UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ALEX PEDROSO DE MORAES

Reengenharia de um sistema ERP visando sua testabilidade

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Profa. Dra. Érika Fernandes Cota

Porto Alegre
2018

AGRADECIMENTOS

À orientação da professora Érika Cota que contribuiu com seu conhecimento durante este ano e ao longo da minha graduação.

Aos professores do Instituto de Informática da UFRGS que colaboraram para a minha formação profissional.

Aos colegas de trabalho que participaram de muitas discussões sobre as ideias abordadas aqui.

E especialmente a minha namorada Tuíla Maciel pela paciência, apoio incondicional e companhia durante a realização desse trabalho.

RESUMO

Uma forma de reduzir os custos de manutenção de um sistema de software é a aplicação de técnicas de verificação e validação para garantir a sua qualidade. Dentre essas técnicas, testes unitários são especialmente recomendados devido a sua execução rápida. A aplicação de testes unitários exige que o sistema respeite alguns requisitos, em particular os de possuir baixo acoplamento e baixa complexidade ciclomática, para permitir que seus métodos sejam executados em isolamento durante o teste. O objetivo deste trabalho foi a aplicação da técnica de reengenharia em um sistema de software comercial com seis anos de mercado para adequá-lo a esses requisitos e permitir a adoção de testes unitários em seu processo de desenvolvimento. Foi realizada uma análise da comunicação interna da empresa, na qual foram detectados muitos defeitos no sistema encontrados por usuários, bem como atrasos na entrega de novas funcionalidades. Foi adotada a estratégia de reengenharia parcial para retrabalhar o módulo de emissão de notas fiscais do sistema, que consiste de um único método com mais de 400 linhas de código, uma complexidade ciclomática de 114 e 61 dependências, o que inviabiliza a adoção de testes unitários. Após a aplicação da técnica, o módulo de emissão de notas fiscais foi desmembrado em 33 métodos em 10 classes distintas, respeitando os princípios SOLID. Em média, o número de linhas de código desses novos métodos é de 5,03, com complexidade ciclomática 2,27 e 4,97 dependências por método. Com isso foi possível atingir 100% de cobertura sob o critério de cobertura de caminhos primos com apenas 68 casos de teste. Futuramente, é recomendado reavaliar o processo de desenvolvimento, coletando novamente dados sobre a incidência de defeitos e sobre a eficiência do processo de desenvolvimento para verificar o impacto da adoção dos testes unitários na produtividade da equipe.

Palavras-chave: Testes unitários. reengenharia. testabilidade.

Reengineering of an ERP system aiming for its testability

ABSTRACT

A way to reduce maintenance costs in a software system is the application of verification and validation techniques to ensure its quality. Of all those techniques, unit testing is especially recommended because of its fast execution. The use of unit tests demands the system to respect some requirements, mainly to have low coupling and low cyclomatic complexity, allowing its methods to be executed in isolation during testing. The primary goal of this study was the application of the reengineering in a commercial software system that is in the market for six years to comply with those requirements and allow the use of unit tests in its development process. The internal communication of the company was analyzed, and many user-reported failures and new features shipment delays were detected. The reengineering strategy was used to rework one specific module of the system ("emissão de notas fiscais"), which is a single method with more than 400 lines of code, cyclomatic complexity of 114 and 61 dependencies. After the application of the technique, the module was separated in 33 methods in 10 different classes, respecting SOLID principles. The number of lines of code of these new methods was, on average, 5.03, with a cyclomatic complexity of 2.27 and 4.97 dependencies per method. Coverage of 100% was achieved under the prime paths criteria through only 68 test cases. In the future, it is recommended the re-evaluation of the development process and a new data collection about the incidence of failures and the efficiency of the development process to check the impact of the use of unit testing on the productivity of the team.

Keywords: Unit testing, testability, reengineering.

LISTA DE ABREVIATURAS E SIGLAS

CFG	Control Flow Graph
ERP	Enterprise Resource Planning
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated development environment
MVC	Model-View-Controller
ORM	Object-Relational Mapping
SQL	Structured Query Language

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Sistemas de *software* precisam se modificar continuamente para que possam permanecer desempenhando a sua função e se manterem úteis aos seus usuários. Novas demandas, evolução tecnológica e correção de falhas são alguns dos motivos que levam um *software* a precisar passar por manutenções. De acordo com Rajlich and Bennett (2000), cerca de 50% do custo total de um *software* vem de manutenções pelas quais este passa durante a sua vida útil.

Uma forma de reduzir os custos de manutenção é através do uso de testes para possibilitar a detecção de falhas que possam ser introduzidas no sistema durante a implementação de alterações. Quanto mais cedo no processo uma falha é detectada, menor é o custo para a sua remoção, o que faz dos testes unitários uma ferramenta eficiente para este fim, uma vez que sua execução é rápida e pode ser feita a cada mudança.

A aplicação de testes unitários exige que o sistema seja composto de unidades fracamente acopladas e possua baixa complexidade ciclomática, caso contrário a implementação dos casos de testes pode ser custosa ou mesmo inviável. Essas características precisam estar presentes no sistema desde o seu projeto e, quando não estão, a introdução de testes unitários necessita de uma etapa prévia de readequação do sistema. Quando a decisão de inserir testes unitários é tomada em um sistema já em uso no qual esses critérios não são seguidos, é necessária uma forma sistemática de repensar o projeto do sistema.

A reconstrução do sistema desde o seu início nem sempre é uma opção economicamente viável. Reconstruir o sistema implica repensar decisões que muitas vezes não estão formalmente documentadas. A forma mais vantajosa de fazer essa readequação é através da aplicação da técnica de reengenharia, que aproveita o conhecimento já integrado ao sistema para reestruturá-lo.

Neste trabalho será analisado um sistema *web* de ERP (planejamento de recursos coorporativos) com seis anos de mercado. Para atender às necessidades dos usuários, o sistema passa por mudanças frequentes, visando tanto a adição de novas funcionalidades quanto a adaptação das funcionalidades existentes à realidade do processo produtivo dos clientes. Nos últimos dois anos, a empresa passou por um período de crescimento intensificado e a quantidade de clientes aumentou consideravelmente. Isso gerou um aumento na quantidade de requisições de mudanças por parte dos clientes, o que se refletiu em um aumento significativo no número de falhas encontradas, que passaram a dificultar o processo de adição de novas funcionalidades.

Até esse momento, a empresa não adota testes unitários no seu processo de desenvolvimento, de modo que as falhas são, muitas vezes, identificadas pelos próprios clientes, o que aumenta o custo do desenvolvimento. Uma vez que o sistema não foi projetado para ser testado, é necessário que sejam realizadas transformações antes da adoção de testes unitários. Devido ao tamanho do sistema e à complexidade de suas funcionalidades, essas modificações não podem ser pontuais, sendo indispensável a utilização de uma técnica mais sistematizada de adaptação do projeto, como a técnica de reengenharia.

1.1 Objetivos

O objetivo deste trabalho é projetar e implementar testes unitários no processo produtivo de um *software* de ERP com seis anos de mercado que não adota atualmente nenhuma estratégia formal de testes em seu processo. Para tanto, será proposta uma estratégia de reengenharia para colocar a estrutura interna do sistema em um estado em que seja possível a aplicação de testes unitários. Durante a reengenharia, também será apresentada uma proposta de arquitetura para o sistema que seja testável e que esteja em consonância com as práticas atualmente adotadas na indústria.

1.2 Estrutura do trabalho

No capítulo 2, encontram-se conceitos essenciais para a compreensão do trabalho realizado. São descritos aspectos fundamentais de evolução e qualidade de *software* e como estes se relacionam com a necessidade de um processo automatizado de teste. Também são destacados os passos para a realização de um procedimento de reengenharia em um sistema em funcionamento. São revisadas técnicas para geração de casos de teste com ênfase no estabelecimento de critérios de cobertura para testes unitários. Por fim, são realçadas as métricas utilizadas para aferir testabilidade e é introduzida uma proposta de arquitetura para o sistema alvo.

No capítulo 3, é feita uma descrição do sistema e das tecnologias envolvidas na sua implementação. Em seguida, são apresentados os resultados da análise das métricas de qualidade e uma descrição do módulo que foi alvo desta reengenharia. É apresentada uma visão detalhada da implementação da funcionalidade destacando os pontos de melhoria. Além disso, é descrita a estratégia de aceitação que será usada para validar as alterações

feitas durante a reengenharia.

No capítulo 4, são descritos os procedimentos adotados para a renovação do sistema visando resolver os problemas encontrados na análise e de acordo com a estratégia planejada. Também é apresentado o procedimento adotado para a geração de casos de teste. Os estados inicial e final do sistema são comparados e discutidos.

Por fim, são apresentadas as considerações finais e sugestões de prosseguimento do trabalho realizado.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Evolução e qualidade de *software*

Evolução de um *software* é o processo de modificação progressiva dos requisitos funcionais e não-funcionais do sistema. Atualizações de hardware ou de outros sistemas acessórios, como bancos de dados, e aumento na demanda por performance devido ao aumento no número de usuários são exemplos de alterações no ambiente que afetam os requisitos não-funcionais do sistema. Requisitos funcionais mudam, entre outros motivos, quando a necessidade que o *software* atende muda, quando o usuário demanda o atendimento de outras necessidades ou quando o uso do sistema faz com que a percepção que o usuário tem da sua necessidade mude. Em qualquer desses casos, a mudança nos requisitos faz com que todo ou parte do sistema precise passar por um retrabalho.

Por estes motivos, é comum que os custos de manutenção de um *software* superem os custos do seu desenvolvimento inicial (JONES; BONSIGNOUR, 2011). Assim, a qualidade interna traz benefícios econômicos para um produto de *software*. Além disso, um *software* com alta qualidade interna também demanda menos investimento em suporte ao consumidor, pois tende a apresentar menos defeitos durante a operação (JONES; BONSIGNOUR, 2011), tanto na funcionalidade apresentada quanto no atendimento de requisitos de usabilidade ou performance.

Em geral, o custo de correção de uma falha aumenta com o tempo decorrido entre a sua introdução no sistema e a sua detecção (PEZZè; YOUNG, 2007). Isso se dá devido ao retrabalho necessário, que aumenta à medida que aumentam as dependências entre os outros módulos do sistema com o módulo que apresenta a falha. Assim, investir em qualidade desde as primeiras etapas do processo é uma forma eficaz de reduzir custos, apresentando uma economia média de 50% e adiantando em cerca de 20% o tempo de entrega (JONES; BONSIGNOUR, 2011) para o desenvolvimento de novos sistemas ou de novas funcionalidades.

Outro obstáculo da indústria de *software* são as elevadas taxas de cancelamento de projetos. Grandes quantidades de retrabalho necessárias para a remoção de falhas detectadas tardiamente no processo podem alavancar o custo do desenvolvimento acima do orçamento do projeto ou atrasar a entrega para além do ponto em que o produto traria valor para o usuário. Estas situações podem gerar cancelamentos que custam caro, uma vez que suas causas só são detectadas nos estágios finais do projeto. Processos focados

em qualidade interna resultam em projetos com menor chance de serem cancelados por perderem o valor econômico e, quando um cancelamento é inevitável, ele acontece mais cedo no processo, quando ainda não foram investidas grandes somas no projeto (JONES; BONSIGNOUR, 2011).

