UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM QUALIDADE DE SOFTWARE

Cristiano Chiele

# ESTUDO SOBRE PRÁTICAS ÁGEIS DE REFATORAÇÃO E TESTES AUTOMATIZADOS NO DESENVOLVIMENTO DE SOFTWARE PARA MELHORIA DA QUALIDADE DE SISTEMAS LEGADOS

São Leopoldo

2017

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM QUALIDADE DE SOFTWARE

Cristiano Chiele

# ESTUDO SOBRE PRÁTICAS ÁGEIS DE REFATORAÇÃO E TESTES AUTOMATIZADOS NO DESENVOLVIMENTO DE SOFTWARE PARA MELHORIA DA QUALIDADE DE SISTEMAS LEGADOS

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Especialista em Qualidade de Software, pelo curso de Pós-Graduação Lato Sensu em Qualidade de Software da Universidade do Vale do Rio dos Sinos – UNISINOS.
Orientador: Prof. Me Guilherme Lacerda

São Leopoldo

2017

# Estudo sobre práticas ágeis de refatoração e testes automatizados no desenvolvimento de software para melhoria da qualidade de sistemas legados

**Cristiano Chiele**[1]

[1]Universidade do Vale do Rio dos Sinos (Unisinos) – São Leopoldo – RS – Brasil

cchiele@gmail.com

***Abstract.*** *Legacy systems may have poor code quality, which makes its maintenance difficult, this problem is often called technical debt. In this context, this paper performs a study on agile refactoring practices and automated tests in the development of legacy software, in order to verify if, in fact these techniques can minimize the technical debt of the code. This paper is presented as a practical study in part of a real legacy system, applying refactoring techniques and automated testing. In the end, a comparative use of how it was before and how it was after the application of the techniques was carried out. The results showed that through refactoring techniques, it was possible to reduce the average number of lines of code per function by 40%, improving legibility and maintainability. With decreasing coupling and increased cohesion, it was possible to apply unit tests in 78% of the functions of a given file, proving that these techniques can indeed minimize the technical debt of the code in the medium and long term.*

***Resumo.*** *Sistemas legados podem apresentar baixa qualidade de código, tornando a sua manutenção difícil, problema esse muitas vezes chamado de dívida técnica. Nesse contexto, o presente trabalho realiza um estudo sobre práticas ágeis de refatoração e testes automatizados no desenvolvimento de software legado, visando verificar se, de fato estas técnicas podem minimizar a dívida técnica do código. O artigo engloba um estudo prático em parte de um sistema legado real, aplicando técnicas de refatoração e testes automatizados. Ao final do processo é realizado um comparativo de uso de como era antes e como ficou depois da aplicação das técnicas. Os resultados mostraram que através das técnicas de refatoração, foi possível reduzir em 40% a média de linhas de código por função, melhorando a legibilidade e manutenibilidade. Com a diminuição do acoplamento e aumento da coesão, foi possível aplicar testes unitários em 78% das funções de um determinado arquivo, comprovando que estas técnicas de fato podem minimizar a dívida técnica do código no médio e longo prazo.*

## 1. Introdução

Segundo Warren (2000) e Pressman (2011), sistemas legados geralmente são antigos e construídos com pouca ou nenhuma das boas prática de engenharia de *software*, mesmo assim são sistema críticos, que permanecem de forma vital para muitas organizações.

Um dos problemas mais críticos para os desenvolvedores deste tipo de sistema está relacionado a legibilidade e a manutenibilidade do código. Isso porque as mudanças feitas ao longo da vida do *software*, fazem com que sua estrutura interna se degrade, consequentemente, tornam-se mais difíceis de serem entendidos e alterados à medida que envelhecem [SOMMERVILLE 2011].

A cada ano as empresas de *software* desperdiçam mais e mais horas e recursos na manutenção de sistemas legados devido à má qualidade de seu código. De acordo com Kerievsky (2008), no momento em que um sistema legado apresenta má qualidade em seu código, dificultando a legibilidade e a manutenibilidade, esse acúmulo de código com má qualidade é muitas vezes conhecido como dívida técnica. Quanto mais tempo um sistema legado conviver com esta dívida técnica, mais cara e difícil será para removê-la [KERIEVSKY 2008].

Em sistemas assim, a manutenção se torna uma atividade extremamente difícil e arriscada, já que é grande a probabilidade de se incluir defeitos durante as manutenções, deixando os programadores com receio de alterar o código [FEATHERS 2013].

Desta forma, o investimento em código de qualidade é uma forma muito efetiva de reduzir os custos com manutenção, pois economiza horas do tempo de cada desenvolvedor para realizar correções, modificações ou para acrescentar alguma regra de negócio ao sistema [FOWLER 2004].

Para minimizar a dívida técnica, transformando o código de um sistema legado em algo de fácil entendimento e manutenção, é preciso investir tempo e esforço para melhorar a qualidade de seu código. Para isso, diversas técnicas podem ser utilizadas, como por exemplo a refatoração, onde através da mesma, é possível aplicar melhorias na estrutura interna do código já existente, sem alterar seu comportamento observável [FOWLER 2004].

Nesse contexto, o objetivo principal do presente trabalho é fazer uma contribuição na área de qualidade de código, realizando um estudo sobre práticas ágeis de refatoração e testes automatizados no desenvolvimento de *software* legado, visando verificar se, de fato estas técnicas podem minimizar a dívida técnica do código. Para atingir o objetivo principal, definiram-se os seguintes objetivos específicos:

1. Realizar pesquisas em trabalhos relacionados a refatoração e testes unitários em *softwares* legados;
2. Realizar um estudo prático de refatoração e testes em um sistema legado real;
3. Avaliar os resultados do estudo, e;
4. Construir heurísticas para refatoração do código legado.

O presente trabalho está organizado da seguinte forma: a seção 2 exibe uma visão geral sobre *software* legado, refatoração e testes de *software*. Na seção 3 apresentam-se os trabalhos relacionados. Em seguida, na seção 4 aborda-se a metodologia aplicada no presente trabalho. A seção 5 apresenta o estudo prático realizado e seus resultados. Por fim, a seção 6 é responsável pela conclusão.

## 2. Referencial Teórica

O referencial teórico utilizado para elaboração e execução da pesquisa realizada neste trabalho é apresentado nesta seção.

### 2.1. Sistemas de *Software* Legados

Segundo Warren (2000), sistemas legados geralmente são antigo, mas que ainda estão em operação nas organizações. Muitos destes sistemas são críticos para o negócio das organizações e caros de se manter. As razões para esse alto custo de manutenção incluem utilização de práticas imaturas de engenharia de *software* durante a sua construção, sacrifício da sua manutenibilidade para satisfazer outras restrições de projeto, tornando estes sistemas de difícil manutenção.

Pressman (2011) diz que "Sistemas de *software* legado foram desenvolvidos décadas atrás e tem sido continuamente modificados para se adequar às mudanças dos requisitos de negócio e a plataformas computacionais. A proliferação de tais sistemas está causando dores de cabeça para grandes organizações que os consideram dispendiosos de manter e arriscados de evoluir", [PRESSMAN 2011, p.8].

Pressman (2011) também cita que "muitos sistemas legados permanecem dando suporte para funções de negócio vitais e são 'indispensáveis' para o mesmo". Por isso, um sistema legado é caracterizado pela longevidade e criticidade de negócios [PRESSMAN 2011, p.8].

Pressman (2011) entende que sistemas legados às vezes têm projetos inextensíveis, código de difícil entendimento, documentação deficiente ou inexistente, casos de testes e resultados que nunca foram documentados e um histórico de alterações mal gerenciado. Ainda assim esses *softwares* são indispensáveis para o negócio de muitas organizações.

Para Sommerville (2011), algumas das razões que podem explicar a dificuldade de manutenção em sistemas legados são:

1) **Estabilidade da equipe**. Normalmente as equipes ao terminar um projeto, são realocadas para outros projetos, isso pode implicar em uma não padronização do código fontes, dificultando o entendimento da nova equipe que irá realizar uma manutenção.
2) **Má prática de desenvolvimento**. Normalmente sistemas antigos foram desenvolvidos sem técnicas modernas de engenharia de *software* e mal estruturados e talvez tenham sido otimizados para serem mais eficientes do que inteligíveis.
3) **Qualificações de pessoal**. Sistemas antigos podem ter sido escritos em linguagens obsoletas de programação. A equipe de manutenção pode não ter muita experiência de desenvolvimento nessas linguagens e precisa primeiro aprender para depois manter o sistema.
4) **Idade do sistema**. Com as alterações feitas no sistema, sua estrutura tende a degradar. Consequentemente, como os sistemas envelhecem, tornam-se mais difíceis de serem entendidos e alterados.

Apenas para exemplificar um destes sistemas de *software* legado que tem importância vital para a empresa, os bancos mantém seus sistemas de controle das transações bancárias em Cobol até os dias de hoje pois o risco de migração de um sistema destes é altíssimo, além do mais o risco de um sistema novo incluir um erro no sistema é alto e os prejuízos seriam absurdos. É devido a estas e outras razões que os bancos mantém seus controles num sistema legado.

## 2.2. Refatoração de *Software*

A constante manutenção de um *software* pode ocasionar alguns problemas, como deixar o código: desorganizado, mal codificado, com perda do desempenho, mais complexo, entre outros.

Segundo Sommerville (2011) o tempo e esforço gasto para a manutenção de um *software* pode ser maior do que foi gasto para o seu desenvolvimento.

Existem práticas ágeis que podem ser aplicadas no projeto de *software* que tem por objetivo diminuir o esforço na manutenção e melhorar a qualidade do mesmo. Uma dessas práticas é a refatoração de *software*.

Refatoração é uma alteração feita na estrutura interna do *software* para torná-lo mais fácil de ser entendido e menos custoso de ser modificado sem alterar seu comportamento externo [FOWLER 2004].

A refatoração se baseia nos princípios básicos de mudar o programa em passos pequenos, o bom código deve ser compreendido por pessoas e não só por computadores. A refatoração é uma ferramenta que ajuda a manter os códigos mais limpo e seguros. Ou seja, refatorar melhora o projeto de *software* [FOWLER 2004].

Segundo Kerievsky (2008), refatoração envolve a retirada de código duplicado, simplificação de lógica, e clarificação do código fonte.

Normalmente, durante o desenvolvimento, principalmente de sistemas legados, o código é feito para funcionar, e nem sempre é escrito dentro dos padrões e das melhores práticas. Com o passar do tempo, a alteração do código fará com que o projeto da aplicação sofra deterioramento. Com a refatoração, pode-se arrumar o código pois, a refatoração sistemática o ajuda a conservar a sua forma. Quando a refatoração do código é feita corretamente, o projeto fica melhor, mais legível e menos propenso a falhas. Em um código bem projetado e estruturado, o desenvolvimento é mais rápido e seguro. A refatoração pode ser feita em alguns momentos: quando se acrescenta novas funções, ao consertar falhas, ou ao revisar o código [FOWLER 2004].

Segundo Kerievsky (2008), para refatorar com segurança, é necessário testar as mudanças para verificar que o código não foi quebrado. Para ajudar neste processo, é de grande importância que os testes sejam automatizados, isso dará mais coragem a equipe para refatorar e estarão mais confiantes se puderem rodar testes automatizados rapidamente para confirmar que o código ainda funciona.

Kerievsky (2008) diz que a refatoração deve ser feita em pequenos passos para prevenir a introdução de defeitos no *software*.

## 2.3. Testes de *Software*

"O teste é destinado a mostrar que um programa faz o que é proposto a fazer e para descobrir os defeitos do programa antes do uso." [SOMMERVILLE 2011].

Segundo Pressman (2011), os teste de *software* são uma função de controle de qualidade com um objetivo principal de encontrar erros, e para isso é necessário planejá-los apropriadamente e conduzi-los eficientemente de modo que se tenha maior produtividade.

Segundo Fowler (2004 p.83) "um conjunto de testes é um detector poderoso de falhas que diminui o tempo que se leva para encontrá-las".

Os testes podem detectar a presença de erros, mas não que o *software* está livre deles. Mesmo que nenhum defeito tenha sido encontrado durante o teste, não significa que eles não existam. Portanto, o objetivo da equipe é projetar testes que revelem o maior número possível de defeitos com uma quantidade mínima de tempo e de esforço [SOMMERVILLE 2011].

