on source code comprehension? Results from two controlled experiments. In *18th Working Conference on Reverse Engineering (WCRE)*, pages 67–76.

Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 161–173.

Hussain, S., Keung, J., and Khan, A. (2017). The effect of gang-of-four design patterns usage on design quality attributes. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 263–273.

Izurieta, C. and Bieman, J. (2013). A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal*, 21(2):289–323.

Khomh, F. and Guéhéneuc, Y. (2008). Do design patterns impact software quality positively? In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 274–278.

Leger, P. and Fukuda, H. (2014). Why do developers not take advantage of the progress in modularity? In *Proc. 8th Int. Conf. on Bioinspired Information and Communications Technologies*, pages 388–389.

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Mayvan, B., Rasoolzadegan, A., and Yazdi, Z. (2017). The state of the art on design patterns: A systematic mapping of the literature. *J. Syst. Softw.*, 125:93–118.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software engineering*, (4):308–320.

Prechelt, L., Unger, B., and Tichy, W. e. a. (2001). A controlled experiment in maintenance: comparing design patterns to simpler solutions. *IEEE Trans. Softw. Eng.*, 27(12):1134–1144.

Pree, W. (1994). Meta patterns – a means for capturing the essentials of reusable object-oriented design. In *European Conf. on Object-Oriented Programming (ECOOP'94)*, pages 150–162.

Pressman, R. S. (2016). *Engenharia de Software: uma abordagem profissional*. AMGH, Porto Alegre, 8 edition.

Riaz, M., Breaux, T., and Williams, L. (2015). How have we evaluated software pattern application? a systematic mapping study of research design practices. *Inf. Softw. Technol.*, 65:14–38.

Roo, A., Hendriks, M., and Havinga, W. e. a. (2008). Compose*: a language-and-platform-independent aspect compiler for composition filters. In *Int. Workshop on Advanced Software Development Tools and Techniques*, pages 1–14.

Scanniello, G., Gravino, C., Risi, M., Tortora, G., and Dodero, G. (2015). Documenting design-pattern instances: a family of experiments on source-code comprehensibility. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):14.

Sfetsos, P., Ampatzoglou, A., Chatzigeorgiou, A., Deligiannis, I., and Stamelos, I. (2014). A comparative study on the effectiveness of patterns in software libraries and standalone applications. In *9th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 145–150.

Stevens, W. P., Myers, G. J., and Constantine, L. L. (1974). Structured design. *IBM Systems Journal*.

Subramanyam, R. and Krishnan, M. (2003). Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):291–301.

Tanter, E., Figueroa, I., and Tabareau, N. (2014). Execution levels for aspect-oriented programming: design, semantics, implementations and applications. *Sci. Comput. Program.*, 80:311–342.

Van Roy, P. and Haridi, S. (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA.

Wedyan, F. and Abufakher, S. (2020). Impact of design patterns on software quality: a systematic literature review. *IET Software*, 14(1):1–17.

Wedyan, F., Ghosh, S., and Vijayasarathy, L. (2015). An approach and tool for measurement of state variable based data-flow test coverage for aspect-oriented programs. *Inf. Softw. Technol.*, 59:233–254.

Weiss, M. (2008). Patterns and their impact on system concerns. In *13th Annual European Conf. on Pattern Languages of Programming (EuroPLoP)*, pages S2–1–S2–10.

Zhang, C. and Budgen, D. (2012). What do we know about the effectiveness of software design patterns? In *IEEE Trans. Softw. Eng.*, volume 38, pages 1213–1231.