Uma vez entregue, o *software* precisa manter o seu valor durante todo o tempo em que o usuário mantiver as suas necessidades. Porém, o ambiente onde o *software* atua geralmente é dinâmico e o *software* precisa evoluir para responder às mudanças na realidade em que se encontra inserido. Isso faz com que o desenvolvimento de um *software* seja um processo evolutivo contínuo que se estende durante toda a sua vida útil (SOMMERVILLE, 2011).

2.2 Reengenharia

Um processo que busque a qualidade é essencial para elevar o valor econômico de um produto de *software* e para reduzir seus custos de produção e manutenção. É possível argumentar que, para *software*, investimento em qualidade interna apresenta um retorno sobre investimento maior que qualquer outro fator conhecido (JONES; BONSIGNOUR, 2011). Um dos fatores mais importantes para alcançar esta característica é uma alta eficiência na detecção de falhas. Uma estratégia amplamente utilizada na indústria para identificação precoce de falhas é a aplicação de testes unitários durante o desenvolvimento.

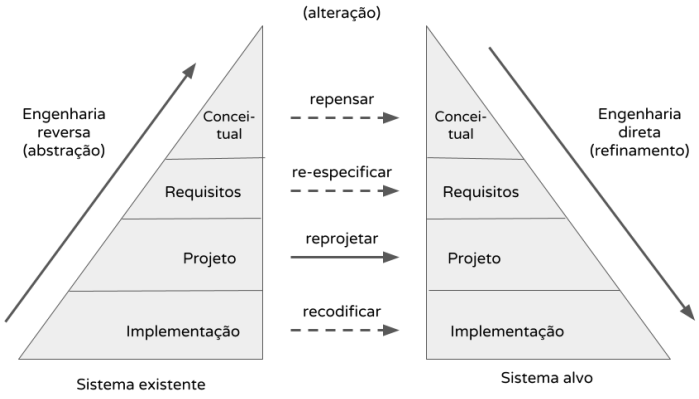
Quando o projeto de um sistema apresenta um alto grau de acoplamento (dependências entre classes concretas) e alta complexidade ciclomática, a aplicação de testes unitários fica comprometida. Nesse caso, é necessária a readequação do projeto aos critérios de testabilidade. Uma vez que reescrever um *software* desde o início raramente é viável, utilizar uma estratégia de reengenharia adequada é uma forma mais eficiente de adequar o *software* às novas exigências.

Chikofsky and Cross (1990) definem reengenharia como a análise e alteração de um sistema para reconstituí-lo em uma nova forma, seguidas da implementação dessa nova forma. Sendo assim, é possível aplicar reengenharia para reconstituir o sistema em uma forma que facilite a aplicação de testes unitários. Dado o objetivo deste trabalho, é possível utilizar a redução do acoplamento e da complexidade ciclomática como um critério para guiar o processo de reengenharia.

Byrne (1992) propõe um processo de reengenharia baseado nos princípios de abstração, alteração e refinamento. A partir destes fundamentos conceituais é possível cons-

truir uma estratégia de reengenharia que seja a mais adequada ao tipo de melhoria pretendida. A Figura 2.1 representa essas estratégias. Partindo da implementação, se reconstrói o projeto, a especificação de requisitos ou o modelo conceitual do sistema através de engenharia reversa. Tendo a representação no nível de abstração desejado se faz a alteração dessa representação. De posse da nova descrição de alto nível do sistema se procede o seu refinamento através da engenharia direta. Segundo o modelo de processo definido por Byrne and Gustafson (1992), o processo de reengenharia é composto de cinco fases distintas: análise e planejamento, renovação, teste de aceitação, redocumentação, aceitação e transição. Neste trabalho, o objetivo será reprojetar o sistema partindo do seu código-fonte de forma a acomodar os testes unitários.

Figura 2.1: Modelo geral para reengenharia de software.



Fonte: Adaptado de (BYRNE, 1992)

2.3 Teste de software

Teste é uma técnica de verificação que consiste em avaliar o comportamento do sistema através da observação da sua execução. Verificação é a checagem da conformidade da implementação com relação à especificação, que aqui se refere a papeis e não a artefatos particulares (PEZZè; YOUNG, 2007). Testes podem ser feitos manualmente, porém testes automatizados apresentam diversas vantagens, como reprodutibilidade e agilidade, e são largamente utilizados. Além disso, os testes automatizados podem ser armazenados para serem executados sempre que uma alteração for feita no sistema, como uma forma de garantir o funcionamento durante a evolução deste. Essa estratégia é chamada de teste de regressão.

Testes podem ser classificados de acordo com o seu escopo e o tipo de falha que

se busca identificar. Em particular destacam-se os testes de aceitação, de integração e unitários. De acordo com Ammann and Offutt (2008), testes de aceitação verificam se o sistema desenvolvido está de acordo com a especificação de requisitos. Testes de integração avaliam se as interfaces entre os módulos do sistema estão corretas e se são usadas corretamente. Testes unitários avaliam o comportamento das unidades produzidas durante o desenvolvimento. No caso de programação orientada a objetos, via de regra, essas unidades são classes.

Por serem executados contra as menores unidades testáveis do sistema, testes unitários apresentam a vantagem de poderem ser executados rapidamente. Isso permite que esses testes sejam executados a cada alteração no sistema apontando falhas introduzidas no código muito cedo no processo. Por outro lado, a exigência de que classes sejam testadas em isolamento torna difícil a aplicação dos testes a classes que possuem um alto grau de acoplamento.

Uma forma de reduzir o acoplamento entre as classes e melhorar a testabilidade é fazer com que as classes não dependam de implementações específicas e sim de abstrações. Esse princípio é conhecido como princípio da inversão de dependência (MARTIN, 2000). Em projetos que seguem esse princípio é possível fornecer a implementação desejada das dependências em tempo de execução, podendo esta ser uma implementação real da aplicação ou um *fake*.

Fakes são implementações falsas de uma classe do sistema usada como valor de entrada para os testes. *Fakes* fornecem para o sistema em teste o comportamento esperado sem se relacionar com o restante da aplicação. A complexidade de um *fake* pode variar desde uma implementação vazia que retorna para o sistema em teste um valor pré-definido até um comportamento mais complexo que permite coletar informações sobre como o *fake* foi utilizado dentro do sistema em teste. Nesse último caso é comum que a implementação falsa seja chamada de *mock object*.

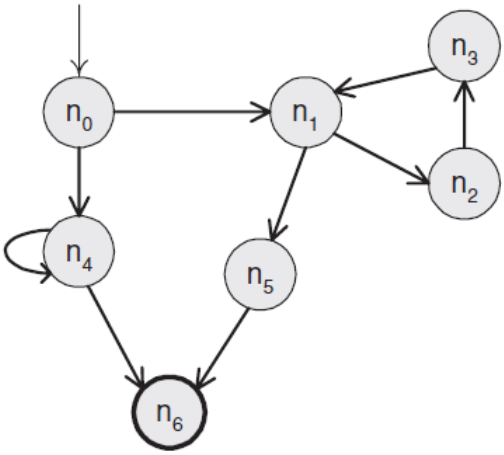
De acordo com Beizer (1995), o projeto de testes deve seguir uma estratégia bem definida, ou seja, deve existir um método sistemático para a produção de casos de teste. Um caso de teste é composto dos valores de entrada para a execução do teste e de todos os ajustes necessários para colocar o sistema no estado apropriado para receber os valores de entrada e da saída esperada. Ammann and Offutt (2008) definem essa estratégia sistemática como um "critério de cobertura" que gera requisitos que devem ser satisfeitos por pelo menos um dos testes do conjunto.

Dentre os critérios de cobertura que podem ser utilizados durante o processo de

desenvolvimento no projeto de testes, os critérios de cobertura de grafos podem ser aplicados sobre o grafo de fluxo de controle (do inglês *control flow graph* – CFG) de métodos do sistema para gerar requisitos para testes unitários baseados em seu código-fonte. Nesse caso, os critérios se diferenciam por quais elementos do grafo são considerados requisitos. Algumas possibilidades de critérios de cobertura de grafos são: nodos, arestas, pares de arestas, caminhos ou caminhos primos. Dentre estes, destaca-se o critério de caminhos primos. Um caminho primo entre dois nodos é um caminho simples (um caminho que não passa mais de uma vez por um mesmo nodo intermediário) entre esses nodos que não está contido em nenhum outro caminho simples. De acordo com os objetivos desse trabalho, o uso desse critério é mais vantajoso com relação a outros pois gera requisitos mais abrangentes no caso de estruturas de decisão mais complexas.

Um exemplo de como calcular caminhos primos de um grafo é apresentado nas Figuras 2.2, 2.3 e 2.4. Inicialmente tomam-se todos os caminhos de comprimento 0 (itens 1 a 7 na Figura 2.3). Para obter os caminhos de comprimento n+1, estendem-se os caminhos de comprimento n seguindo as arestas do último nó. Caso o nó alvo já faça parte do caminho de comprimento n, o caminho de comprimento n+1 é desconsiderado, pois não é um caminho simples. Quando o último nó de um caminho é igual ao primeiro nó deste, o caminho não pode mais ser estendido, pois deixaria de ser um caminho simples (itens 14, 27, 28 e 30). Isso se repete até que todos os caminhos simples sejam listados. Os caminhos primos são aqueles que, dentre caminhos listados, não estão contidos em nenhum dos outros caminhos (Figura 2.4) (AMMANN; OFFUTT, 2008). Os caminhos do grafo são representados como uma sequência dos números dos nodos entre colchetes (Figura 2.3).

Figura 2.2: Exemplo de CFG para o cálculo dos caminhos primos



Fonte: (AMMANN; OFFUTT, 2008)

Figura 2.3: Caminhos simples no grafo da Figura 2.2

1) [0]
2) [1]
3) [2]
4) [3]
5) [4]
6) [5]
7) [6] !
8) [0, 1]
9) [0, 4]
10) [1, 2]
11) [1, 5]
12) [2, 3]
13) [3, 1]
14) [4, 4] *
15) [4, 6] !
16) [5, 6] !
17) [0, 1, 2]
18) [0, 1, 5]
19) [0, 4, 6] !
20) [1, 2, 3]
21) [1, 5, 6] !
22) [2, 3, 1]
23) [3, 1, 2]
24) [3, 1, 5]
25) [0, 1, 2, 3] !
26) [0, 1, 5, 6] !
27) [1, 2, 3, 1] *
28) [2, 3, 1, 2] *
29) [2, 3, 1, 5]
30) [3, 1, 2, 3] *
31) [3, 1, 5, 6] !
32) [2, 3, 1, 5, 6]!

Fonte: (AMMANN; OFFUTT, 2008)

Figura 2.4: Caminhos primos no grafo da Figura 2.2

14) [4, 4] *
19) [0, 4, 6] !
25) [0, 1, 2, 3] !
26) [0, 1, 5, 6] !
27) [1, 2, 3, 1] *
28) [2, 3, 1, 2] *
30) [3, 1, 2, 3] *
32) [2, 3, 1, 5, 6]!