A atividade de teste de *software* está se tornando cada vez mais importante, não apenas para detectar erros, mas também como parte fundamental para garantir a qualidade do produto final [MOLINARI 2003].

A figura 1 é um modelo abstrato do processo tradicional de testes apresentado por Sommerville (2011). Os casos de testes são especificações das entradas para o teste e da saída esperada do sistema. Os dados de teste são valores de entradas para testes efetivamente um sistema. Os resultados esperados são automaticamente comparados aos resultados descritos nos casos de testes [SOMMERVILLE 2011].

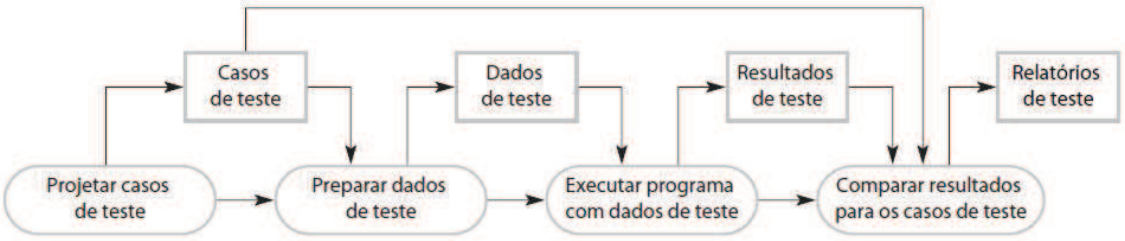



**Figura 1. Modelo de Teste de *Software***

**Fonte: Sommerville, 2011**

Atualmente várias abordagens sobre testes são encontradas, algumas delas são: testes de caixa branca, caixa preta, caixa cinza, regressão, técnicas não funcionais, unidade, TDD (*Test-Driven Development*), integração, sistema, aceitação, operação, alfa e beta e candidato a lançamento. Neste artigo serão abordados com maiores detalhes os testes de unidade e TDD.

### 2.3.1. Teste de Unidade

"O teste unitário é o processo de testar os componentes de programa, como métodos ou classes de objeto" [SOMMERVILLE 2011 p. 148].

Teste de unidade ou teste unitário, tem como alvo os menores elementos testáveis de um sistema, que podem ser funções, procedimentos, métodos ou classes. Nesta fase, o principal objetivo é encontrar defeitos de fluxo lógico, de dados e implementação das unidade, fornecendo evidências de que a unidade testada funciona corretamente de forma isolada [DELAMARO et al. 2007].

Segundo Feathers (2013), testar isoladamente é uma parte importante da definição de um teste de unidade, pois muitos erros podem ocorrer quando as partes de um *softwares* são integradas. Feathers (2013) diz também que, testes grandes abrangendo partes do sistema, também são considerados importantes, porém podem apresentar grandes problemas tais como:

- **Localização de erros** – Quando os testes são muito abrangentes, fica mais difícil examinar a falha e determinar em que local do trajeto entre a entrada e saídas ela ocorreu;
- **Tempo de execução** – Testes maiores normalmente podem demorar mais para serem executados, consequentemente acabam não sendo executados, e;
- **Cobertura** – Pode ser difícil identificar a conexão entre um trecho de código e os valores que o exercitam quando os testes são muito abrangentes.

Entretando, Feathers (2013) comenta que em sistemas legados desenvolvidos com linguagens procedurais, com frequência é difícil testar funções isoladamente. Funções de nível superior chamam outras funções, que chamam outras, até chegar ao nível de máquina. Desta maneira, pode ser necessário a construção de simuladores pois somente os funções superiores ao que serão testados estarão disponíveis, as inferiores não, fazendo com que os testes tornem-se demorados e trabalhosos [BARTIÉ 2002].

É verdade que testes de unidade custam tempo e aumentam os custos. Mas deve-se, considerar isto como um investimento e não como despesa, de forma que a cada novo conjunto de testes unitários rodar apresentando execuções bem sucedidas, aumentará a confiança da equipe e este começa a ser o retorno do investimento [MARINESCU 2002].

Para Fowler (2004) os testes devem ser orientados pelo risco, não é necessário testar todas as funções, métodos ou classes, quanto a sua implementação por muito simples, pois dificilmente algum erro será encontrado. É preciso concentrar os esforços onde há realmente riscos para o negócio se acontecer um erro.

"A chave é testar as áreas que mais teme que possam dar errado. Desta maneira, você obtém o maior benefício com seu trabalho de testes." [FOWLER 2004].

Para Feathers (2013), as qualidades encontradas quando são empregados bons testes de unidade são: testes sendo executados rapidamente e a ajuda na localização dos problemas mais facilmente.

### 2.3.2. *Test-Driven Development* (TDD)

Desenvolvimento dirigido a testes, ou TDD é uma abordagem de teste ágil, originária da metodologia XP (*Extreme Programming*). Nessa técnica o desenvolvimento é seguido por um teste. O desenvolvedor escreve o teste, executa o teste comprovando que o mesmo falhe, realiza a codificação e aplica o teste novamente [SOMMERVILLE 2011].

De acordo com Beck (2010), TDD é uma técnica na qual o próprio desenvolvedor escreve o teste antes mesmo de escrever o código. O TDD possibilita uma forma de refletir sobre a modelagem antes de implementar a funcionalidade, e também possibilita obter um código fonte com cobertura de teste.

TDD utiliza uma das técnicas do XP chamada *refactoring* para conseguir a compreensão do código e gerenciar a complexidade do mesmo. A medida que um sistema sofre modificações, ele torna-se mais complexo, sendo extremamente necessária a facilidade de manutenção. Esta técnica é essencial para reduzir a complexidade do *software* e torná-lo manutenível [FOWLER 2004].

Segundo Beck (2010), o primeiro passo no TDD é escrever os testes unitários, que irão validar se o código realmente funciona. Por segundo, o desenvolvedor executa os testes e, consequentemente os mesmos falham. Então, é desenvolvido os códigos necessários para que os teste passem. E por último, o código e os testes são refatorados, melhorados através das técnicas de refatoração. A Figura 2 ilustra os passos do TDD conforme descritos acima.

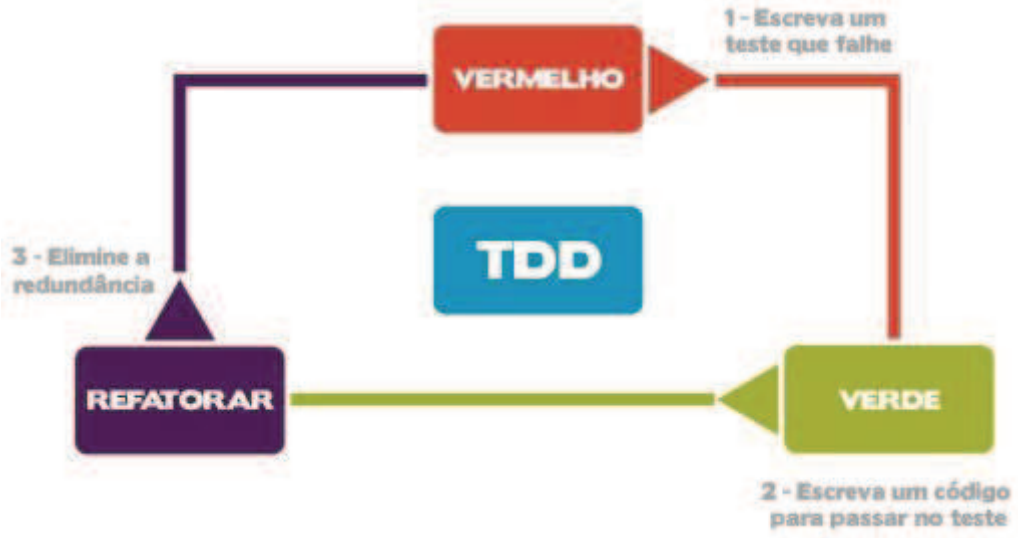



**Figura 2. Passos do TDD**

**Fonte: Devmedia, 2015**

Com TDD, os erros são identificados e corrigidos pelo próprio desenvolvedor. Por isso, esta técnica de teste é tão importante no desenvolvimento ágil [BECK 2010].

Segundo Aniche (2013), o TDD traz benefícios garantindo qualidade no código desenvolvido:

- **Foco no teste e não na implementação** – Ao começar pelo teste, o programador consegue pensar somente no que a classe deve fazer, e esquece por

um momento da implementação. Isso o ajuda a pensar em melhores cenários de teste para a classe sob desenvolvimento.

- **Código nasce testado** – Se o programador pratica o ciclo corretamente, isso então implica em que todo o código de produção escrito possui ao menos um teste de unidade verificando que ele funciona corretamente.
- **Simplicidade** – Ao buscar pelo código mais simples constantemente, o desenvolvedor acaba por fugir de soluções complexas, comuns em todos os sistemas. O praticante de TDD escreve código que apenas resolve os problemas que estão representados por um teste de unidade.

Para Aniche (2013), a grande dificuldade de se praticar TDD em sistemas legados é que geralmente eles apresentam uma grande quantidade de código procedural. Códigos procedurais geralmente contém muitas linhas de código, lidam com diferentes responsabilidades, acessam banco de dados, modificam a interface do usuário, o que os tornam difíceis de serem testados.

Nestes casos, Aniche (2013) sugere que o desenvolvedor, antes de começar a refatorar o legado, escreva testes automatizados da maneira que conseguir. Se conseguir escrever um teste de unidade, excelente. Mas caso não consiga, então ele deve começar a subir o nível até conseguir escrever um teste.

Com estes testes escritos, o desenvolvedor conseguirá refatorar o código legado problemático que está por baixo, e ao mesmo tempo garantir a qualidade do que está fazendo. Nessa refatoração, o desenvolvedor deve criar classes/funções menores e mais coesas e já escrever testes de unidade para elas [ANICHE 2013].

### 2.3.3. Estratégias para Manuseio de Código Legado

Segundo Feathers (2013) a maior parte do medo envolvido em fazer mudanças em grandes bases de código legado é o medo de introdução de *bugs* sutis, o medo de mudar as coisas inadvertidamente. Porém, grande parte de sistemas de informação partem de códigos legados, na grande maioria das vezes um sistema não é escrito do zero.

Por isso, é importante o desenvolvimento de estratégias para lidar com as mudanças. Feathers (2013) sugere uma estratégia composta de 5 passos:

1. Identifique os pontos de alteração.
2. Encontre os pontos de teste.
3. Elimine dependências.
4. Escreva testes.
5. Faça alterações e refatore.

O objetivo que desenvolvedores devem perseguir ao trabalhar com código legado é fazer alterações, mas não qualquer alteração. Devem ser feitas alterações funcionais que agreguem valor, trazendo ao mesmo tempo uma parte maior do sistema para a cobertura dos testes. No fim de cada etapa de programação, temos de poder nos dedicar não só a código que forneça alguma nova funcionalidade, mas também a seus testes [FEATHERS 2013].

### 2.3.4. Automação de Testes de *Software*

Para Graham (2012), a automação de testes está diretamente associada a boa qualidade de *software*, agregando valor ao produto final, mesmo que este processo de automação não seja perceptível ao usuário final do *software*. Mas, para que isso se torne verdade, o código fonte dos testes automatizados requer qualidade para que a evolução e manutenção sejam viáveis.

A automação de testes vem se tornando uma atividade de grande importância na qualidade dos projetos de *software* [CAETANO 2008]. Segundo Delamaro et al. (2007), a técnica de automação de testes é voltada principalmente para melhoria da qualidade, tornando os testes manuais em testes automatizados.

Entretanto, segundo Caetano (2008), a implantação da automação de testes depende de testes manuais maduros e consistentes. Os projetos de automação de testes bem sucedidos são aqueles que se baseiam em processos de testes formais e estruturados.

A automação de testes é desenvolvida com o auxílio de programas ou *scripts*, que têm como objetivo exercitar o sistema, testando as funcionalidades e verificando se estão de acordo com as especificações dos requisitos do sistema e os objetivos esperados [BERNARDO 2008].

Segundo Caetano (2008), em geral, os tipos de automação de testes são agrupados em dois paradigmas principais (mas não são os únicos): baseado na interface gráfica e baseado na lógica de negócio.

Na abordagem baseada na interface gráfica, os testes automatizados interagem diretamente com a interface gráfica da aplicação simulando as ações do usuário. Normalmente estas ações são gravadas por meio da ferramenta de testes automatizados. A ferramenta transforma as ações dos usuários em um *script* que pode ser reproduzido posteriormente [CAETANO 2008].

Já na abordagem baseada na lógica de negócio, a interface gráfica é apenas uma camada que fornece um meio para a entrada dos dados e apresentação dos resultados. A camada que abriga a funcionalidade e o comportamento da aplicação é a camada de lógica de negócio. Os testes automatizados exercitam as funcionalidades desta camada, sem interagir com a interface gráfica. Normalmente é necessário realizar modificações na aplicação para torná-la mais fácil de testar (testabilidade). Esta abordagem de testes é baseada no entendimento que 80% das falhas estão associados a erros na lógica de negócio [CAETANO 2008].

## 3. Trabalhos Relacionados

Nesta seção são apresentados alguns trabalhos relacionados com os assuntos de refatoração e testes em sistemas legados.

Heredia (2015), em seu trabalho de conclusão de curso, aplicou dez técnicas de refatoração em um módulo de um sistema legado de vendas de passagens escrito em Java, mostrando como a refatoração interferiu positivamente na qualidade do *software* e o retorno de investimento que pode trazer a projetos com problemas.

Para acompanhar a melhoria na qualidade do código com a refatoração, Heredia (2015), coletou métricas antes de aplicar as refatorações e ao final das refatorações, fazendo assim um comparativo.

Para realizar estas coletas de dados, Heredia (2015) utilizou duas ferramentas chamadas *SonarQube* e *Metrics*. Estas ferramentas avaliam a qualidade do código fonte, controlando inúmeras métricas e apontando possíveis *bugs*.

Ainda no trabalho de Heredia (2015), além da melhoria da qualidade de *software* após as refatorações, foi demonstrado como a inteligibilidade do código se tornou mais compreensível, e diminuindo o esforço para futuras manutenções, aumentando a vida útil do sistema.

Barrozo et al. (2012) realizaram um estudo sobre as várias técnicas de refatoração existentes, suas vantagens e desvantagens. Para isso, primeiramente desenvolveram um sistema de controle de monografias relativamente simples totalmente de forma procedural, mesmo a linguagem sendo orientada a objetos. Este sistema também continha vários "maus cheiros", mas principalmente a duplicação de código.

Segundo Barrozo et al. (2012), a primeira e mais ousada refatoração foi a conversão do sistema procedural para orientado a objetos, deixando o sistema com uma estrutura de classes capaz de unificar a maioria dos problemas de código duplicado.

Segundo Barrozo et al. (2012), os passos para execução desta refatoração foram a análise de cada tabela do banco de dados, verificando quais poderiam ser transformadas em classes e analisando o código fonte e movendo as devidas funções para uma classe de acordo com sua funcionalidade.

As várias refatorações aplicadas ao sistema, possibilitaram uma análise efetiva na melhoria da qualidade do *software*, além de uma redução considerável no número de linhas de código [ARROZO et al. 2012].

Santos (2011) em sua monografia, focou na aplicação de testes unitários utilizando a técnica do TDD juntamente com *framework* JUnit para melhorar a qualidade de um *software Web* de envio de informações agrárias para o Ministério da Agricultura.

Santos (2011) utilizou em seu trabalho as seguintes ferramentas: *NetBeans* IDE e *Framework* JUnit. Santos (2011) aproveitou um novo requisito do sistema para aplicar os conhecimentos. Este novo requisito implicava na criação de duas novas classes para resolver um determinado problema.

Santos (2011) iniciou utilizando a técnica do TDD, criando as classes de testes antes das classes reais, desta forma, as novas funcionalidades do sistema já nasceram testadas, aumentando consideravelmente a qualidade do sistema e garantindo uma maior abrangência da cobertura das falhas que poderiam ocorrer se estes testes não tivessem sido aplicados.

Ainda segundo Santos (2011), com a aplicação dos testes unitários, as falhas puderam ser corrigidas quase que instantaneamente pelo desenvolvedor, e ainda, como

foi utilizado o TDD, pensava-se já na solução para algumas falhas antes destas serem encontradas.

Martins et al. (2016) em seu estudo de caso, realizam a automação de testes em um sistema legado desenvolvido sobre plataforma *Flex*, onde o primeiro passo foi a escolha de uma ferramenta de automação que atendesse os requisitos mínimos levantados, tendo em mente as limitações de automação de testes em *Flex*.

Tendo feito a escolha da ferramenta, foram criados casos de testes, implementação de *scripts* de testes com a ferramenta e então executaram os testes de forma manual e automatizada, em um ambiente controlado e medindo o tempo de cada uma das etapas [MARTINS et al. 2016].

Os dados coletados de esforço e tempo durante a elaboração e realização dos testes levantam indícios que o ganho na automação reduz o erro humano, mas o ganho de tempo é pequeno inicialmente, principalmente devido ao tempo de elaboração dos *scripts* de teste. A vantagem está na rápida execução dos testes e nas execuções futuras quando o sistema receber novas funcionalidades ou manutenções [MARTINS et al. 2016].

Chaves (2011) apresentou em seu trabalho, um estudo de caso onde aplica testes unitários em um sistema de loja virtual que vende álbuns de músicas on-line. Os testes foram aplicados na principal funcionalidade sistema, que é carrinho de compra.

Chaves (2011) aplicou os conceitos teste unitário na prática, criando métodos de testes para cada um dos métodos das classes envolvidas na funcionalidade de carrinho de compra, e realizando execuções manuais para os testes.

Em seguida com o auxílio de uma ferramenta de automação chamada *Icarus Gallio*, Chaves (2011) automatizou os testes criados, facilitando a execução e visualização dos mesmos.

Segundo Chaves (2011), é notável a facilidade e a praticidade com os testes unitários são executados, resultando em um feedback instantâneo para o desenvolvedor.

Após comparar a proposta do presente trabalho com os trabalhos relacionados, pode-se observar que há uma semelhança com relação à aplicação das técnicas de refatoração e de testes unitários para ajudar na melhoria da qualidade do *software*. Todos os trabalhos relacionados tiveram como base a aplicação das técnicas na prática, e posteriormente avaliando os resultados. Na pesquisa desenvolvida neste trabalho, também pretendesse aplicar algumas das técnicas de refatoração e testes seguindo as boas práticas citadas na literatura, porém em um sistema legado de mais de 20 anos, desenvolvido com linguagem de programação procedural e com poucas ferramenta de apoio.

## 4. Metodologia

Nesta seção, aborda-se a metodologia adotada na realização do presente trabalho.

### 4.1. Delineamento da Pesquisa

Este trabalho consiste em uma pesquisa com enfoque qualitativa de natureza aplicada, pois segundo Kaplan (1988), as principais características dos métodos qualitativos são a

imersão do pesquisador no contexto e a perspectiva interpretativa de condução da pesquisa. Métodos qualitativos se ocupam de variáveis que não podem ser medidas, apenas observadas [WAINER 2007].

Segundo Gil (2010), a pesquisa aplicada engloba análises que propõem a resolução dos problemas descobertos no ambiente do pesquisador. São pesquisas voltadas a aquisição de conhecimentos com aplicação numa situação específica. Diante disso, do ponto de vista do objetivo da pesquisa pode-se dizer que a mesma é exploratória. Segundo Azevedo (2011), a pesquisa exploratória é uma pesquisa que permite a obtenção de novos conhecimentos, bem como ampliação e complementação acerca do tema abordado.

A metodologia de pesquisa escolhida será a pesquisa-ação. Para Engel (2000), a pesquisa-ação possibilita para um participante do processo, que apresenta o desejo de melhorar uma situação, a realização da pesquisa, possibilitando assim, a execução de modificações necessárias durante a execução do estudo.

Segundo Gil (2010), a pesquisa-ação é preferencial quando o pesquisador visa diagnosticar um problema específico em uma situação específica e está envolvido no contexto de modo cooperativo e participativo, buscando alcançar algum resultado prático, sendo ele não generalizável.

### 4.2. Unidade de Análise

A empresa apresentada como exemplo, que, por confidencialidade chamaremos ABC, foi fundada em 1911, e é uma empresa metalúrgica 100% brasileira sediada na Serra Gaúcha do Rio Grande do Sul. Atualmente a empresa ABC é composto por 10 fábricas espalhadas pelo território nacional, onde emprega 7.000 pessoas, produz mais de 18 mil itens diferentes para o lar, que vão de utensílios domésticos a jardinagem.

A empresa ABC possui um setor de TI em sua sede, que é responsável pelo desenvolvimento de *software*, onde todo o ERP (*Enterprise Resource Planning*) da empresa é desenvolvido. Este setor é composto por uma equipe 24 profissionais, onde a mesma se divide em 1 gerente de TI, 1 gerente de desenvolvimento e 22 desenvolvedores.

Como unidade de análise de pesquisa para o presente estudo foi elencada dentre a equipe de desenvolvimento de *software* de 24 pessoas, uma equipe de 3 pessoas, a qual é composta por 2 desenvolvedores sênior, sendo que um é o próprio autor deste trabalho e um gerente de desenvolvimento, os quais detém conhecimento suficiente para avaliar o estudo durante o seu desenvolvimento.

### 4.3. Técnicas de Coleta de Dados

A principal técnica utilizada foi a observação participante, que é uma modalidade de observação na qual o pesquisador não é apenas um observador passivo. Em vez disso, o pesquisador pode assumir uma variedade de funções dentro de uma pesquisa e pode, de fato, participar dos eventos que estão sendo estudados [YIN 2010].

A escolha desta técnica se dá principalmente as limitações da linguagem de desenvolvimento utilizada, pois não é compatível com nenhuma ferramenta de métricas de *software* disponíveis no mercado.

Para auxiliar no estudo, utilizou-se a pesquisa bibliográfica, que de acordo com Gil (2010), é elaborada com base em material já publicado, proporcionando aos pesquisadores o contato com os documentos que tratam do tema que está em estudo, contribuindo para o embasamento teórico-metodológico utilizado na construção do estudo.

### 4.4. Técnicas de Análise de Dados

A principal técnica utilizada será a análise de conteúdo, que segundo Moraes (1999), de certa forma, a análise de conteúdo é uma interpretação pessoal por parte do pesquisador vinculado ao entendimento que tem dos dados. Uma leitura neutra não é realizável. Toda leitura atribui-se a uma interpretação.

Também será utilizada técnicas de comparações qualitativas, realizando comparações do "antes e depois" da refatoração e implementação dos testes.

### 4.5. Limitações do Estudo

Este estudo será limitado a práticas que o desenvolvedor pode utilizar no seu dia-dia para melhorar a qualidade do código que produz, deste modo, os testes de *software* se restringem a testes unitários, que é a menor parte dos testes de *software*.

Devido ao pouco tempo para realização deste trabalho, as técnicas de refatoração e testes serão aplicados a uma pequena parte de um determinado módulo do sistema.

## 5. Estudo

Nesta seção será apresentado o desenvolvimento do presente trabalho.

### 5.1. Contextualização da Unidade de Análise

Conforme já comentado no subseção 4.2, a empresa ABC possui um setor de TI que é responsável pelo desenvolvimento de todo o sistema de gestão ERP da empresa.

O ciclo de vida adotado pela equipe de desenvolvimento é o modelo cascata adaptado a realidade da empresa e também o modelo incremental. A escolha entre um modelo e outro depende da complexibilidade do *software* a ser desenvolvido. No momento, a equipe de desenvolvimento não adota nenhuma metodologia ágil de desenvolvimento, e nenhum modelo de maturidade como CMMI ou MPS-BR.

As demandas de manutenções e de novas funcionalidades, chegam a equipe de TI através dos GTs (Grupos de Trabalhos). Os GTs são grupos de usuários chaves de cada sistema que se reúnem periodicamente com data e integrantes predefinidos para avaliar e discutir novas funcionalidades para o sistema. Normalmente o gerente de desenvolvimento e alguns desenvolvedores também participam destas reuniões para auxiliar nas dúvidas dos usuários chaves.