Fonte: (AMMANN; OFFUTT, 2008)

Pelas razões apresentadas, fica evidente que um processo de desenvolvimento de *software* necessita de uma infraestrutura sólida de testes automatizados. Testes são uma forma eficiente de detecção de falhas e contribuem para a qualidade interna, reduzindo os custos de desenvolvimento e manutenção e aumentando o valor do produto. A capacidade de detecção rápida de falhas também favorece a evolução ao garantir que defeitos não serão introduzidos em funcionalidades já entregues e ao permitir a manutenção da estrutura interna do código via refatoração.

2.4 Métricas

Métricas são medidas de atributos das entidades de um *software* e servem para avaliar diversos aspectos do sistema, como por exemplo custos, confiabilidade e flexibilidade (COOK, 1982). São importantes como medidas quantitativas da adequação do *software* a determinada característica desejada, permitindo que sejam tomadas decisões no sentido de melhorar e refinar o produto (BOEHM et al., 1978).

Existem diversos tipos de métricas de *software*, que atendem a necessidades variadas. Elas podem ser objetivas ou subjetivas, primitivas ou compostas, de produto ou de processo (MILLS, 1988). As métricas utilizadas neste trabalho podem ser caracterizadas

como objetivas, primitivas e de produto. Métricas objetivas independem das condições de coleta ou do julgamento de alguém e são obtidas por meio de procedimentos definidos e representados por valores. Métricas primitivas representam o resultado de uma única medida. Métricas de produto são obtidas a partir dos artefatos do *software* e podem ser dinâmicas ou estáticas. Métricas de produto estáticas são obtidas a partir de representações do sistema tais como código fonte e documentações e podem ser, dentre outras, medidas de tamanho, de complexidade e de orientação à objeto. Como exemplos dessas medidas temos número de linhas de código, complexidade ciclomática e acoplamento, respectivamente.

A complexidade ciclomática é uma métrica amplamente utilizada na engenharia de *software* para expressar o grau de testabilidade de um sistema (MCCABE, 1976). Seu cálculo é feito sobre o CFG que representa a lógica dos métodos do sistema. A complexidade ciclomática v de um grafo G é dada pela Equação 2.1, onde e é o número de arestas do grafo, n é o número de nós e p é o número de componentes conectados. O acoplamento é uma medida da interação entre classes do sistema e reflete a quantidade de classes das quais são chamados métodos ou consultados atributos (CHIDAMBER; KEMERER, 1994).

$$v(G) = e - n + 2 * p \tag{2.1}$$

2.5 Boas práticas de programação

Uma forma de se alcançar baixo acoplamento e baixa complexidade ciclomática é a adoção de princípios e padrões de projeto que favoreçam a qualidade, evitando anti-padrões (BROWN, 1998) e eliminando *bad smells* (FOWLER, 2004) encontrados no código. Nesta seção, serão discutidos os princípios e os padrões adotados nesse trabalho:

SOLID

SOLID (MARTIN, 2000) é um acrônimo que representa um conjunto de cinco princípios que guiam o desenvolvimento de *software* com o objetivo de alcançar um *software* que seja testável, de fácil manutenção e aberto a evolução. Os cinco princípios são:

- **Única responsabilidade:** as unidades do sistema devem ser responsáveis por uma

única função

- **Aberto/fechado:** as classes devem estar abertas para extensão e fechadas para modificação
- **Substituição de Liskov:** a substituição de uma dependência por uma de suas subclasses não deve afetar a correção do sistema
- **Segregação de interface:** restringir a abrangência da responsabilidade das interfaces
- **Inversão de dependências:** as classes devem depender de abstrações e não de classes concretas

Os princípios de substituição de Liskov e de inversão de dependências favorecem diretamente a testabilidade, uma vez que permitem o desacoplamento entre a funcionalidade em teste e o restante do sistema. Indiretamente, o princípio de única responsabilidade contribui para testabilidade, pois resulta em classes e métodos com menor complexidade.

Bad smells

Segundo Fowler (2004), *bad smells* são estruturas recorrentes encontradas em código de baixa qualidade que indicam pontos nos quais este pode ser melhorado. Esses aspectos do código podem indicar problemas estruturais mais profundos que sugerem uma degradação da qualidade do sistema. O autor oferece um catálogo de técnicas de refatoração para remover diferentes tipos de problemas assumindo que o sistema em questão possui um conjunto de testes unitários para validar alterações.

As técnicas de refatoração sugeridas por Feathers (2004) para lidar com *bad smells* são voltadas para o que o autor chama de código legado, ou seja, sistemas que não possuem nenhum tipo de teste. Ao contrário das técnicas propostas por Fowler (2004) que visam eliminar completamente os *bad smells*, a abordagem de Feathers (2004) propõe alterações mais conservadoras para reduzir a probabilidade de inserção de falhas no sistema.

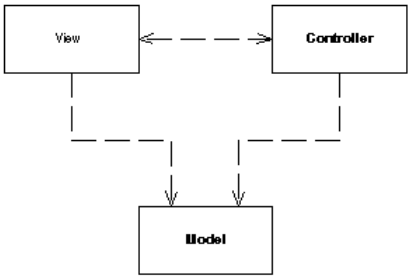
Padrões de projeto e de arquitetura

Um padrão é um modelo para a solução de um problema recorrente. Os padrões foram propostos como uma forma reutilizável de sistematizar a solução do problema e

são consideradas a formalização de boas práticas de projeto. Porém, é preciso levar em consideração a possibilidade de um modelo de solução não ser adequado por gerar consequências prejudiciais ao sistema, apesar de resolver o problema específico em questão. Nesse caso, essa soluções são chamadas de anti-padrões, e podem ocorrer por falta de planejamento, falta de conhecimento por parte da equipe ou pelo uso equivocado de padrões em situações onde estes não se aplicam (BROWN, 1998).

Como exemplo de padrões, podem ser citados MVC (*Model-View-Controller*) como padrão arquitetural e *singleton*, *abstract factory* e *unity of work* como padrões de projeto. O padrão MVC separa as responsabilidades da aplicação em três camadas com o objetivo de isolar a representação interna dos dados do que é apresentado na interface de usuário (Figura 2.5). A camada de apresentação (*View*) é responsável por apresentar para o usuário os dados oriundos da camada de modelo (*Model*). A camada de controle (*Controller*) é responsável por receber as entradas do usuário e manipular os dados do modelo.

Figura 2.5: Padrão *MVC*

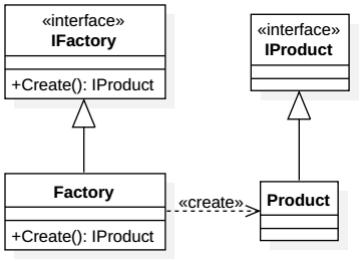


Fonte: (FOWLER, 2002)

O padrão *abstract factory* encapsula a lógica de instanciação de objetos em uma classe *factory* (Figura 2.6). Fazendo os métodos da classe *factory* retornarem abstrações, é possível permitir que os objetos clientes solicitem instâncias sem a necessidade de especificar qual a implementação da abstração será retornada. Além de reduzir o acoplamento do sistema, esse padrão centraliza as chamadas aos construtures dos objetos, evitando a repetição de código, especialmente em casos de construtores que necessitam de muitos parâmetros.

Outro padrão que busca meios de se obter instâncias de objetos sem chamadas explícitas aos seus construtores é o padrão *singleton* (Figura 2.7). Diferentemente do padrão *abstract factory*, o objetivo do *singleton* é fornecer a todo o sistema uma única instância de uma determinada classe. Para isso, o construtor da classe é tornado privado e é fornecido um método estático que retorna a instância do objeto. Este padrão é bastante

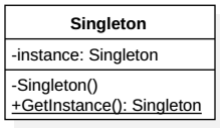
Figura 2.6: Padrão *abstract factory*



Fonte: Adaptado de (GAMMA et al., 1995)

utilizado para restringir a instanciação de classes que representam recursos físicos do sistema, porém pode ser considerado um anti-padrão por criar uma restrição em casos em que uma instância única não seja desejada como, por exemplo, no uso de *mocks* para a aplicação de testes unitários ao sistema.

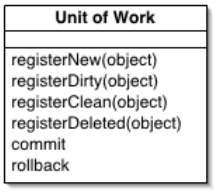
Figura 2.7: Padrão *singleton*



Fonte: Adaptado de (GAMMA et al., 1995)

O padrão *unity of work* (Figura 2.8) é uma forma de controlar o acesso à camada de persistência do sistema. A classe que representa a unidade de trabalho fornece métodos para leitura e escrita, bem como métodos para efetivar ou descartar as alterações realizadas. Esse padrão centraliza a lógica de acesso à camada de persistência, permitindo ao restante da aplicação o acesso às classes do modelo de dados sem a necessidade de manter o controle sobre o momento em que esses dados são persistidos.

Figura 2.8: Padrão *unit of work*



Fonte: (FOWLER, 2002)

O uso de padrões é importante, pois tratam-se de soluções conhecidas e amplamente discutidas pela comunidade de desenvolvedores. Isso dá ao projetista a segurança de que sua solução atenderá a requisitos mínimos de qualidade e manutenabilidade.

3 PROCESSO DE REENGENHARIA: ANÁLISE E PLANEJAMENTO

Na fase de análise e planejamento, foi avaliado o estado atual do sistema e foram traçadas as estratégias usadas durante o processo de reengenharia. Devido ao tamanho do sistema e à necessidade de avaliação do impacto do processo, a abordagem adotada foi a de reengenharia parcial (BYRNE; GUSTAFSON, 1992) na qual somente uma parte do sistema é retrabalhada de cada vez. Não foi abordada a melhoria na estrutura da interface de usuário: todas as alterações feitas no sistema foram somente no *back-end* da aplicação. Melhorias no projeto da interface de usuário são possíveis, mas não foram executadas neste trabalho.

O *plugin Resharper 2017.3.5* para *Visual Studio* foi usado para a obtenção de diagramas de dependências, que serviram como base para os diagramas de classes apresentados neste trabalho, e como ferramenta de inspeção estática de código durante o desenvolvimento. O *plugin* também se soprepõe à ferramenta de refatoração automática do *Visual Studio* e, portanto, foi usado para automatizar as refatorações mais simples. Os CFGs usados para o desenvolvimento dos testes unitários foram obtidos com a ferramenta *Understand (Build 939)* da empresa *SciTools*.