A figura 3 demonstra o processo macro de desenvolvimento de *software* adotado pela equipe de desenvolvimento.

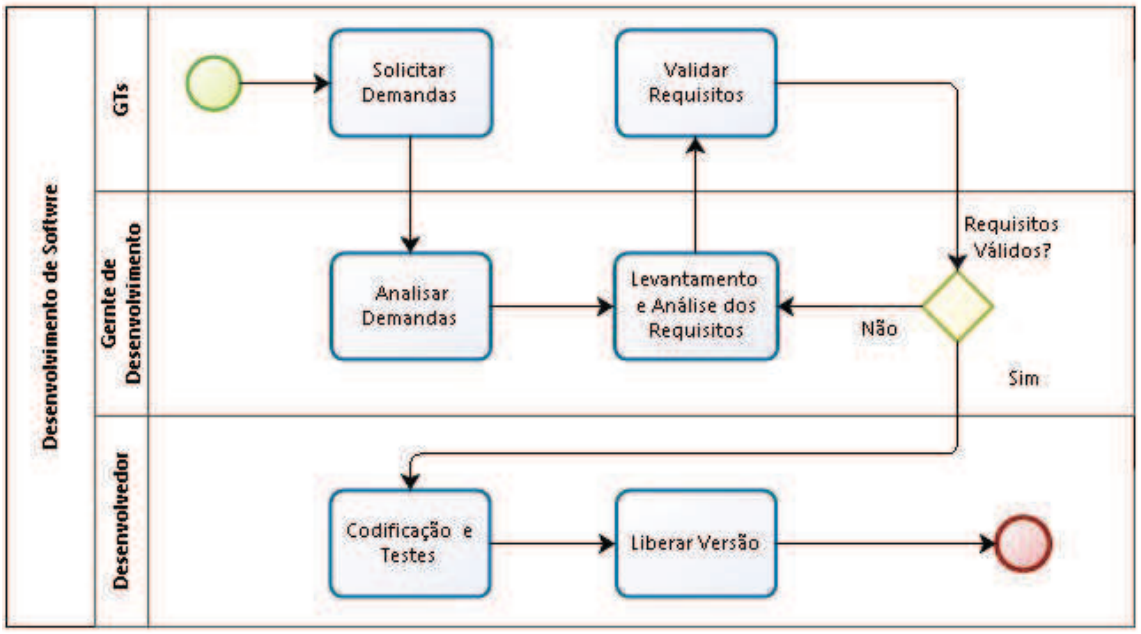



**Figura 3. Processo macro de desenvolvimento de *software***

**Fonte: Elaborado pelo autor**

Um ponto importante de se observar, é que não existe o papel do testador, ou seja, o próprio desenvolvedor é responsável por realizar todos os testes para garantir a qualidade do *software* que desenvolve.

A linguagem de programação utilizada pela empresa é chamada de *Genero*, que é uma linguagem pouco conhecida e utilizada no Brasil, apenas 3 grandes empresas a utilizam no país.

*Genero* é a evolução da antiga linguagem de programação *Informix 4GL*[1]. A linguagem *Genero* tem como principais caracteristicas:

- Linguagem totalmente procedural e não orientada a objetos;
- Arquitetura server ao invés de cliente/server;
- Linguagem interpretada e não executada, ou seja, necessita de um *runtime* para executar as aplicações;
- Multiplataforma, onde o lado server (*runtime*) pode rodar em sistema operacionais *Linux* e *Windows* e o lado cliente pode ser executado em ambientes *desktop* (*Windows*, *Linux* e *MAC*), *web* (*Chrome*, *Firefox* e *Internet Explorer*) e *mobile* (*iOS* e *Android*);
- Curva de aprendizagem muito baixa, pois sua sintaxe é muito simples, facilitando a compreensão.

[1] A linguagem de programação *Informix 4GL* pertencia a empresa *Informix Software*, que em 2001 foi adquirida pela IBM. Por volta de 2003, a empresa francesa chamada *Four J's Development Tools* adquiriu os direitos da ferramenta *Informix 4GL*, modernizando-a e renomeando para *Genero*.

O ERP desenvolvido pela empresa é vital para o negócio da organização, pois oferece todo o suporte necessário para as diversas atividades da organização. O quadro 1, apresenta algumas informações macro acerca do sistema.

**Quadro 1. Informações macro acerca do sistema**

| | |
|---|---|
| Número de usuário do sistema | 2.205 |
| Número de módulos | 20 |
| Número de programas | 2.900 |
| Linhas de código | 5.700.000 |

**Fonte: Elaborado pelo autor**

O ERP desenvolvido pela empresa, é um sistema legado antigo que vem sendo continuamente modificado a mais de 20 anos. Naquela época, em sua construção inicial, não existiam, ou não tinha conhecimento das boas práticas de engenharia de *software*, consequentemente, a medida que o sistema envelhece, torna-se mais difícil a compreensão e manutenção do sistema.

Atualmente, pouca documentação é gerada acerca do sistema, ou seja, muito do conhecimento das regras de negócio e particularidades envolvidas acabam ficando apenas com o desenvolvedor. Isso acaba gerando problemas de desconhecimento das razões que levaram a determinadas decisões no código.

A equipe de desenvolvimento vem percebendo que, à medida que o sistema vai sendo modificando, o código está ficando cada vez mais desorganizado, mal codificado, com maus cheiros, mais complexo e com mais erros. E isso se reflete diretamente na manutenibilidade do sistema, que está ficando cada vez mais difícil.

Diante disso, o principal objetivo do presente trabalho é realizar um estudo prático para verificar se, de fato as técnicas de refatoração e testes unitários automatizados podem contribuir na melhoria da qualidade deste sistema, estendendo a sua vida útil..

## 5.2. Identificando Pontos de Mudança

O primeiro passo na realização deste estudo foi descobrir em que ponto do sistema seriam aplicadas as refatorações e os testes, levando em consideração o curto prazo para a finalização do presente trabalho.

Atualmente, a linguagem de programação *Genero* não conta com ferramentas para extração de métricas de código fonte, e também as ferramentas disponíveis no mercado, como por exemplo *SonarQube* e *Metrics*, não são compatíveis com a linguagem *Genero*.

Diante dessa falta de métricas acerca do código, foi realizada análise dos arquivos de código fonte dos módulos do sistema, procurando os maiores arquivos em tamanho, pois arquivos muito grandes podem apresentar problemas, como por exemplo: funções muito grandes, alto acoplamento, baixa coesão, maus cheiros, etc.

Esta primeira análise resultou no levantamento dos 10 maiores (em KB) arquivos de código fontes do sistema, conforme quadro 2.

**Quadro 2. Análise dos 10 maiores arquivos do sistema**

| Módulo | Sigla do Módulo | Arquivo do Módulo | KB | LOC (*Line of Code*) | Qtde. de Funções | Média de LOC/ Função |
|---|---|---|---|---|---|---|
| PCP (Planejamento e Controle da Produção) | PC | pc_atu60.4gl | 992 | 22.000 | 73 | 300 |
| Departamento de Informática | DI | di_bid_transportadora.4gl | 422 | 11.000 | 113 | 97 |
| Compras | CM | cm_livro.4gl | 615 | 12.000 | 46 | 260 |
| Centro de Distribuição | CD | cd_ender.4gl | 391 | 9.000 | 22 | 409 |
| Faturamento | FT | ft_comprm.4gl | 304 | 7.000 | 35 | 200 |
| Faturamento | FT | ft_ender.4gl | 239 | 5.700 | 30 | 190 |
| Contabilidade | CG | cg_atulanc.4gl | 222 | 5.600 | 29 | 193 |
| Compras | CM | cm_box.4gl | 200 | 5.300 | 42 | 126 |
| Departamento de Informática | DI | di_inicio_prog.4gl | 151 | 4.300 | 41 | 104 |
| Compras | CM | cm_ras.4gl | 119 | 3.000 | 11 | 272 |

**Fonte: Elaborado pelo autor**

Com base no quadro 2, foi realizada uma análise para verificar a maior função em número de linhas de cada arquivo de código, conforme pode ser observado no quadro 3.

**Quadro 3. Análise das 10 maiores funções em número de linhas**

| Sigla do Módulo | Arquivo do Módulo | Função do arquivo com maior LOC | LOC |
|---|---|---|---|
| CM | cm_livro.4gl | gerar_livros | 1.500 |
| FT | ft_comprm.4gl | atualiza_localizacao_compr | 1.380 |
| FT | ft_ender.4gl | atu_ender_entrada | 1.360 |
| CD | cd_ender.4gl | atu_atacado_entr | 1.300 |

| | | | |
|---|---|---|---|
| DI | di_inicio_prog.4gl | inicio_prog | 930 |
| PC | pc_atu60.4gl | seleciona_atualizacao | 870 |
| CG | cg_atulanc.4gl | atualiza_contabilidade_geral | 830 |
| DI | di_bid_transportadora.4gl | grava_condicao_veneza_fabricas | 523 |
| CM | cm_box.4gl | controla_alocacao1 | 470 |
| CM | cm_ras.4gl | atu_rastreabilidade | 415 |

**Fonte: Elaborado pelo autor**

Para auxiliar na identificação dos arquivos do módulo que foram refatorados, a IDE (*Integrated Development Environment*) de desenvolvimento chamada de *Genero Studio*, disponibiliza dois tipos de diagramas, o diagrama de dependência (também chamado de diagrama de componente) e o diagrama de sequência.

Para Furlan (1998), um diagrama de componentes mostra as dependências entre componentes de *software*, inclusive componentes de código fonte (.h em C++ ou .java em Java), componentes de código binário (.dll em Visual Basic ou .class em java) e componentes executáveis.

Segundo Furlan (1998), um diagrama de seqüência mostra a interação entre os objetos, exibindo cada participante com uma linha de vida, que corre verticalmente na página, e a ordem das mensagens, lendo a página de cima para baixo.

Com o auxílio da IDE, foram gerados diagramas de dependência para cada um dos módulos apresentados no quadro 3. As figuras 4 e 5 são exemplos de dois destes módulos, módulo de CD (Centro de Distribuição), e módulo de FT (Faturamento).

**Figura 4. Diagrama de dependência do Módulo CD**

**Fonte: Elaborado pelo autor**

**Figura 5. Diagrama de dependência do Módulo FT**

**Fonte: Elaborado pelo autor**

É possível observar através dos diagramas das figuras 4 e 5, que estes módulos a princípio estão organizados e com pouca dependência entre seus arquivos, apesar de terem funções com muitas linhas de código.

Entretanto, a figura 6 mostra o diagrama de dependência do Módulo DI (Departamento de Informática). Este módulo é basicamente um *framework* com funções comuns que são utilizadas por todos os outros módulos e também pelos programas.

É possível observar no diagrama da figura 6, que há vários problemas com este módulo, como por exemplo: forte dependência dos arquivos entre si mesmo, dando indícios de desorganização, os nomes de vários arquivos do módulo não revelam o seu real agrupamento de funções e também é possível notar a existência de muita dependência circular sobre o arquivo `di_inicio_prog.4gl`.

Além disso, podemos ver no quadro 3, que duas das maiores funções listadas estão dentro do Módulo DI. Isso mostra que o Módulo DI é um excelente artefato para o estudo prático de refatoração e testes.