3.1 Descrição do sistema

O sistema alvo deste trabalho de reengenharia é um *software* de ERP, ou seja, é um sistema que integra diversas ferramentas para o gerenciamentdo das atividades no processo produtivo de uma empresa. O ERP em questão é uma aplicação *web* que conta com os seguites módulos:

- Vendas
- Compras
- Notas fiscais
- Estoque
- Engenharia e orçamentação
- Planejamento e produção
- Financeiro
- Contabilidade

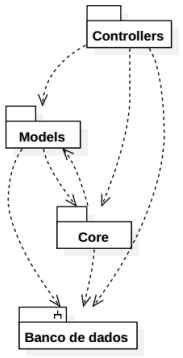
Dentre esses, o módulo de notas fiscais tem um papel central para o funcionamento do sistema. Através do módulo de notas fiscais, usuários podem interagir com o sistema das seguintes formas:

- **Criar e aprovar propostas**: Propostas aprovadas tornam-se pedidos de venda;
- **Criar e aprovar pedidos de venda**: Pedidos de venda aprovados geram necessidades para o módulo de planejamento e produção, que faz a alocação dos recursos materiais e de mão-de-obra para a produção dos itens vendidos;
- **Criar e aprovar cotações**: Cotações aprovadas tornam-se pedidos de compra;
- **Criar e aprovar pedidos de compra**: Pedidos de compra aprovados são considerados pelo módulo de planejamento e produção para estimar o tempo necessário de produção baseado na previsão de recebimento dos insumos necessários;
- **Receber notas fiscais**: Notas fiscais recebidas adicionam produtos ao estoque da empresa, geram entradas nos livros fiscais e criam contas a pagar;
- **Emitir notas fiscais**: Notas fiscais emitidas removem produtos do estoque da empresa, geram entradas nos livros fiscais e criam contas a receber;

É possível perceber que o módulo de notas fiscais interage diretamente com a maior parte dos outros módulos do sistema alimentando-os com dados e também consumindo os dados produzidos por eles. Todos esses tipos de documentos são tratados como notas fiscais no sistema, pois seus dados possuem certa similaridade e são armazenados em uma única tabela no banco de dados. Permitir que a estrutura do banco de dados permeie a lógica de negócios é uma má prática comum no desenvolvimento do sistema. Essa centralidade faz com que o módulo de notas fiscais seja aquele com o qual os usuários têm maior interação dentro do uso normal do sistema e, por consequência, o que necessita de correções e modificações com maior frequência. Por esse motivo, esse foi o módulo escolhido para ser coberto nesse trabalho.

Inicialmente, o sistema estava organizado de acordo com a arquitetura apresentada na Figura 3.1. A camada de *controllers* recebe as requisições HTTP e delega o processamento para a camada de classes utilitárias `Core`. A camada `Models` é composta por classes geradas automaticamente e são mapeadas para as tabelas do banco de dados. É possível perceber que todas as camadas possuem uma dependência com o banco de dados e isso prejudica a aplicação de testes unitários.

Figura 3.1: Arquitetura original do sistema



Fonte: O Autor

3.2 Tecnologias

O sistema é uma aplicação *web* cujo *back-end* é totalmente implementado em C# e usa o *framework* ASP.NET MVC que fornece ferramentas para o desenvolvimento de aplicações *web* usando o padrão MVC (*Model-View-Controller*). Para a criação das páginas *web* da camada de apresentação, o *framework* conta com uma linguagem de marcação chamada *Razor* que permite adicionar ao código HTML declarações em C# que são executadas pelo servidor apenas na geração da página.

A persistência dos dados é feita em um banco de dados *Oracle 11*. A comunicação com o banco de dados é feita usando a ferramenta de ORM *Entity Framework* que permite que seja feito um mapeamento entre as tabelas do banco de dados e classes que representam as entidades do domínio da aplicação que compõem a camada `Models` da Figura 3.1. O código gerado pela ferramenta, além de conter as classes mapeadas, contém também uma classe de contexto que funciona como uma implementação do padrão *unit of work*, disponibilizando métodos para o acesso aos repositórios de entidades do domínio. As declarações em SQL que correspondem às consultas e comandos gerados pelo *Entity Framework* são geradas pelo provedor de dados *dotConnect for Oracle* da *Devart*.

3.3 Descrição do processo de desenvolvimento

Foi feita uma análise da comunicação interna da empresa para demonstrar a necessidade de melhoria no código do sistema. Foram observadas as datas de especificação, de início do desenvolvimento, de fim do desenvolvimento e de avaliação. Neste caso, data de avaliação se refere a uma avaliação manual pela qual alguns desenvolvimentos

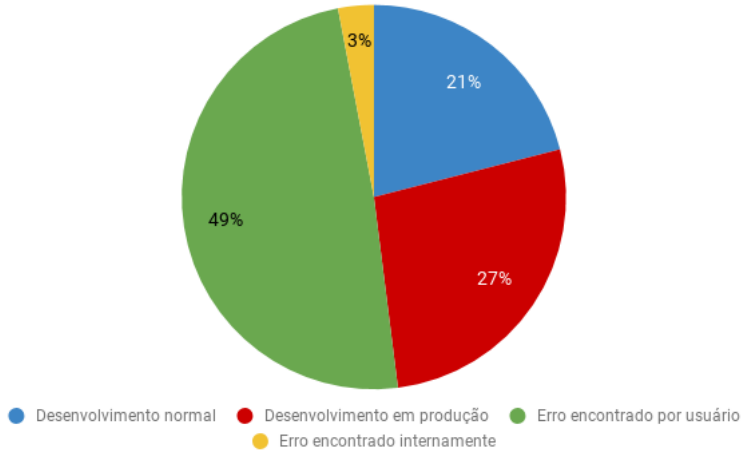
passam. Essa avaliação é feita de forma *ad-hoc* pela equipe de suporte ao cliente e não consiste em um teste sistemático da qualidade do que foi desenvolvido. Essa observação colheu dados a partir da contagem das solicitações de desenvolvimentos que tenham sido marcadas como concluídas pelo desenvolvedor no período de 08/02/18 a 13/03/18 e cada solicitação foi classificada em uma das seguintes categorias: (a) desenvolvimento normal, (b) desenvolvimento de urgência, (c) erro encontrado pelo cliente e (d) erro encontrado pela equipe.

Um desenvolvimento foi considerado normal quando sua entrega foi feita de acordo com o plano de atualização mensal do sistema para ser colocado em produção na atualização seguinte. Um desenvolvimento foi considerado de urgência sempre que tenha sido desenvolvido diretamente em produção sem ser uma correção de erro, burlando a agenda mensal de atualizações. Erros encontrados pelo cliente e pela equipe também são categorizados de acordo.

Desta análise, pode-se concluir que 49% das solicitações de desenvolvimento foram oriundas de erros encontrados pelos clientes. Em 25% de tudo o que foi desenvolvido no período não houve nem mesmo a avaliação manual *ad-hoc* anteriormente citada, significando que a funcionalidade saiu diretamente do desenvolvedor para o cliente sem passar por nenhum tipo de validação. Somente 5% dos erros corrigidos foram identificados internamente, sendo todos os outros encontrados pelos clientes. Essas informações estão apresentadas na Figura 3.2.

O tempo entre a especificação da necessidade e o início da sua implementação também foi considerado. Em média, um desenvolvimento que entra no ciclo padrão de atualizações leva 98 dias para começar a ser desenvolvido após a sua especificação. Funcionalidades desenvolvidas diretamente em produção têm um tempo médio de 29 dias entre especificação e início do desenvolvimento. Esses dados podem ser consequência do elevado número de correções necessárias após cada atualização. O tempo despendido em correções atrasa o desenvolvimento de novas funcionalidades. Dessa forma, quando uma nova funcionalidade crítica é planejada para uma dada atualização, é comum seu desenvolvimento se estender além da data em que a nova versão é lançada e sua entrega ser feita sem passar pelo ciclo planejado.

Figura 3.2: Distribuição dos tipos de manutenção no sistema



Fonte: O Autor

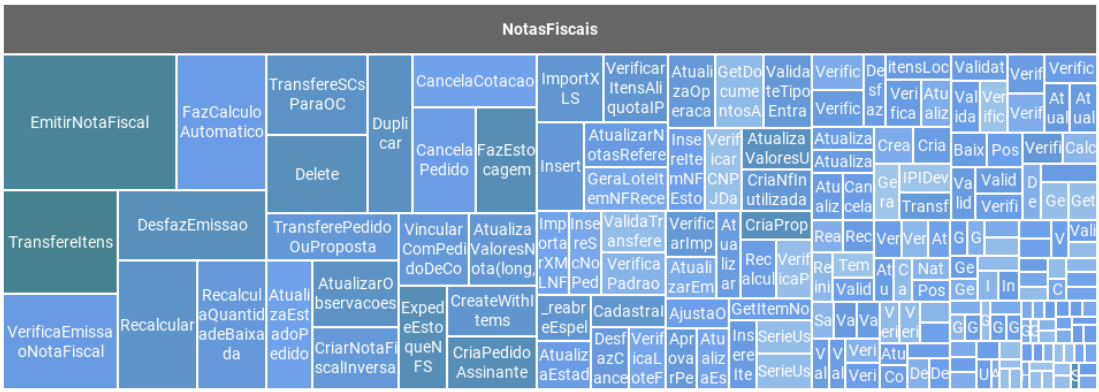
3.4 Métricas de qualidade

A IDE (ambiente integrado de desenvolvimento) *Visual Studio* dispõe de uma ferramenta de análise de métricas estáticas de código. Com ela, foram coletadas as métricas para analisar a testabilidade do sistema, visando identificar pontos de melhoria e, ao final, avaliar a eficácia do processo de reengenharia. Foram coletados dados de número de linhas de código, complexidade ciclomática e acoplamento do sistema.

Como o módulo de notas fiscais é o principal módulo do sistema, essa análise foi focada na classe responsável pela lógica desse módulo. O resultado pode ser visto na Figura 3.3 que apresenta um *treemap* de todos os métodos dessa classe. Um *treemap* é um gráfico que apresenta dados hierárquicos na forma de retângulos aninhados. Os retângulos que representam as folhas da hierarquia têm a área proporcional à dimensão dos dados e cores podem ser usadas como representação de alguma outra dimensão desses dados. No caso da Figura 3.3, os retângulos são métodos, sua área representa complexidade ciclomática e a cor indica o número de dependências. É possível notar que o método `EmitirNotaFiscal` é o método com a maior complexidade ciclomática desse módulo e também um dos métodos com o maior número de dependências. A partir dessa análise, foi tomada a decisão de se prosseguir com o trabalho de reengenharia sobre a funcionalidade de emissão de notas fiscais do sistema.

Os resultados da análise de métricas foram sintetizados na Tabela 3.1 que mostra os valores obtidos para o método `EmitirNotaFiscal`. Como pode ser visto na tabela,

Figura 3.3: *Treemap* das métricas da classe `NotasFiscais`.



A área dos retângulos é proporcional à complexidade ciclomática do método correspondente. Quanto maior o número de dependências do método, mais escura a cor do retângulo. Fonte: O Autor

o código apresenta uma complexidade ciclomática elevada, o que implica a necessidade de um número elevado de casos de teste para atingir o critério de cobertura de caminhos primos. A tabela também apresenta a medida obtida da métrica de acoplamento, que representa a quantidade de dependências do método. O valor obtido também se mostrou elevado. Isso implica em um grande número de variáveis que precisam ser controladas para a construção de cada caso de teste.

Tabela 3.1: Métricas de qualidade do método `EmitirNotaFiscal`.