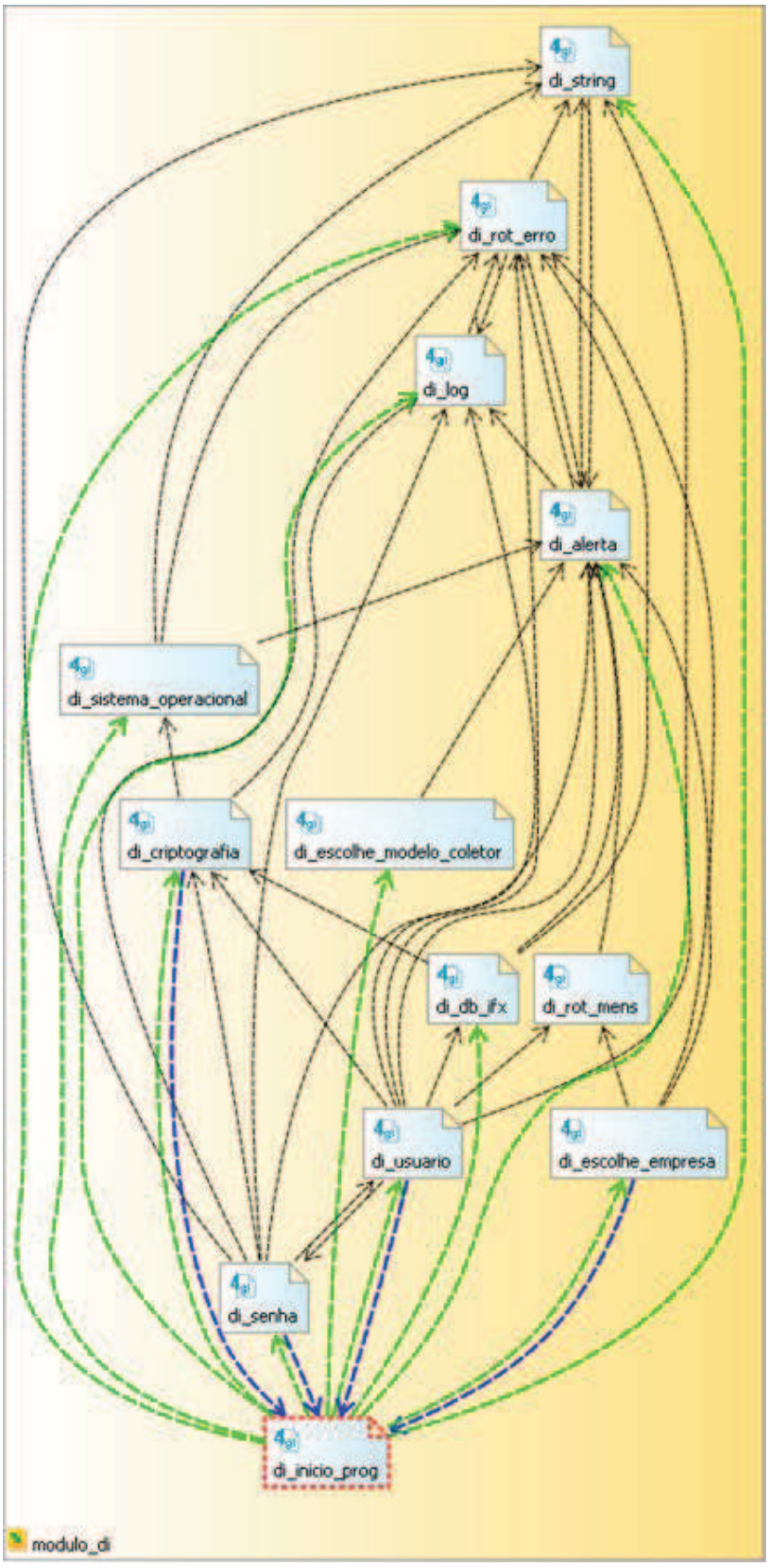



**Figura 6. Diagrama de dependência do Módulo DI**

**Fonte: Elaborado pelo autor**

Tendo em vista o curto prazo para a finalização deste trabalho, as refatorações e testes unitários foram aplicados apenas ao arquivo `di_inicio_prog.4gl` pertencente ao Módulo DI. Este arquivo não é necessariamente o maior arquivo, mas contém funções de extrema importância para o funcionamento do sistema.

O arquivo `di_inicio_prog.4gl` contém a função chamada de `inicio_prog` que é sua principal função, e é a partir dela que as refatorações tiveram início. O objetivo atual desta função é: inicializar variáveis globais, preparar arquivo de log, processar parâmetros passados pelo programa, carregar arquivos de configuração, validar usuário, validar permissão de acesso e conexão com o banco de dados. Esta função é altamente crítica, pois todos os programas do sistema a utilizam na sua inicialização, portanto, se a mesma falhar, nenhum programa irá ser executado corretamente.

A escolha pelo arquivo `di_inicio_prog.4gl` se deu devido ao forte acoplamento existente, e a escolha da função `inicio_prog` se deve aos seguintes motivos:

1) Grande em número de linhas, atualmente com 930 linhas;
2) Complexa, dificultando o entendimento;
3) Alto acoplamento, dificultando a manutenibilidade;
4) Baixa coesão;
5) Duplicação de código, e;
6) Baixa testabilidade.

Com a escolha do arquivo do módulo e função a ser refatorada, foi possível através da IDE de desenvolvimento, gerar diagramas de sequência da função `inicio_prog`, a fim de visualizar o grau de interação que a função tem com outros módulos e as funções que ela invoca. A figura 7 mostra o diagrama de sequência da função `inicio_prog`.

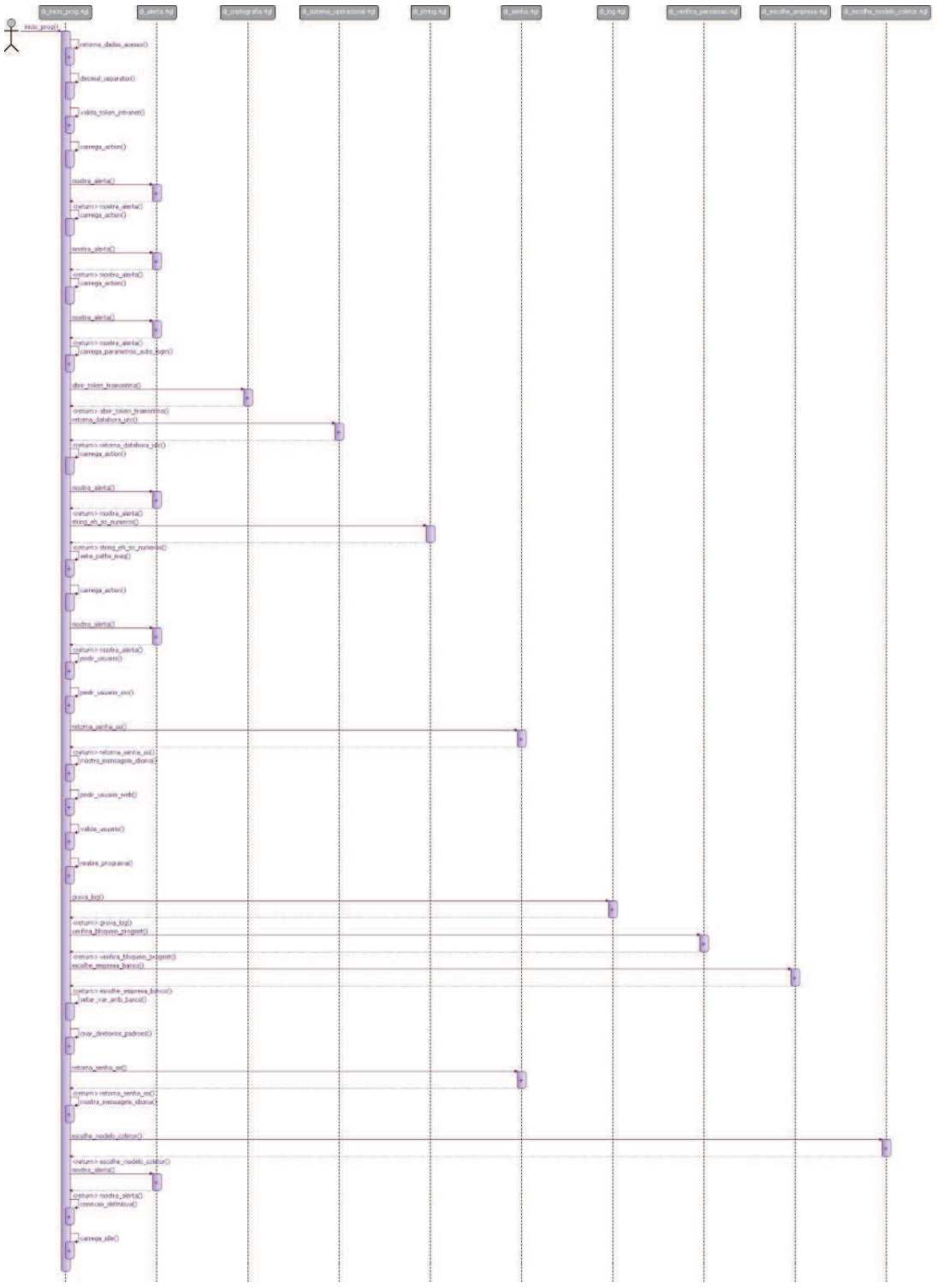


**Figura 7. Diagrama de sequência da função “inicio_prog”**

**Fonte: Elaborado pelo autor**

É possível observar na figura 7 que a função `inicio_prog` tem um grande acoplamento, pois como pode ser visto horizontalmente, depende de outros 9 arquivos do Módulo DI para realizar o seu trabalho. Também é possível observar a baixa coesão, pois realiza trabalho excessivo, isso pode ser visto na vertical.

Após ter identificado os pontos a serem refatorados, e antes de começar a modificar o código, foi extremamente importante familiarizar-se com o seu entorno. Para isso foi preciso ler o código várias vezes, saltar de arquivo para arquivo, conversar com outros desenvolvedores e pensar sobre como seria possível organizar o projeto um pouco melhor.

### 5.3. Aplicando as Técnicas de Refatoração

Esta subseção tem como objetivo apresentar como as refatorações foram aplicadas na prática, juntamente com alguns exemplos de como era antes e como ficou depois das refatorações.

Para este estudo prático, foram utilizadas 5 técnicas de refatoração. As técnicas apresentadas foram adaptadas para serem utilizadas em uma linguagem de programação procedural, pois as principais literaturas descrevem estas técnicas utilizando orientação a objetos.

#### 5.3.1. Extrair Método/Função

Extrair função constitui-se em dividir uma função grande em pequenas funções que terão nomes apropriados, tornando-os um código autoexplicativo, que terá maior legibilidade para seus desenvolvedores [FOWLER 2004].

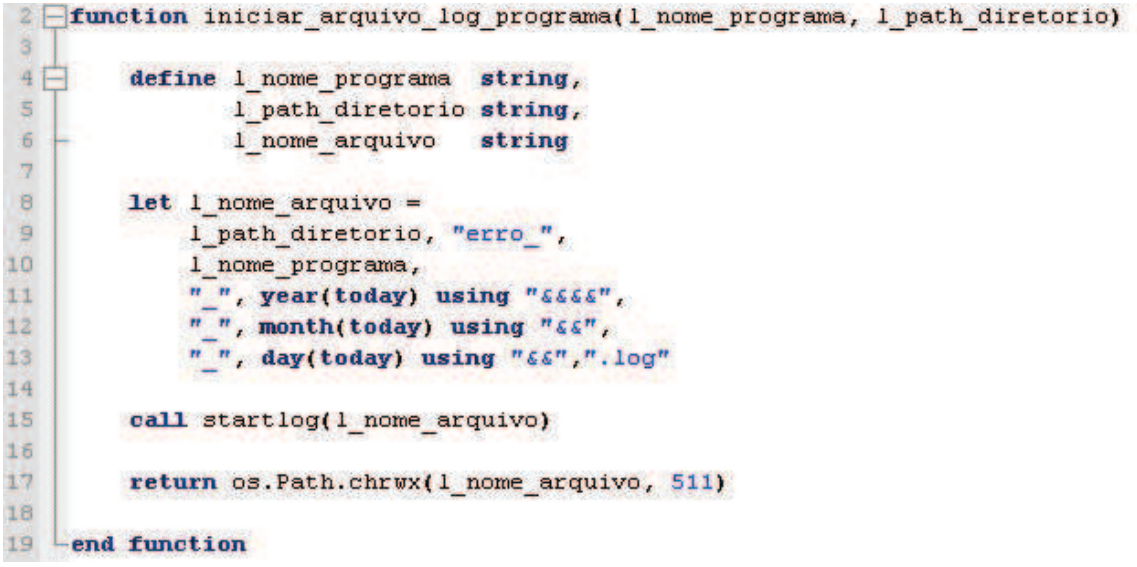
O trecho de código apresentado na figura 8 estava contido dentro da função `inicio_prog`, utilizando-se de variáveis globais do módulo, tornando difícil a implementação de uma função de teste para este código.

```
1   --- Inicializa os Logs
2   let gg_path.erro_log = fgl_getenv("TRA_GENERODIR"),"/logs/"
3
4   let l_aux = gg_path.erro_log, "erro_",
5               base.application.getProgramName() clipped,
6               "_", year(today) using "&&&&",
7               "_",month(today) using "&&",
8               "_",day(today) using "&&",".log"
9
10  call startlog(l_aux)
11
12  call os.Path.chrwx(l_aux, 511) returning l_status
```

**Figura 8. Trecho de código original**

**Fonte: Elaborado pelo autor**

Após a refatoração, o trecho de código ficou com nome de função coerente ao que realiza, baixo acoplamento, alta coesão e altamente testável. Isso se deve, pois o trecho de código foi extraído da função `inicio_prog` e foi eliminada a dependência de variáveis globais do módulo, conforme mostrado na figura 9.



```
function iniciar_arquivo_log_programa(l_nome_programa, l_path_diretorio)

    define l_nome_programa  string,
           l_path_diretorio string,
           l_nome_arquivo   string

    let l_nome_arquivo =
        l_path_diretorio, "erro_",
        l_nome_programa,
        "_", year(today) using "&&&&",
        "_", month(today) using "&&",
        "_", day(today) using "&&",".log"

    call startlog(l_nome_arquivo)

    return os.Path.chrwx(l_nome_arquivo, 511)

end function
```

**Figura 9. Trecho de código refatorado**

**Fonte: Elaborado pelo autor**

É possível observar na figura 9 um aumento no número de linhas de código em relação ao código original mostrado na figura 8, mas isso é irrelevante diante das diante dos benefícios obtidos.