Métrica	Valor
Linhas de código	431
Complexidade ciclomática	114
Acoplamento	61

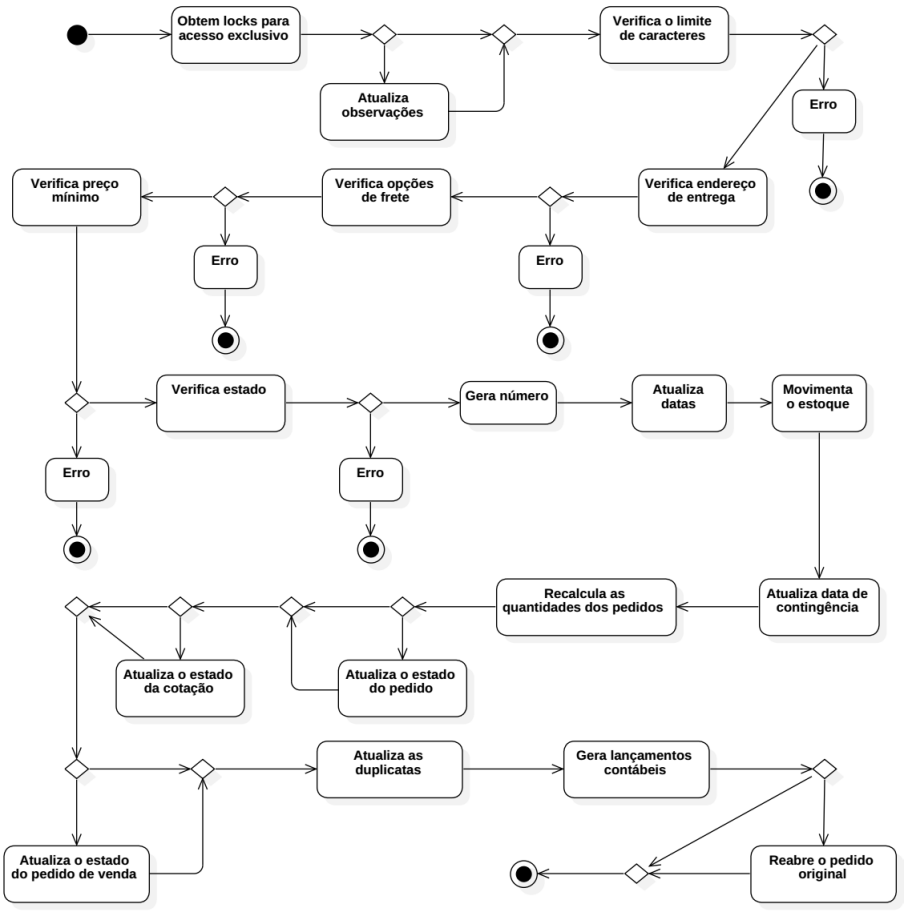
Fonte: O Autor

3.5 Descrição do método `EmitirNotaFiscal`

Apesar do nome, o método `EmitirNotaFiscal` é responsável pela emissão e recebimento de notas fiscais e emissão de pedidos de compra e venda, propostas e cotações. Neste trabalho todos esses documentos serão referenciados como notas fiscais, de forma genérica. O método é invocado quando um usuário requisita a emissão de uma nota fiscal previamente cadastrada no sistema e, além de efetuar a mudança no seu estado, desencadeia as ações necessárias para propagar essa alteração para os outros módulos. A

Figura 3.4 apresenta a sequência de ações executadas na emissão da nota fiscal.

Figura 3.4: Atividades da emissão de notas fiscais



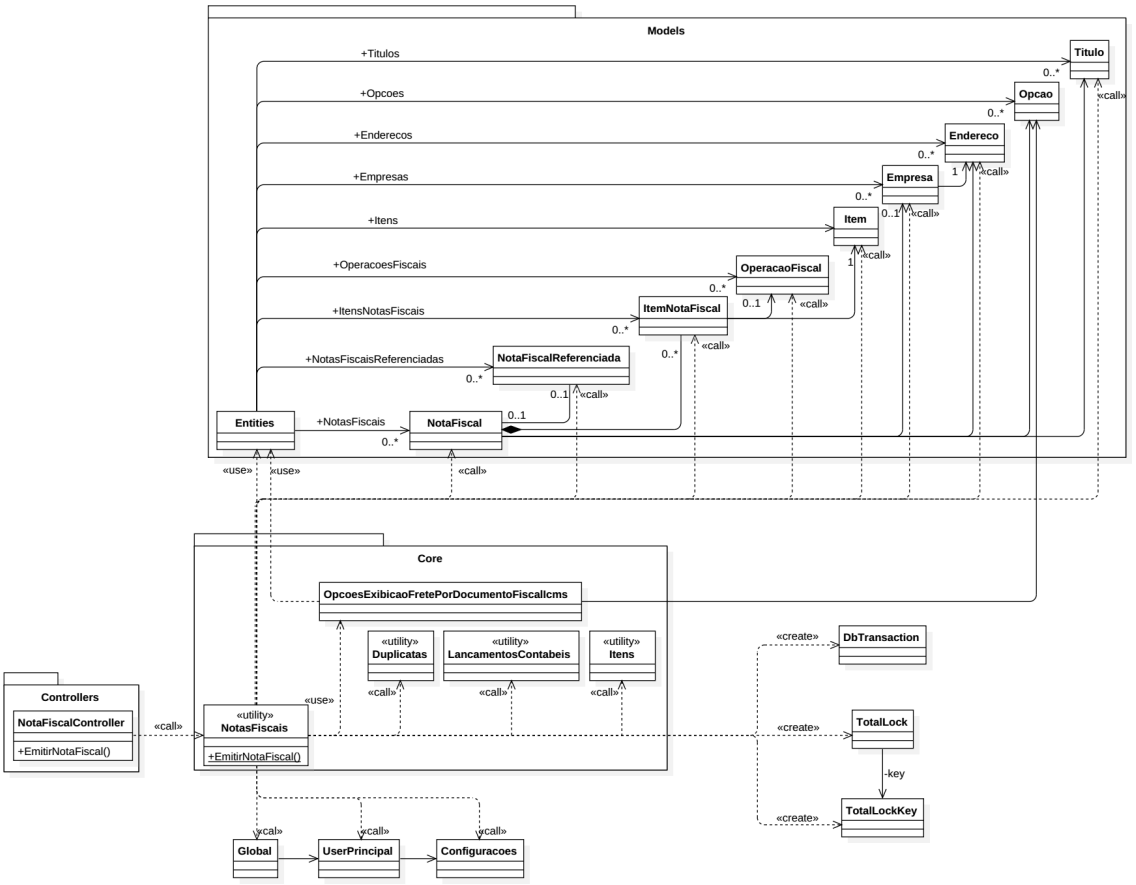
Fonte: O Autor

3.6 Implementação

A Figura 3.5 apresenta, em um diagrama de classes, as dependências do método `EmitirNotaFiscal`. Divergindo do padrão MVC, a implementação da lógica da aplicação se encontra implementada em uma camada adicional chamada de `Core`. Esse *namespace* é composto por classes utilitárias, ou seja, classes estáticas compostas de métodos públicos para serem utilizados por toda a aplicação.

O uso desse padrão de classes utilitárias aumenta o acoplamento do sistema, pois não é possível criar uma abstração para as classes estáticas, o que viola o princípio da inversão de dependências e dificulta a aplicação de testes unitários. Para eliminar esse problema, a lógica de aplicação contida nessas classes deve ser reimplementada em classes não-estáticas das quais possam ser extraídas abstrações.

Figura 3.5: Classes das classes envolvidas na emissão de notas fiscais



Fonte: O Autor

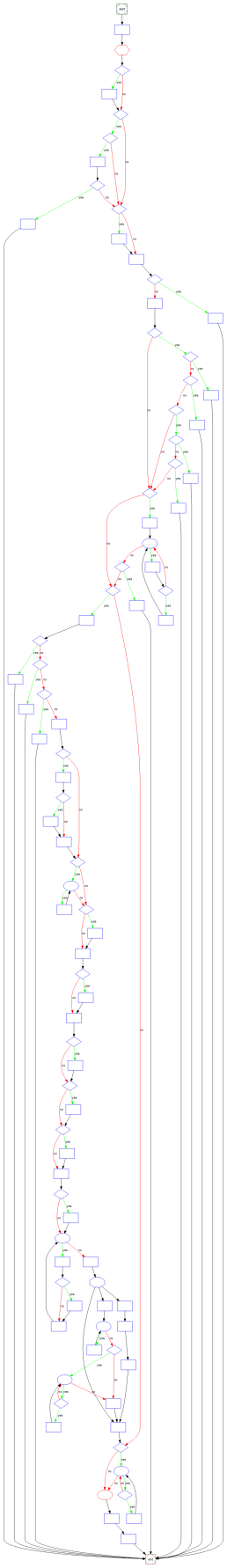
Outra característica importante apresentada na Figura 3.5 é uma dependência com a classe `Entities`. Esta classe é gerada pelo *Entity Framework* e realiza acessos ao banco de dados. Com o uso explícito dessa classe, não é possível exercitar o método em um teste unitário sem gerar consultas ao banco de dados, o que impede o controle total sobre o estado pré-existente do sistema antes do teste e também torna a sua execução lenta. Para evitar esses efeitos, também deve ser utilizado o princípio da inversão de dependências no acesso à essa classe.

Enquanto o diagrama de classes permite a visualização das dependências do método `EmitirNotaFiscal`, para compreender a origem da sua complexidade ciclomática podemos analisar o seu grafo de fluxo de controle, apresentado na Figura 3.6. Destaca-se na figura a grande quantidade de nós de decisão presentes no grafo. A maior parte dessas decisões se relacionam ao tipo de nota fiscal sendo emitida. O fato de um mesmo método ser responsável pela lógica de emissão de tipos diferentes de documento que exigem tratamentos distintos viola o princípio da única responsabilidade e se reflete no aumento na complexidade causado por essas decisões.

Durante a análise da implementação, foram encontradas algumas situações que dificultam a testabilidade e que devem ser removidas para adequar o código. Algumas se encaixam na classificação de *bad smells*, enquanto outras são violações dos princípios SOLID ou usos de anti-padrões. A seguir, estão listados os problemas encontrados:

- **Método longo:** quantidade de linhas de código.
- **"Inveja dos dados":** métodos que usam principalmente dados de suas dependências.
- **Dados agrupados:** dados que sempre são manipulados juntos e poderiam ser extraídos para outra classe.
- **Comandos *switch*:** uso excessivo de construções do tipo *switch*.
- **Classes de dados:** classes que não possuem comportamento e servem apenas para carregar dados.
- **Violação do princípio de única responsabilidade:** classes ou métodos que acumulam responsabilidades.
- **Uso de classe utilitária:** classes utilitárias podem ser consideradas anti-padrões.
- **Uso do padrão *singleton*:** *singleton* pode ser considerado anti-padrão.

Figura 3.6: Grafo de fluxo de controle do método `EmitirNotaFiscal`



Fonte: O Autor

3.7 Testes de aceitação

A partir dos requisitos funcionais do sistema recuperados nessa análise, foram construídos diagramas de atividades que descrevem o comportamento esperado do sistema na sua forma atual. Esses diagramas de atividades serviram, então, como base para a construção de um conjunto de casos de teste de sistema. Como o sistema é uma aplicação *web*, os testes foram executados a partir da sua interface com o uso do *framework Selenium* que permite automatizar a interação com o navegador. A aceitação da nova versão do sitema se deu atráves da constatação de que todos os casos de testes definidos nesta etapa foram executados com sucesso após a etapa de reengenharia.

O projeto dos testes para a aceitação das alterações teve como base o diagrama de atividades da Figura 3.4. É possível perceber que existe uma grande quantidade de caminhos no diagrama de atividades, apesar de não existirem laços. Por esse motivo foi feita a opção pelo critério de cobertura de pares de arestas para a geração desses casos de teste. Além disso, o conhecimento sobre os requisitos funcionais do sistema foi aplicado para identificar caminhos inviáveis no diagrama de atividades.