Esta técnica de refatoração permitiu a reutilização de vários trechos de código e fez com que as funções que o chamam ficassem mais fáceis de se compreender.

### 5.3.2. Mover Método/Função

Mover função é usado, normalmente, quando o módulo/arquivo têm um excesso de comportamento, gerando um alto nível de acoplamento e baixa coesão. Com esta técnica pode-se simplificar o módulo/arquivo, definir melhor a responsabilidade, deixando suas funções mais claros [FOWLER 2004].

O arquivo `di_inicio_prog.4gl` continha muita responsabilidade, pois agrupava funções relacionada a configuração de programas, validação e permissão de usuário, mensagens do tipo popup, controle de exceções e conexão com o banco de dados.

Para minimizar este problema, algumas das funções do arquivo `di_inicio_prog.4gl` foram realocadas em outros arquivo do módulo mais apropriados, conforme quadro 4.

**Quadro 4. Funções movidas para outros arquivos do módulo**

| Função | Arquivo destino |
|---|---|
| acesso_liberado | di_usuario.4gl |
| carrega_estabelecimentos_do_usuario | di_usuario.4gl |
| he_usuario_root | di_usuario.4gl |

| | |
|---|---|
| pedir_usuario_web | di_usuario.4gl |
| pedir_usuario_ow | di_usuario.4gl |
| pedir_usuario_rfid | di_usuario.4gl |
| pedir_usuario | di_usuario.4gl |
| valida_usuario | di_usuario.4gl |
| seta_idioma_login_rfid | di_usuario.4gl |
| seta_idioma_login | di_usuario.4gl |
| verifica_situacao_afastamento | di_usuario.4gl |
| carregar_combo_idioma_login | di_usuario.4gl |
| valida_token_intranet | di_criptografia.4gl |
| he_senha_suporte | di_senha.4gl |
| erro_stop | di_rot_erro.4gl |
| erro_cont | di_rot_erro.4gl |
| mostra_informacao_idioma | di_alerta.4gl |
| mostra_mensagem_idioma | di_alerta.4gl |
| mostra_mensagem_login | di_usuario.4gl |
| carregar_info_encerramento_empresa | di_escolhe_empresa.4gl |

**Fonte: Elaborado pelo autor**

Através desta técnica, foram realocadas 20 funções para outros arquivos, reduzir consideravelmente a responsabilidade do arquivo `di_inicio_prog.4gl`, deixando o mesmo mais coeso e com menos número de linhas.

### 5.3.3. Decompor Condicional

Lógicas condicionais podem ser escritas de forma bem complexa, e normalmente são encontradas dentro de funções muito longos. Esta técnica apresenta a intenção de tornar as expressões condicionais mais claras, decompondo e substituindo blocos de códigos por chamadas a funções cujo o nome representa a intenção deste bloco de código [FOWLER 2004].

O trecho de código apresentando na figura 10, representa uma expressão condicional simples, porém esta expressão foi encontrado inúmeras vezes dentro do arquivo `di_inicio_prog.4gl`, gerando problemas de duplicação de código.

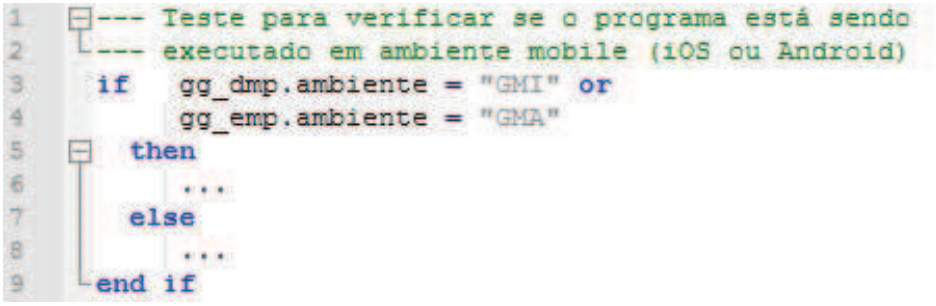



**Figura 10. Trecho de código original**

**Fonte: Elaborado pelo autor**

Com a refatoração, a expressão condicional foi transformada em uma função simples, com nome coerente com a expressão que realiza, sem comentários, desacoplada e altamente testável, pois está separada da função `inicio_prog` e não utiliza-se de variáveis globais do módulo, conforme mostrado na figura 11.

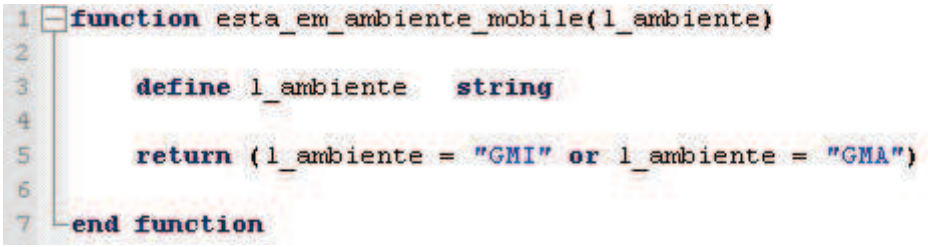



**Figura 11. Trecho de código refatorado**

**Fonte: Elaborado pelo autor**

Com esta técnica, foi possível quebrar diversas condicionais existentes na função `inicio_prog` em pequenas funções, melhorando a reutilização e legibilidade, pois o nome da função expressa sua condição.

### 5.3.4. Condensar Hierarquias

Essa refatoração é utilizada quando existem módulos que são semelhantes em suas responsabilidade, ou tratam do mesmo assunto, podemos então mover todas as funções de um módulo para outro, excluindo o arquivo do módulo caso o mesmo tenha ficado vazio [FOWLER 2004].

Com a utilização desta técnica, verificamos que 4 arquivos do módulo tinham responsabilidades semelhantes, que são:

1) `di_usuario.4gl` - Este arquivo contém funções relacionadas ao usuário do sistema, como por exemplo: funções para obter dados cadastrais do usuário, funções de atualização destes dados no banco de dados, validações em geral referente a usuário, etc;
2) `di_senha.4gl` - Este arquivo contém funções relacionadas a senhas do usuário, como por exemplo: função para obter a senha do usuário no banco de dados, função alterar a senha do usuário, etc;
3) `di_alerta.4gl` - Este arquivo contém todo tipo de funções para apresentar mensagens do tipo popup para o usuário, e;

4) `di_rot_mens.4gl` - Este arquivo também contém algumas funções para apresentar mensagens para o usuário.

É possível notar que os arquivos `di_usuario.4gl` e `di_senha.4gl` tem responsabilidades em comum, pois ambos arquivos contém funções relacionadas a usuários do sistema. Igualmente, o arquivo `di_alerta.4gl` e `di_rot_mens.4gl` têm responsabilidades comuns, pois ambos arquivos contém funções relacionadas a apresentarem mensagens na tela do o usuário.

Desta forma, foram realizar duas refatorações, transferindo todas as funções do arquivo `di_senha.4gl` para o arquivo `di_usuario.4gl` e eliminando o primeiro. Da mesma forma, foram transferidas todas as funções do arquivo `di_rot_mens.4gl` para o arquivo `di_alerta.4gl` e eliminando o primeiro.

### 5.3.5. Renomear nomes de Métodos/Funções/Módulos/Variáveis

Essa refatoração é utilizada quando nomes de um métodos, funções, módulos ou variáveis não revela seu real propósito, deve-se então alterar este nome para que fique de fácil compreensão para o desenvolvedor [FOWLER 2004].

Com o intuito de melhorar a legibilidade e compreensão das funções, o quadro 5 apresenta algumas das funções que foram renomeadas.

**Quadro 5. Funções do arquivo `di_inicio_prog.4gl` renomeadas**

| Nome Original | Nome Refatorado | Propósito da Função |
|---|---|---|
| inicio_prog | inicializar_programa | Inicializar as configurações básicas para o funcionamento de um programa |
| retorna_dados_acesso | retornar_config_arquivo_xml | Função responsável por ler e retornar as configurações do arquivo XML base. |
| carrega_action | carregar_arquivo_action_default | Função responsável por carregar na memória o arquivo de “actions defaults” para o programa. |
| carrega_idle | retornar_tempo_limite_ociosidade_programa | Função que retorno o tempo limite em segundos de ociosidade de um programa. |
| decimal_separator | config_separador_decimal_teclado | Função para configurar o tecla de separador decimal no teclado numérico. |
| seta_paths_maq | config_paths_servidor | Função responsável por configurar os “paths” de um servidor. Caminho onde irão |

| | | |
|---|---|---|
| | | ser gerados os relatórios, arquivos temporários, etc. |

**Fonte: Elaborado pelo autor**

Com o mesmo intuito de melhorar a legibilidade e compreensão, o quadro 6 apresenta alguns dos arquivos do Módulo DI que foram renomeados.

**Quadro 6. Arquivos do Módulo DI renomeados**

| **Nome Original** | **Nome Refatorado** | **Propósito do Módulo** |
|---|---|---|
| di_inicio_prog.4gl | di_programa.4gl | Conter apenas funções referentes a informações básicas de um programa. |
| di_escolhe_empresa.4gl | di_empresa.4gl | Conter apenas funções referentes a um empresa que utiliza o sistema. |
| di_escolhe_modelo_coletor.4gl | di_coletor.4gl | Conter apenas funções referentes aos coletores de dados utilizados no chão de fábrica. |
| di_alerta.4gl | di_mensagem_popup.4gl | Conter apenas funções referentes a mensagem do tipo popup para o usuário. |
| di_string.4gl | di_string_util.4gl | Conter apenas funções referentes a manipulação de String. |

**Fonte: Elaborado pelo autor**

### 5.4. Discussão Sobre Aplicação das Técnicas de Refatoração

Após o processo de refatoração ser aplicado, os diagramas de dependência e sequência foram gerados novamente, a fim de comparação com os primeiros diagramas. A figura 12 mostra o diagrama de dependência do Módulo DI após o processo de refatoração.