Foi possível observar que muitas das decisões representadas no diagrama de atividades possuem um único resultado possível uma vez que se tenha determinado qual dos quatro tipos de documento está sendo emitido. Com base nessa informação, o diagrama de atividades foi desmembrado em quatro diagramas de atividades, um para notas fiscais, um para notas fiscais recebidas, um para pedidos e um para propostas e cotações. Indo além, alguns caminhos nos grafos resultantes não puderam ser exercitados devido a validações anteriores que não permitem que alguns estados de erro previstos no diagrama de atividades sejam alcançados a partir da interface do sistema.

Cada um dos diagramas de atividades resultantes foi ainda separado em dois: um contendo todos os caminhos que resultam em uma mensagem erro para o usuário e outro contendo os caminhos que resultam em sucesso no processamento. Como a quantidade de efeitos colaterais que resulta do processo de emissão da nota fiscal é elevado, uma tentativa de se avaliar o estado completo do sistema após o processamento necessitaria de uma análise dos requisitos funcionais de todos os outros módulos e, portanto, foi tomada a decisão de não considerar esses efeitos colaterais ao se fazer a verificação do resultado da execução.

Para considerar um caso de teste como sucesso ou falha somente foram observadas as mensagens de erro que podem surgir na interface e as variáveis que compõem o estado

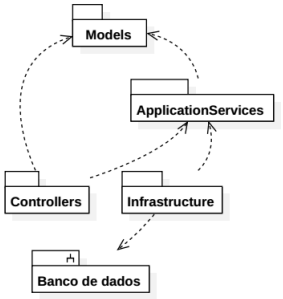
da própria nota que está sendo emitida. Feitas essas ressalvas, a aplicação do critério de cobertura resultou em um total de 14 casos de teste que foram implementados usando *Selenium* e, uma vez que todos tenham passado, armazenados para serem executados novamente após a renovação do sistema.

4 PROCESSO DE REENGENHARIA: IMPLEMENTAÇÃO

Na reengenharia, o módulo de emissão de notas fiscais foi reprojetado para acomodar os requistos de baixo acoplamento e baixa complexidade ciclomática necessários para se atingir o objetivo de testabilidade proposto para esse processo. Para tanto, foi feita uma engenharia reversa do sistema para se buscar uma representação do seu estado atual através do conhecimento obtido na etapa de análise. A partir da representação do estado atual, se buscou compreender o papel das diversas atividades desempenhadas pelo módulo, a maneira como os componentes do sistema se relacionam e os pontos onde intervenções no projeto seriam mais efetivas para se alcançar a testabilidade desejada.

A arquitetura proposta para o sistema está representada na Figura 4.1. A reimplementação do sistema segundo esse novo projeto foi feita de forma incremental usando as técnicas de refatoração propostas por Fowler (2004) e Feathers (2004) para reduzir o risco de se introduzir falhas na nova versão do sistema.

Figura 4.1: Arquitetura final do sistema



Fonte: O Autor

Nas próximas sessões, serão descritos os procedimentos realizados para eliminar os problemas encontrados na análise da implementação visando adequar o sistema aos requisitos de testabilidade. Para isso será descrito o processo de transformação usado para algumas atividades do método `EmitirNotaFiscal` representativas do trabalho realizado.

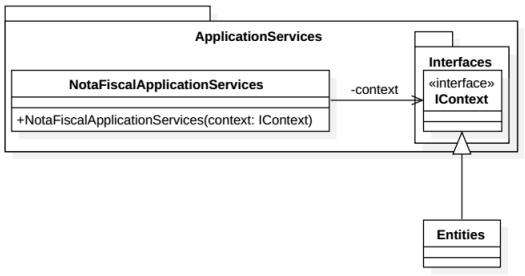
4.1 Eliminação do padrão *singleton* no acesso ao banco de dados

A classe `Entities` é uma implementação do padrão *unit of work* com métodos para o acesso a repositórios das entidades do modelo e métodos para a efetivação das alterações no banco de dados. Sendo assim, não há necessidade de alterar a maneira como

ela é utlizada. Porém, a obtenção das instâncias dessa classe no sistema é problemática por ser feita através da implementação do padrão *singleton*. Uma alternativa é o uso da injeção de dependências, cabendo à raiz da composição a responsabilidade de prover a instância adequada ao compor a árvore de dependências.

O uso do padrão *singleton* foi substituído pelo padrão de injeção de dependências. Para isso, foi extraída uma interface `IContext` contendo os métodos do padrão *unit of work*. Com isso, as classes do sistema que necessitarem do acesso aos repositórios devem receber uma instância de `IContext` através dos seus construtores, conforme exibido na Figura 4.2. Isso facilita a adoção de testes unitários, pois, no teste, pode ser fornecida uma implementação falsa de `IContext` com o objetivo de isolar as unidades em teste do banco de dados da aplicação.

Figura 4.2: Dependência com a classe `Entities` injetada através do construtor



Fonte: O Autor

4.2 Substituição da classe utilitária

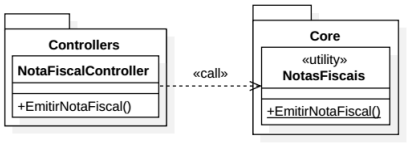
O uso de classes utilitárias dificulta a implementação de testes unitários, pois não é possível substituir na unidade em teste as chamadas aos métodos dessas classes por referências a uma implementação falsa que permita isolar o comportamento da unidade em teste. As seguintes estratégias foram adotadas para remover as dependências com classes utilitárias:

Introdução de uma camada de aplicação

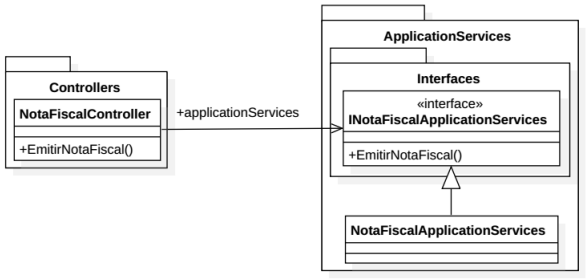
O método `EmitirNotaFiscal` foi movido para a classe de aplicação e tornado não-estático. A única referência a esse método acontece na *controller*, portanto esta transição de estático para não-estático pôde ser feita de forma simples, substituindo a referência estática pela instanciação de um objeto da nova classe. Como essa instanciação

também é uma dependência que se deseja remover, a variável local que mantém essa referência foi transformada em um campo da *controller* inicializado através do seu construtor. Por fim, foi criada uma interface para a nova classe para permitir que a *controller* dependa somente de uma abstração e não da sua implementação. Isso não altera a lógica do método que está sendo movido, somente altera a forma de referenciá-lo onde ele é chamado. Essa alteração está representada no diagrama de classes da Figura 4.3.

Figura 4.3: Extração do método `EmitirNotaFiscal` para uma classe própria



(a) Inicialmente a *controller* faz uma chamada para o método estático

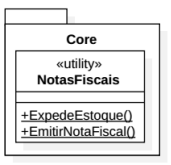


(b) Após a extração, a *controller* depende de uma interface cuja implementação é fornecida através do seu construtor

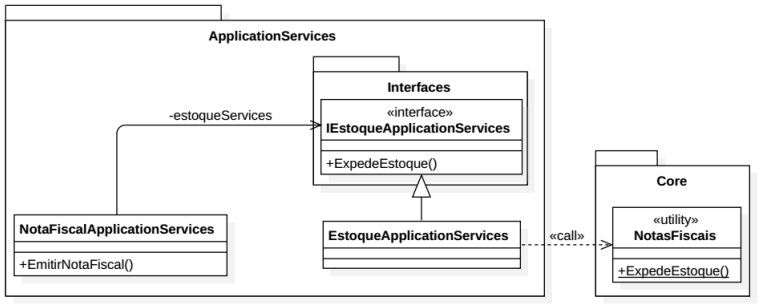
Fonte: O Autor

Essa mesma estratégia foi usada em outros casos semelhantes, nos quais existem dependências com métodos de classes utilitárias que implementam lógica relacionada com outros módulos do sistema. Como a abordagem adotada foi a de reengenharia parcial, nesses casos esses métodos não serão alterados. Para contornar esse obstáculo e obter o desacoplamento necessário, foi adotada uma estratégia semelhante à citada anteriormente nessa seção. Porém, apenas as chamadas aos métodos foram movidas para a classe introduzida, sem que o método original seja movido, conforme exemplificado na Figura 4.4.

Figura 4.4: Encapsulamento do método `ExpedeEstoque`



(a) Estado inicial



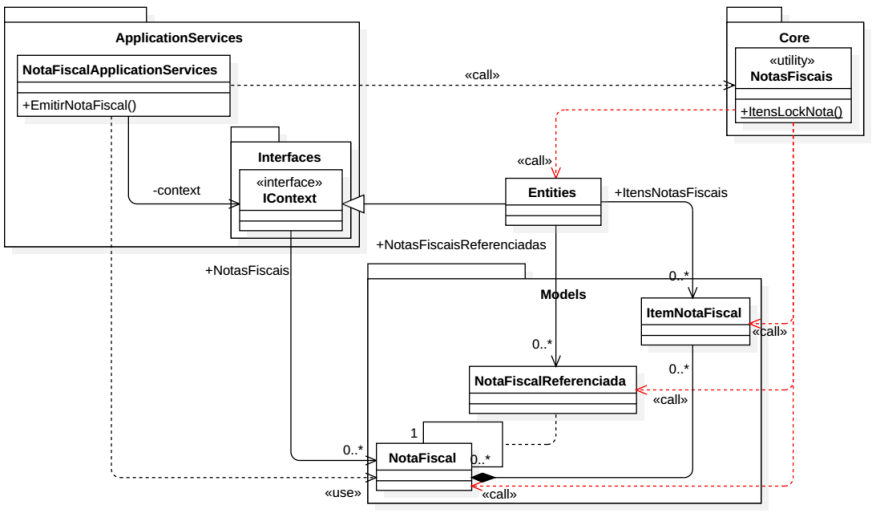
(b) Estado final

Fonte: O Autor

Introdução de comportamento nas classes do modelo

O método `ItensLockNota` da classe utilitária `NotasFiscais` é responsável por buscar no banco de dados todos os `ItemNotaFiscal` de uma dada nota fiscal e também de todas as notas fiscais associadas a ela através da classe de associação `NotaFiscalReferenciada`. As relações entre os módulos envolvidos na obtenção dos *locks* de acesso exclusivo ao banco de dados estão representadas no diagrama de classes da Figura 4.5. As associações entre as classes do *namespace* `Models` são construídas pelo *Entity Framework* e poderiam ser usadas para fazer essa consulta a partir da instância da nota fiscal, porém o método `ItensLockNota` faz consultas explícitas para cada um dos níveis dessa hierarquia, levando às dependências destacadas na figura.

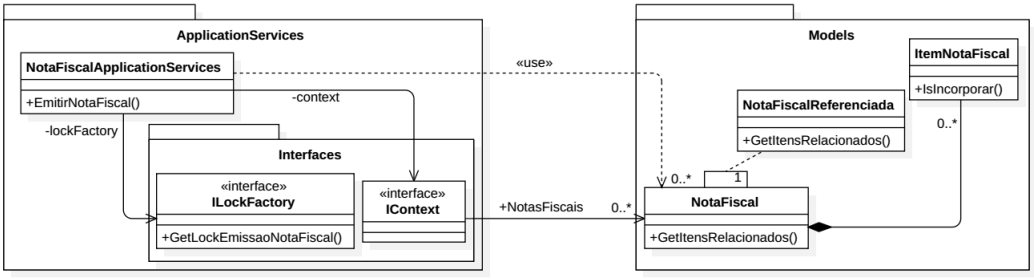
Figura 4.5: Método `ItensLockNota`



Fonte: O Autor

Para eliminar essas dependências desnecessárias, o método `ItensLockNota` foi movido para a classe `NotaFiscal`, que era usada como uma classe de dados e pode ser responsável por esse comportamento. Seguindo a análise do método, foi possível encontrar trechos do código que seriam mais adequados em outras classes do modelo. Estendendo a ideia de transformar as classes de dados em classes com comportamento, partes do método foram extraídas e movidas para as classes `ItemNotaFiscal` e `NotaFiscalReferenciada`, conforme a Figura 4.6

Figura 4.6: Resultado da reengenharia do método `ItensLockNota`



Fonte: O Autor

4.3 Exploração do polimorfismo na classe `NotaFiscal`

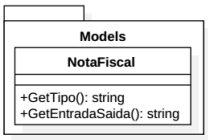
Grande parte da complexidade encontrada na análise se deve ao acúmulo de responsabilidades da classe `NotaFiscal` que representa tipos diferentes de documentos tratados pelo sistema, sendo eles: nota fiscal, nota fiscal recebida, pedido de compra, pedido de venda, cotação de compra e proposta de venda. A diferenciação entre esses tipos de documentos é feita pelo valor de dois campos da `NotaFiscal` que são mapeados diretamente para colunas do banco de dados, conforme a Figura 4.7a. Como cada um desses documentos necessita de tratamentos diferentes, os métodos na classe utilitária `NotasFiscais` não podem utilizar polimorfismo e sobrecarga para reduzir a complexidade das suas implementações, dependendo de condicionais explícitos baseados no valor desses campos.

Como métodos com complexidade ciclomática elevada necessitam de uma quantidade muito grande de casos de teste, foi introduzida a hierarquia da Figura 4.7b. Os métodos retrabalhados que dependiam de uma instância de uma `NotaFiscal` e tomavam decisões baseados no conteúdo dos campos que diferenciam o tipo de nota fiscal no banco de dados foram substituidos por sobrecargas que tiram vantagem do polimorfismo para reduzir a complexidade. A responsabilidade de retornar o tipo correto de acordo com o conteúdo do registro no banco de dados ficou a cargo do *Entity Framework* e não mais do desenvolvedor.

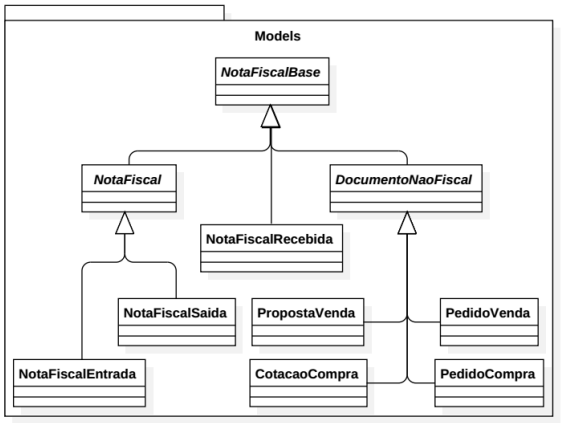
Como a classe `NotaFiscal` é gerada automaticamente pelo *Entity Framework*, a implementação dessa hierarquia de tipos deve ser feita através da própria ferramenta. Nas configurações do mapeamento entre as classes do modelo e as tabelas do banco de dados é possível estabelecer uma relação de herança entre entidades mapeadas para uma mesma tabela. Nesse modo, chamado de *table-per-hierarchy*, a ferramenta permite definir quais são as colunas que serão mapeadas para a classe pai e quais pertencem às classes

filhas e também a condição que determinará a qual classe da hierarquia um determinado registro será mapeado. Essa condição se dá na forma de um teste no valor de uma coluna, tal como é feito nos métodos da classe `NotasFiscais`, porém essa verificação é feita implicitamente pela ferramenta, que garante que uma consulta sempre retornará uma instância do tipo adequado, sendo possível se beneficiar do polimorfismo para reduzir a complexidade da lógica de aplicação.

Figura 4.7: Hierarquia de tipos de nota fiscal



(a) Antes: O tipo de documento deve ser consultado explicitamente



(b) Depois: A diferenciação entre os tipos de documento se dá por polimorfismo

Fonte: O Autor

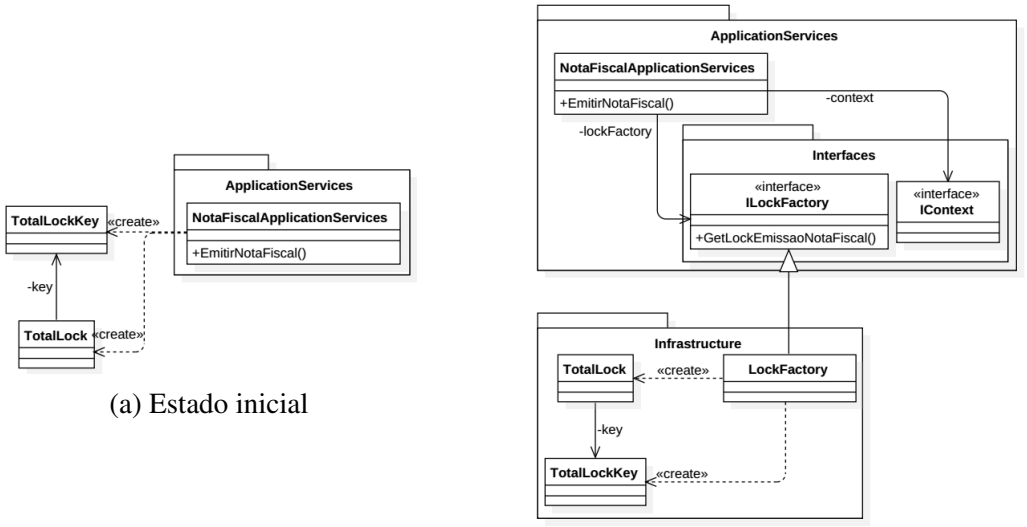
4.4 Adequação ao princípio de única responsabilidade

A lógica de criação dos objetos `TotalLock` e `TotalLockKey` era responsabilidade do método `EmitirNotasFiscais`. Para isolar essa responsabilidade, a lógica foi extraída para um método próprio a partir do qual foi criada uma classe que implementa o padrão *abstract factory* e pode ser fornecida por injeção de dependências. O diagrama de classes do estado final desse passo da renovação se encontra na Figura 4.8.

Outro caso de violação do princípio de única responsabilidade é a instanciação do objeto que representa a transação no banco de dados pelo método `EmitirNotaFiscal`. Nesse caso, como a classe que representa a transação depende unicamente do contexto do banco de dados gerado pelo *Entity Framework*, foi extraído um método que faz a sua instanciação e este foi movido para classe `Entities` como pode ser visto na Figura 4.9. Isso está de acordo com o padrão de *unity of work* uma vez que o objeto do contexto passa a ser responsável por gerenciar a transação dentro da unidade de trabalho.

O método `EmitirNotaFiscal` também era responsável por fazer a atualização

Figura 4.8: Padrão *abstract factory* na obtenção do *lock*

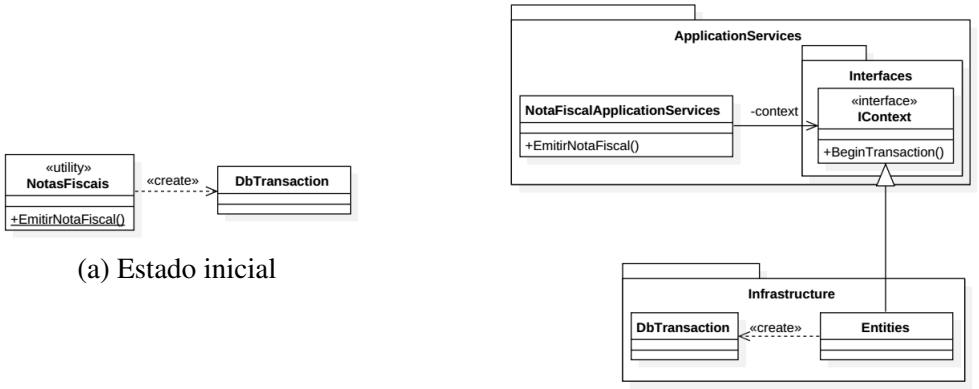


(a) Estado inicial

(b) Estado final

Fonte: O Autor

Figura 4.9: Encapsulamento da instanciação da classe `DbTransaction`



(a) Estado inicial

(b) Estado final

Fonte: O Autor

das duplicatas utilizadas no módulo financeiro. A lógica de manipulação dessas entidades, além de ser uma responsabilidade adicional do método, utiliza excessivamente dados das duplicatas, apesar de ser um método que trata de nota fiscal. Além disso, todos os acessos às duplicatas estavam concentrados no mesmo trecho de código evidenciando a necessidade de isolá-los como uma funcionalidade independente. Dessa forma, foi extraído um método que foi movido para uma nova classe na camada de aplicação. Essa lógica pertence a outro módulo e, portanto, não foi retrabalhada até o fim, de forma semelhante ao descrito na seção 4.2. Essas alterações resulta em métodos com responsabilidades melhor delimitadas, o que se reflete em casos de teste mais simples.

4.5 Resultados

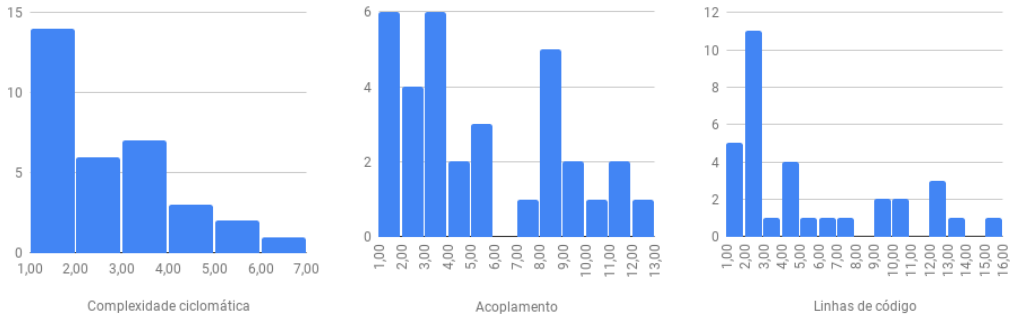
Após a aplicação das técnicas apresentadas nas seções anteriores em todo o método `EmitirNotaFiscal`, este foi desmembrado em um total de 33 métodos em 10 classes sendo 7 novas e 3 pré-existentes. A Tabela 4.1 e a Figura 4.10 apresentam um resumo das métricas obtidas para esses métodos que podem ser comparadas com a Tabela 3.1.