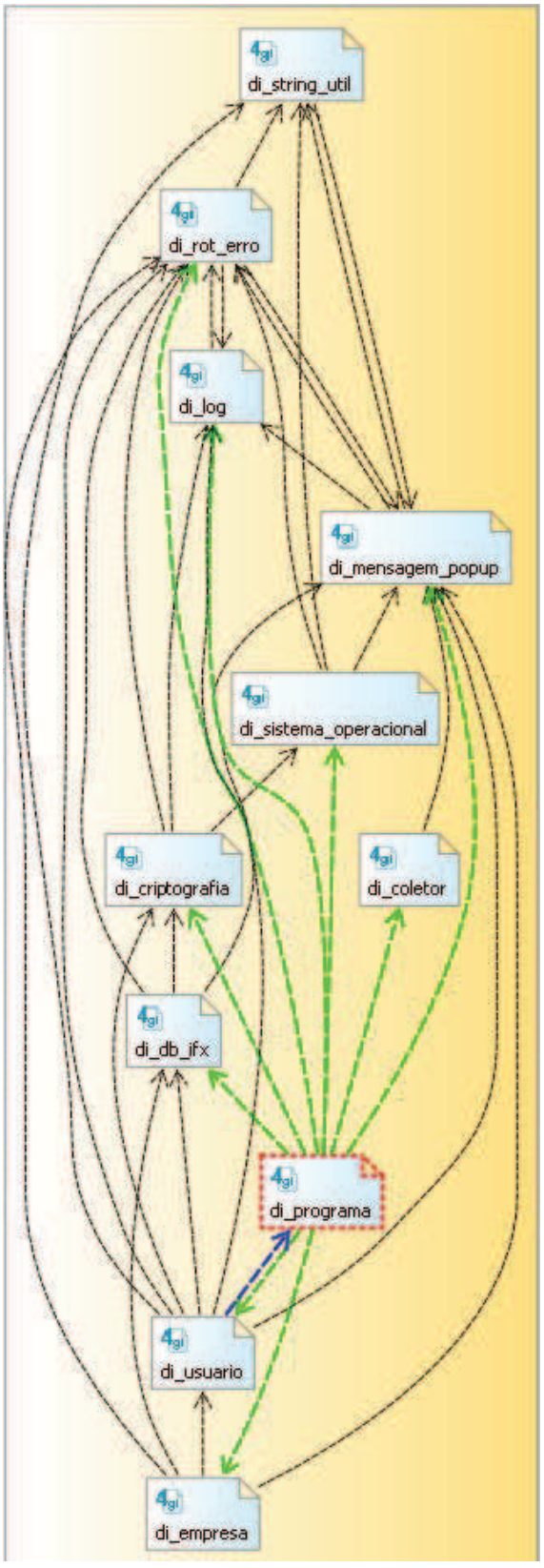



**Figura 12. Diagrama de Dependência do Módulo DI após as refatorações**

**Fonte: Elaborado pelo autor**

Realizando um comparativo entre o diagrama de dependência atual (figura 12) e o diagrama de dependência antes das refatorações (figura 6), é possível observar 5 pontos de melhoria:

1) Melhora na compreensão do significado dos arquivos, pois foram renomeados;
2) Diminuição do número de arquivos do módulo, que passou de 13 para 11, devido a refatoração na hierarquia;
3) Aumento da coesão dos arquivos, devido a redistribuição das funções entre os mesmo;
4) Diminuição da dependência entre os arquivos do módulo, pois existem menos setas entre os arquivos do módulo, e;
5) Diminuição na referência circular do arquivo `di_programa.4gl` (antigo `di_inicio_prog.4gl`), também devido a redistribuição das funções.

A figura 13 mostra o diagrama de sequência gerado após as refatorações da função `inicializar_programa` (antiga `inicio_prog`), que foi a função principal de estudo deste trabalho.

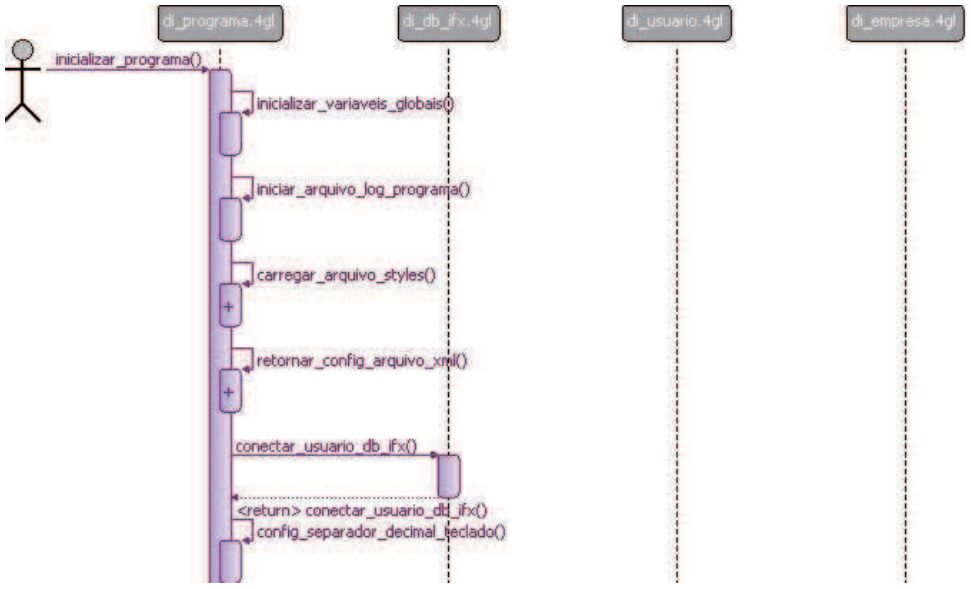



**Figura 13. Parte do Diagrama de sequência da função inicializar_programa refatorada**

**Fonte: Elaborado pelo autor**

Realizando um comparativo entre o diagrama de sequência atual (figura 13) e o diagrama de sequência antes das refatorações (figura 7), é possível observar que houve uma diminuição significativa do acoplamento da função `inicializar_programa` com outros arquivos do módulo, passando de 9 para apenas 3 arquivos. Isso se dá devido a redistribuição das funções entre os arquivos do módulo.

Através da percepção do pesquisador, foi possível observar outros pontos de melhoria no código:

1) Diminuição significativa do LOC da função `inicializar_programa`, que passou de 930 para 123 LOC, devido principalmente a técnica de extrair método/função;

2) Melhora na manutenibilidade da função e das funções que a mesma chama, pois a complexidade foi diluída em funções menores e em arquivos apropriados;
3) Melhora na legibilidade do código, pois as funções ficaram menores e mais simples, e;
4) Aumento significativo da testabilidade do arquivo `di_programa.4gl`, pois as funções se tornaram mais simples e com menor acoplamento.

O quadro 7 apresenta um comparativo geral do Módulo DI e da função `inicializar_programa` antes das refatorações e como ficou depois da aplicação das técnicas.

**Quadro 7. Comparativo geral do antes e como ficou depois das refatorações**

| | **Antes** | **Depois** | **% Redução** |
|---|---|---|---|
| **Módulo DI** | | | |
| Número de arquivos | 13 | 11 | 15 |
| Acoplamento entre arquivos do módulo (setas no diagrama de dependência) | 49 | 37 | 14 |
| **Arquivo di_programa.4gl** | | | |
| Tamanho do arquivo (KB) | 151 | 61 | 60 |
| LOC | 4.300 | 1.765 | 59 |
| Quantidade de funções | 41 | 28 | 32 |
| Média de LOC/função | 104 | 63 | 40 |
| **Função inicializar_programa** | | | |
| LOC | 930 | 123 | 87 |

**Fonte: Elaborado pelo autor**

## 5.5. Aplicando Testes Unitários

Para auxiliar no desenvolvimento e execução dos testes, foi utilizada a ferramenta GGC (*Genero Ghost Client*), que faz parte da suíte de desenvolvimento utilizada pela empresa. Esta ferramenta está em seu estágio inicial de desenvolvimento e no momento não conta com interface gráfica para apresentação dos resultados ao desenvolvedor, os resultados são apresentados em modo texto.

Os testes unitários foram elaborados com o intuito de testar a maior parte das funções existentes no arquivo `di_programa.4gl` (originalmente chamado de `di_inicio_prog.4gl`) pertencente ao Módulo DI. Todos os testes realizados para este arquivo foram adicionados a um novo arquivo chamado `di_programa_test.4gl` dentro da pasta do projeto, desta forma, a medida que os

testes forem criados, cada arquivo do módulo terá um arquivo de testes correspondente com o mesmo nome acrescentado do sufixo `_test.4gl`.

Da mesma forma, para cada função, foi criada uma função de teste com o mesmo nome da função original acrescentando ao final o sufixo `_test`. Esta função de teste contém todos os casos de testes desenvolvidos para a determinada função.

A figura 14, mostra a função `carregar_arquivo_styles` já refatorada do arquivo `di_programa.4gl`. Esta é uma função simples, que tem o objetivo de carregar o arquivo de estilo que o programa irá utilizar dependendo do ambiente que estiver sendo executado, onde o ambiente pode ser *Desktop* ou *Mobile*. O retorno da função também é simples, verdadeiro caso consiga realizar a carga do arquivo, caso contrário retorna falso.

```
function carregar_arquivo_styles(l_ambiente)

    define l_ambiente   string
    define l_style_name string

    if   esta_em_ambiente_mobile(l_ambiente)
      then
         let l_style_name = "styles_mobile"
      else
         let l_style_name = "styles"
    end if

    try
        call ui.Interface.loadStyles(l_style_name)
    catch
        call rot_erro(status||" - "||err_get(status)) # Registra no Log de Erro
        return false
    end try

    return true

end function
```

**Figura 14. Código da função carregar_arquivo_styles**

**Fonte: Elaborado pelo autor**

A figura 15 mostra a função desenvolvida para testar a função da figura 14. Esta função de teste foi alocada no arquivo `di_programa_test.4gl`.

A função de teste possui dois casos de teste, onde o objetivo é percorrer todos os caminhos possíveis da função a ser testada, sendo que, em ambos os casos o retorno esperado deve ser verdadeiro.

```
5  function carregar_arquivo_styles_test()
6
7      # Caso de teste 1
8      # Testando a função em ambiente de execução Desktop
9      ASSERT_EQUALS(carregar_arquivo_styles("GDC"), true)
10
11     # Caso de teste 2
12     # Testando a função em ambiente de execução Mobile (iOS)
13     ASSERT_EQUALS(carregar_arquivo_styles("GMI"), true)
14
15 end function
```

**Figura 15. Código da função de teste carregar_arquivo_styles_test**

**Fonte: Elaborado pelo autor**

Com a utilização da ferramenta GGC, foi executado a função de teste da figura 15. A ferramenta apresentou o resultado, mostrando o arquivo que foi executado, a função de teste executada, o resultado, informando se passou no teste ou se deu erro e o tempo decorrido para a execução dos teste, conforme mostrado na figura 16.

```
1  *** Running 'di_programa_test' ***
2
3  :::info:(GS-1025) Display client already running on 'localhost:6401'
4  :::info:Started 2017-07-10 09:28:30.515
5
6  di_programa_test.4gl:carregar_arquivo_styles_test:35:info:carregar_arquivo_styles("GDC") - Passed
7  di_programa_test.4gl:carregar_arquivo_styles_test:39:info:carregar_arquivo_styles("GMI") - Passed
8
9  :::info:Finished 2017-07-10 09:28:30.518
10
11 No error detected.
12
13 *** Execution of 'di_programa_test' finished. Exit code: 0 ***
```

**Figura 16. Resultado da execução do teste da função carregar_arquivo_styles_test**

**Fonte: Elaborado pelo autor**

No exemplo da figura 16, é possível observar que os 2 casos de testes foram executados com sucesso. Um ponto muito importante para a automação de testes é a velocidade da execução dos mesmo. É possível observar na figura 16 o tempo de execução do teste, que foi extremamente baixo.

## 5.6. Discussão Sobre Aplicação dos Testes Unitários

A ferramenta de testes se mostrou promissora, mesmo estando no seu estágio inicial, pois mesmo sem muitos recursos, contempla os conceitos básicos de "*asserts*" e também é possível utilizá-la através de script de automação de testes.

Apesar da literatura normalmente mostrar a aplicação de testes em linguagens orientadas a objetos, foi possível verificar que mesmo em linguagens procedurais, como é o caso da linguagem de programação *Genero*, a aplicação de testes unitários é viável, pois foi possível aplicar testes na maioria das funções do arquivo `di_programa.4gl`.

O quadro 8 apresenta alguns números obtidos ao final do processo de implementação e execução dos testes.