Tabela 4.1: Métricas de qualidade após a reengenharia

Métrica	Mínimo	Médio	Máximo
Complexidade ciclomática	1	2,27	6
Acoplamento	1	4,97	12
Linhas de código	1	5,03	15

Fonte: O Autor

Figura 4.10: Contagem de métodos por valor das métricas



(a) Complexidade (b) Acoplamento (c) Linhas de código

Fonte: O Autor

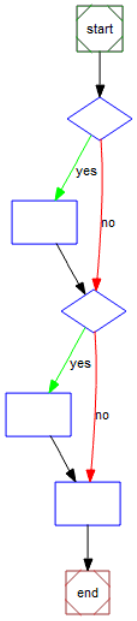
A grande quantidade de métodos que resultaram do processo de reengenharia demonstram o acúmulo de responsabilidades que existia no método `EmitirNotaFiscal`. Um indicativo de que os novos métodos do sistema seguem o princípio de única responsabilidade é a sua baixa contagem de linhas de código, que não passa de 15. Adicionalmente, nenhum método tem complexidade ciclomática maior que seis, o que indica o sucesso do processo de reengenharia, e isso se reflete em uma baixa quantidade de casos de teste para cada um desses métodos.

O acoplamento entre as classes do sistema também teve uma melhora significativa, pois os novos métodos do sistema não têm mais do que 12 dependências cada. Uma vez que a medida de complexidade obtida pelo *Visual Studio* considera como dependências

classes como `String`, esse número está superestimado. Durante a reengenharia, foi tomado o cuidado para que todos os métodos sigam o princípio inversão de dependências, portanto, essas dependências são abstratas e podem facilmente ser substituídas por *mocks* na execução dos testes unitários.

Essa diminuição nos valores das métricas pode ser visto pelo CFG da Figura 4.11 que representa o estado final do método `EmitirNotaFiscal`. A maior parte das decisões contidas no CFG original (Figura 3.6) eram responsabilidade de outras classes e foram realocadas. O restante apresentado na Figura 4.11 representa somente a lógica fundamental da atividade de emissão dos documentos tratados pelo sistema.

Figura 4.11: CFG do método `EmitirNotaFiscal` após a reengenharia



Fonte: O Autor

4.6 Testes unitários

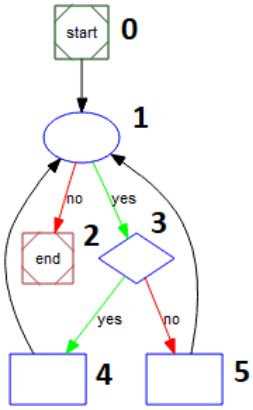
Para demonstar a adequação do novo projeto aos requisitos de testabilidade, testes unitários foram implementados para as diversas classes que passaram a compor o novo módulo de emissão de notas fiscais. A geração dos casos de teste foi feita pelo critério de cobertura de caminhos primos a partir dos grafos de fluxo de controle dos métodos após eles terem sido completamente implementados segundo o novo projeto.

Conforme demonstrado no seção 4.5, os novos métodos são pequenos e pouco complexos. Isso permite que o critério de caminhos primos seja atingido com uma quantidade baixa de casos de teste. O baixo acoplamento alcançado com a reengenharia também

permite que cada caso de teste seja simples de ser implementado. O critério de caminhos primos foi utilizado em todos os métodos porque garante uma boa cobertura para casos em que existem laços no CFG e não acarreta em nenhum prejuízo em casos em que isso não acontece. Essa decisão foi tomada para manter a homogeneidade da técnica utilizada e facilitar a reprodutibilidade desta pela equipe de desenvolvimento.

Para demonstrar o processo de geração dos casos de teste, a Figura 4.12 apresenta o CFG do método `AtualizaDuplicatas` que apresentou complexidade ciclomática igual a seis, a maior dentre todos os novos métodos do sistema. A aplicação do critério de caminhos primos a este grafo gerou os 13 requisitos de teste da tabela 4.2 que podem ser cobertos com os três casos de teste representados na Tabela 4.3. Essa técnica foi aplicada a todos os métodos, resultando em um total de 68 casos de teste para atingir a cobertura completa do módulo de emissão de notas fiscais. Esses casos de teste foram implementados em C# usando o *framework NUnit* com o auxilio da biblioteca *Moq* de geração de *mock objects* para servirem como entradas dos casos de teste.

Figura 4.12: Grafo de fluxo de controle do método `AtualizaDuplicatas`



Fonte: O Autor

Tabela 4.2: Requisitos de teste para o método `AtualizaDuplicatas`

#	Requisito
1	[1,3,5,1]
2	[3,4,1,2]
3	[3,4,1,3]
4	[1,3,4,1]
5	[0,1,3,4]
6	[0,1,3,5]
7	[3,5,1,2]
8	[5,1,3,4]
9	[5,1,3,5]
10	[4,1,3,5]
11	[3,5,1,3]
12	[4,1,3,4]
13	[0,1,2]

Fonte: O Autor

Tabela 4.3: Casos de teste gerados para o método `AtualizaDuplicatas`

Caso de teste	Requisitos cobertos
[0,1,3,4,1,3,5,1,3,5,1,3,4,1,3,4,1,2]	1, 2, 3, 4, 5, 8, 9, 10, 11, 12
[0,1,3,5,1,2]	1, 6, 7
[0,1,2]	13

Fonte: O Autor

4.7 Redocumentação e transição

Uma vez que a nova implementação passou nos testes de aceitação desenvolvidos no início do processo, o módulo retrabalhado foi integrado ao ciclo produtivo da empresa. Foi adicionado à ferramenta de integração contínua utilizada pela empresa um passo de execução dos testes unitários desenvolvidos. O novo projeto do módulo de emissão de notas fiscais e as estratégias de reimplementação foram expostos para a equipe e a extensão incremental da reengenharia para os demais módulos do sistema foi proposta aos administradores. Toda a documentação gerada durante este trabalho também foi disponibilizada para a equipe. Tendo sido integrada ao ciclo produtivo, a disponibilização da versão retrabalhada será disponibilizada para os usuários na próxima atualização programada do sistema.

5 CONCLUSÃO

Uma forma de reduzir custos de manutenção é a aplicação de técnicas de verificação e validação para garantir a qualidade do *software*. Embora existam outros métodos mais eficientes para deteccção de falhas no desenvolvimento de um *software*, os testes automatizados são uma forma barata e eficaz de identificar essas falhas em um estágio inicial do processo. Dentre os diferentes escopos de testes automatizados, os testes unitários são os que atuam sobre as menores unidades do código e, portanto, permitem uma identificação rápida e localizada das falhas. Para viabilizar a aplicação de testes unitários, o sistema precisa obedecer alguns critérios de testabilidade. Caso não tenha sido projetado tendo em vista essa prática, o sistema precisa ser reestruturado. Quando o tamanho do sistema impede que este seja reconstruído desde o início, a técnica de reengenharia é uma alternativa economicamente viável para adequação do sistema aos critérios de testabilidade.

Este trabalho aplicou técnicas de reengenharia em um *software* com seis anos de atividade para torná-lo adequado para adoção de testes unitários em seu processo de desenvolvimento. Foi encontrada uma grande quantidade de problemas que dificultam a implementação dos testes unitários, como métodos longos e que acumulam muitas responsabilidades, o uso de anti-padrões e a não adequação ao princípio de inversão de dependências. Esses problemas implicam em uma complexidade ciclomática elevada e geram uma quantidade impraticável de casos de teste. A partir dessa análise, foi elaborada uma proposta de reestruturação do projeto do sistema. Como solução, foram utilizadas técnicas de refatoração para dividir as responsabilidades dos métodos e reorganizá-los de acordo com o novo projeto, e assim reduzir a complexidade e o acomplamento no sistema.

Após a aplicação da técnica, foi possível fazer a cobertura de todo o módulo de emissão de notas fiscais usando o critério de cobertura de caminhos primos com um total de 68 casos de teste. Em sequência, é possível utilizar o conhecimento adquirido sobre o sistema para estender o escopo dos testes para abranger outros tipos de testes, como por exemplo, testes de integração e testes de sistema. Uma vez que a abordagem adotada foi a reengenharia parcial do sistema, um caminho natural é estender o procedimento para os outros módulos, facilitando a realização de manutenções futuras e melhorando a qualidade do produto. Futuramente, é recomendado a reavaliação do processo de desenvolvimento, coletando novamente dados de incidência de defeitos e de eficiência do processo para verificar o impacto da adoção de testes unitários na produtividade da equipe.

REFERÊNCIAS

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 1st. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381.

BEIZER, B. **Black-Box Testing: Techniques for Functional Testing of Software and Systems**. Wiley, 1995. ISBN 9780471120940. Available from Internet: <https://books.google.com.au/books?id=8__Jw88RGPcC>.

BOEHM, B. W. et al. **Characteristics of software quality**. Amsterdam: North-Holland, 1978. (TRW Softw. Technol.). Available from Internet: <http://cds.cern.ch/record/104765>.

BROWN, W. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. Wiley, 1998. (Wiley computer publishing). ISBN 9780471197133. Available from Internet: <https://books.google.com.br/books?id=Dp9QAAAAMAAJ>.

BYRNE, E. J. A conceptual foundation for software re-engineering. In: **Proceedings Conference on Software Maintenance 1992**. [S.l.: s.n.], 1992. p. 226–235.

BYRNE, E. J.; GUSTAFSON, D. A. A software re-engineering process model. In: **[1992] Proceedings. The Sixteenth Annual International Computer Software and Applications Conference**. [S.l.: s.n.], 1992. p. 25–30.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on software engineering**, IEEE, v. 20, n. 6, p. 476–493, 1994.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: a taxonomy. **IEEE Software**, v. 7, n. 1, p. 13–17, Jan 1990. ISSN 0740-7459.

COOK, M. L. Software metrics: An introduction and annotated bibliography. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 7, n. 2, p. 41–60, abr. 1982. ISSN 0163-5948. Available from Internet: <http://doi.acm.org/10.1145/1005937.1005946>.

FEATHERS, M. **Working effectively with legacy code**. [S.l.]: Prentice Hall Professional, 2004.

FOWLER, M. **Patterns of enterprise application architecture**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002.

FOWLER, M. **Refatoração**: Aperfeiçoando o projeto de código existente. [S.l.]: Bookman, 2004. ISBN 9788536303956.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-oriented Software**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

JONES, C.; BONSIGNOUR, O. **The Economics of Software Quality**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2011. ISBN 0132582201, 9780132582209.

MARTIN, R. C. **Design Principles and Design Patterns**. 2000. Available from Internet: <https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf>.

MCCABE, T. J. A complexity measure. **IEEE Transactions on software Engineering**, IEEE, n. 4, p. 308–320, 1976.

MILLS, E. E. **Software metrics**. [S.l.], 1988.

PEZZè, M.; YOUNG, M. **Software testing and analysis - process, principles and techniques.** [S.l.]: Wiley, 2007. ISBN 978-0-471-45593-6.

RAJLICH, V. T.; BENNETT, K. H. A staged model for the software life cycle. **Computer**, v. 33, n. 7, p. 66–71, Jul 2000. ISSN 0018-9162.

SOMMERVILLE, I. **Engenharia de software**. 9th. ed. [S.l.]: Pearson Brasil, 2011. ISBN 9788579361081.