**Quadro 8. Resultado da implementação e execução dos testes**

| Descrição | Valor |
|---|---|
| Quantidade de funções do arquivo `di_programa.4gl` após a refatoração | 28 |
| Quantidade de funções com testes | 22 |
| Quantidade de funções sem testes | 6 |
| % de funções com cobertura de testes | 78,5 |
| Quantidade total de casos de testes | 57 |
| Média de casos de testes por função de teste | 2,6 |
| Tempo total para a execução de todos os 22 testes | 0,123 s |

**Fonte: Elaborado pelo autor**

## 5.7. Heurísticas para Refatoração do Código Legado

Em decorrência da incapacidade da linguagem de programação *Genero* em fornecer métricas de *software*, métodos heurísticos foram criados e utilizados para guiar o presente estudo na escolha dos pontos de refatoração e testes.

Heurísticas são procedimentos de solução que muitas vezes se apoiam em uma abordagem intuitiva, na qual a estrutura particular do problema possa ser considerada e explorada de forma inteligente, para a obtenção de uma solução adequada [CUNHA 1997].

A figura 17 apresenta um resumo das heurísticas utilizadas no processo de refatoração do código legado da empresa.

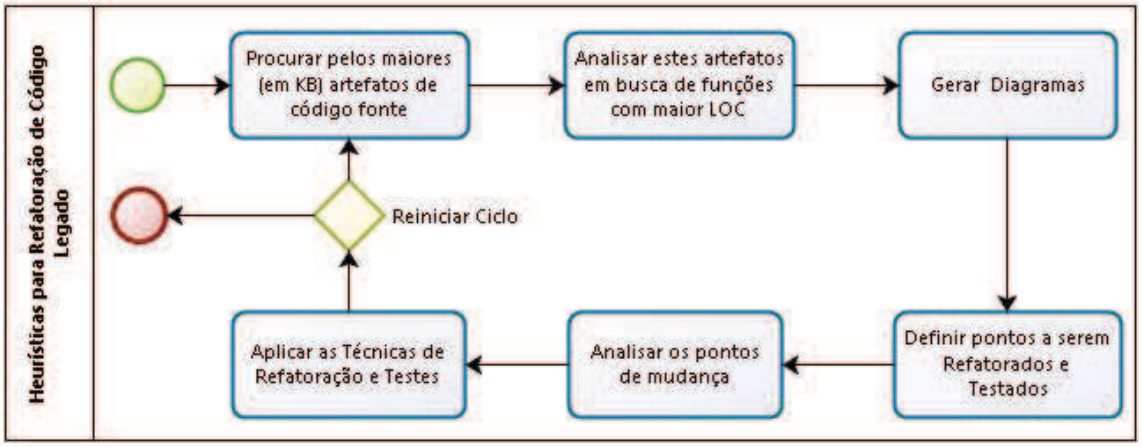



**Figura 17. Resumo das heurísticas utilizadas no processo de refatoração**

**Fonte: Elaborado pelo autor**

Conforme já apresentado na subseção 5.2, um ponto muito importante deste estudo foi como descobrir em que ponto do sistema deveriam ser aplicadas as refatorações e os testes para que realmente tenham efeitos positivos sobre o sistema.

Diante disso, o primeiro passo foi realizada uma pesquisa nos artefatos de código fonte, em busca dos 10 arquivos com tamanho excessivo, pois estes arquivos podem apresentar maus cheiros, como por exemplo, numero excessivo de funções e baixa coesão, pois podem realizar tarefas demais.

O segundo passo foi analisar os 10 artefatos encontrados em busca das funções com maior LOC, pois funções com muitas linhas se tornam muito complexas, deixando a manutenção cada vez mais difícil e dificultando a testabilidade.

Com a ajuda da IDE de desenvolvimento, o terceiro passo foi gerar diagramas de dependência dos módulos onde se encontram os 10 maiores arquivos. Através destes diagramas, podemos ter uma visão geral do nível de dependência entre os arquivos do módulo. Neste ponto já é possível ter uma ideia dos módulos e arquivos que necessitam de refatoração.

Ainda com a ajuda de IDE de desenvolvimento, foi gerar diagramas de sequência das 10 funções com maior LOC. Através destes diagramas, podemos ver o nível de interação que estas funções têm com outras funções de outros arquivos do módulo. Assim facilita a visualização das funções que realizam tarefas demais.

O quarto passo foi analisar as informações obtidas através dos dados coletados nos artefatos de código fonte e dos diagramas gerados. Neste ponto a experiência dos envolvidos é muito importante, pois nem sempre a função/método com maior LOC é a que deve ser refatorada por primeiro, isso pode variar de sistema para sistema, do nível acoplamento do módulo/arquivo, e principalmente da criticidade da função/método.

No quinto passo constitui-se em realizar uma análise mais aprofundada dos pontos a serem refatorados, com intuito de familiarizar-se como entorno dos pontos de refatoração para ganhar confiança e conhecimento. Para isso é preciso ler e compreender o código antes de refatorar.

O sexto passo foi realizar as refatorações e testes nos pontos definidos. No presente trabalho, o sistema atual da empresa não contempla nenhum tipo de cobertura de testes, então, esta foi uma etapa crítica, pois a chance de introdução de *bugs* é grande. Para minimizar este problema, as refatorações foram realizadas em pequenos passos (*baby steps*), fazendo pequenas mudanças de cada vez.

Por fim, podemos encerrar o processo de refatoração, ou podemos reiniciar o ciclo, iniciando novamente o processo de melhoria.

## 6. Conclusão

O presente trabalho apresentou questões relacionadas a sistemas legados que apresentam baixa qualidade de código, tornando a sua manutenção extremamente difícil, problema esse muitas vezes chamado de dívida técnica.

Nesse contexto, o objetivo deste trabalho foi realizar um estudo sobre práticas ágeis de refatoração e testes automatizados no desenvolvimento de *software* legado, visando verificar se, de fato estas técnicas podem minimizar a dívida técnica.

Este estudo teve um desafio particular, pois normalmente a literatura aborda a refatoração com linguagens de programação orientadas a objetos. No entanto, a empresa do presente trabalho adota uma linguagem totalmente procedural, mesmo assim, foi possível adaptar e utilizar as técnicas de refatoração e testes unitários com sucesso.

Para realizar este estudo prático, foram identificados pontos no sistema legado da empresa que apresentavam baixa qualidade do código. Diante disso, foi escolhido um destes pontos e foram aplicadas 5 técnicas de refatoração. A medida que as refatorações reduziam o acoplamento, aumentavam a coesão e deixavam as funções mais simples, também foram desenvolvidos testes unitários, visando melhorar a cobertura de testes do sistema.

Através da percepção do pesquisador e de comparações entre diagramas de como era antes e como ficou depois da aplicação das técnicas, foi possível observar uma melhora significativa na legibilidade, manutenibilidade e testabilidade do código, comprovando que estas técnicas de fato podem minimizar a dívida técnica do código no médio e longo prazo.

Outra contribuição importante, foi a descoberta de heurísticas para guiar os desenvolvedores da empresa no processo de identificação dos pontos onde o código legado provavelmente necessita de refatoração.

Apesar de ter sido feita a refatoração em apenas um módulo do sistema, um projeto de refatoração bem definido e planejado, pode levar este sistema a ter uma vida útil por muito mais tempo. Além de diminuir o esforço para compreensão e manutenções do código. Espera-se através disso, manter uma melhoria contínua no código legado.

A partir do estudo realizado, seria importante realizar novos estudos sobre técnicas de refatoração para banco de dados. Outra sugestão seria avaliar o quanto de esforço e custo seria necessário para minimizar a dívida técnica de um sistema legado.

## 7. Referências


Aniche, Maurício (2013) “Test-Driven Development - Teste e Design no Mundo Real”. Editora Casa do Código.

Azevedo, D., Machado, L. and Silva, L. V. da. (2011) “Métodos e procedimentos de pesquisa: do projeto ao relatório final”, São Leopoldo, RS.

Barrozo, G. C.; Vinhas, H. M. (2012) “Refatoração: Aperfeiçoando um código existente”, Artigo académico, UNIFENAS - Universidade de Alfenas, Alfenas, MG.

Bartié, Alexandre (2002) “Garantia da Qualidade de Software”, Rio de Janeiro, RJ: Elsevier.

Beck, K. (2010) “TDD Desenvolvimento Guiado por Testes”, 1ª Edição, Porto Alegre: Bookman.

Bernardo, Paulo C. (2011) “Padrões de testes automatizados”, Dissertação de Mestrado, IME-USP - Instituto de Matemática e Estatística da Universidade de São Paulo, São Paulo, SP.

Caetano, C. (2008) “Introdução à Automação de Testes”, Revista Engenharia de Software Magazine, 5ª edição. p60-66.

Caetano, C. (2008) “Melhores Práticas na Automação de Testes”, Revista Engenharia de Software Magazine, 5ª edição. p42-47.

Chaves, M. L. (2011) “A importância dos testes unitários em uma empresa de desenvolvimento de software”, Trabalho de conclusão de curso, UNIPLAC - Universidade do Planalto Catarinense, Lages, SC.

Cunha, C. B. (1997) “Uma contribuição para o problema de roteirização de veículos com restrições operacionais”, São Paulo: EPUSP, Departamento de Engenharia de Transportes, Tese de Doutoramento.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. (2007) “Introdução ao teste de software”, Rio de Janeiro, RJ: Elsevier.

Devmedia (2015) “TDD: fundamentos do desenvolvimento orientado a testes”, Disponível em: <http://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151> acesso em Julho 2017.

Engel, G. I. (2000) “Pesquisa-ação. Revista Educar. [on-line]”, Editora da UFPR, N.º 16, Curitiba, http://www.educaremrevista.ufpr.br/arquivos_16/irineu_engel.pdf, Setembro.

Fowler, Martin (2004) “Refatoração: Aperfeiçoando O Projeto de Código Existente”, Porto Alegre: Bookman.

Furlan, José Davi (1998) “Modelagem de objetos através da UML: análise e desenho orientados a objetos”, São Paulo: Makron books.

Graham, Dorothy, Fewster, Mark (2012) “Experiences of Test Automation: Case Studies of Software Test Automation”, Boston: Addison-Wesley Professional.

Gil, A. C. (2010) “Como elaborar projetos de pesquisa”, São Paulo, Atlas, 5ª edição.

Heredia, J. C. C. (2015) “Refatoração em um Sistema Legado”, Trabalho de Conclusão de Curso, PUC Minas, Uberlândia, MG.

Kaplan, Bonnie; Duchon, Dennis (1988) “Combining qualitative and quantitative methods in information systems research: a case study”, MIS Quarterly, v. 12, n. 4, Dec.

Kerievsky, Joshua (2008) “Refatoração para Padrões”, São Paulo: Bookman.

Marinescu, Floyd (2002) “Padrões de Projeto EJB”, Porto Alegre: Bookman.

Martins, A. B. T.; Hauck, J. C. R. (2016) “Automação de Testes em Sistemas Legados: Um Estudo de Caso para a Plataforma Flex”, III Workshop de Iniciação Científica

em Sistemas de Informação, UFSC - Universidade Federal de Santa Catarina, Florianópolis, SC.

Molinari, Leonardo (2003) "Testes de Software: Produzindo sistemas melhores e mais confiáveis", São Paulo: Érica.

Moraes, Roque (1999) "Análise de conteúdo". Revista Educação, Porto Alegre, v. 22, n. 37, p. 7-32.

Pressman, Roger S (2011) "Engenharia de Software Uma Abordagem Profissional", 7ª Ed., AMGH.

Santos, M. F. (2011) "Qualidade de software através de testes unitários", Monografia de Pós-Graduação, Universidade do Sul de Santa Catarina, Araranguá, SC.

Sommerville, Ian (2011) "Engenharia de Software", 9ª Ed., Addison Wesley: São Paulo.

Warren, Ian (2000) "The Renaissance of Legacy Systems: Method to Support Software System Evolution", Practitioner series, Springer.

Wainer, J. (2007) "Métodos de Pesquisa Quantitativa e Qualitativa para a Ciência da Computação". Disponível em: http://www.ic.unicamp.br/~wainer/papers/metod07.pdf

Yin, Robert K. (2010) "Estudo de caso: planejamentos e métodos." 4ª ed. Porto Alegre: Bookman.