UNIVERSIDADE DE CAXIAS DO SUL

GIOVANI BIONDO

UM PROCESSO DE CONVERSÃO DE SISTEMAS LEGADOS PROCEDURAIS PARA ORIENTADO A OBJETOS, DIRECIONADO PELA ARQUITETURA MVC

BENTO GONÇALVES

2017

GIOVANI BIONDO

UM PROCESSO DE CONVERSÃO DE SISTEMAS LEGADOS PROCEDURAIS PARA ORIENTADO A OBJETOS, DIRECIONADO PELA ARQUITETURA MVC

Relatório do Trabalho de Conclusão apresentado ao curso de Sistemas de Informação como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação, no Campus Universitário da Região dos Vinhedos, da Universidade de Caxias do Sul.

Orientador: Prof. Me. Joacir Giaretta

Bento Gonçalves

2017

RESUMO

Com o passar do tempo, um produto de software é submetido a diversas atualizações, normalmente atribuídas a constantes alterações, adição de funcionalidades e correções de problemas. Tais fatores vão deteriorando a capacidade de manutenção do produto, chegando ao ponto de que manter estes softwares é praticamente inviável devido à complexidade em demasia ou inúmeros remendos realizados ao longo dos anos. Este trabalho apresenta uma solução, através do estabelecimento de um processo de conversão destes sistemas legados, desenvolvidos proceduralmente, em orientados a objetos, levantando requisitos e conceitos de organização da arquitetura MVC (*Model-View-Controller*). Foi considerada de grande importância a conceitualização de tópicos sobre linguagens de programação e engenharia de software. Embora existam produtos de software que façam o processo de conversão de algumas linguagens, nem sempre é possível ter um produto final funcional sem a realização manual do processo de conversão, estabelecendo assim uma relação atualizada entre os requisitos do software e seu código-fonte. Ao término do projeto, foi possível ter uma funcionalidade convertida, com o código-fonte desenvolvido através do paradigma da orientação a objetos, atendendo à necessidade do sistema e uma documentação básica do seu funcionamento, através de casos de uso e requisitos funcionais.

Palavras-chave: Processo de conversão. Orientação a objetos. Procedural. Legado. Paradigma de programação. MVC. Código. Software.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

MVC – Model-View-Controller

OO – Orientação a Objetos

PP – Programação Procedural

LSI – Large Scale Integration

VLSI – Very Large Scale Integration

ULSI – Ultra Large Scale Integration

FORTRAN – FORmula TRANslator

COBOL – Common Business Oriented Language

VDF – Visual Dataflex

UML – Unified Modeling Language

SUMÁRIO

1 INTRODUÇÃO

A linguagem natural é um instrumento fundamental para nos comunicarmos e expressarmos pensamentos e sentimentos. Conforme a necessidade de expressar mais e mais informações foi crescendo, a língua foi sofrendo mudanças. Foram, por exemplo, adicionados verbos, juntamente com suas conjugações, para expressar ações passadas, presentes e futuras. Isso adicionou complexidade à linguagem natural, e o mesmo ocorre com as linguagens de programação. Cada vez mais, os desenvolvedores criam algoritmos de maior complexidade, crescendo para fora dos paradigmas preestabelecidos, forçando o desenvolvimento e a maturidade de paradigmas mais robustos.

Paradigmas de programação podem ser definidos como a "forma" em que é desenvolvido algo, podendo até ser chamados de estilo. Existem inúmeros critérios abstratos para a escolha das condições paradigmáticas que definem o que pertence a um paradigma ou não; por exemplo: estrutura do programa e metodologia. Algumas linguagens facilitam a escrita em alguns paradigmas e outras não (WEGNER, 1990).

Com o surgimento da linguagem Pascal e C, em 1970 e 1972, respectivamente, foi concebido o paradigma de programação procedural. Este paradigma é uma derivação do paradigma imperativo, no qual o conceito principal é enviar comandos sequenciais à máquina a fim de que, no final da execução, um problema seja resolvido; esses comandos são chamados de procedimentos. As linguagens de programação que suportam exclusivamente – ou pelo menos parcialmente – o estilo procedural, fazem parte da maioria das linguagens existentes. Isso se deve ao fato de que as arquiteturas dos computadores estão baseadas no modelo de von Neumann (BARANAUSKAS, 2012).

Na década de 90, o paradigma de programação orientado a objetos começou a ser aceito no mercado. Teve seu termo cunhado por Alan Curtis Kay, criador da linguagem Smalltalk, apesar de alguns conceitos deste paradigma já terem sido previamente estabelecidos em 1967 por Ole-Johan Dahl e Kristen Nygaard, através da criação do Simula 67 (NOGUEIRA, 2008). O objetivo do paradigma orientado a objetos é estabelecer uma forma de desenvolvimento "natural", na qual conceitos do mundo real são aproximados ao desenvolvimento de software através de objetos e da forma como esses objetos interagem entre si (NOGUEIRA, 2008).

Independente do paradigma em que ele está desenvolvido, para melhorar a qualidade de um software é necessário adotar padrões de desenvolvimento. A partir destes padrões, moldar e descrever subsistemas e componentes predefinidos: a isso é dado o nome de arquitetura de software (SANTANA, 2011).

MVC (*Model-View-Controller*) é um padrão de arquitetura de software, que divide a aplicação em três componentes: *model* é a parte responsável por executar o processamento da rotina; *controller* que, como o nome sugere, controla as mensagens que são enviadas ao componente do *model* e serve como interface entre o *model* e suas *views* associadas; e *view* que apenas mostra certos aspectos do modelo através da interface com o usuário (KRASNER, POPE, 1988).

A escolha do paradigma orientado a objetos foi solidificada por três pilares fundamentais para este processo de conversão, descritos na seção 1.1. O primeiro foi a característica embutida do paradigma de fornecer uma base de código de grande reutilização devido a sua estrutura em classes. O segundo pilar foi a grande manutenibilidade: quando o software é desenvolvido através de conceitos de superclasses e classes derivadas, a manutenção se torna mais fácil, visto que, muitas vezes, apenas a superclasse deve ser alterada (SANTOS, 2015). E, por fim, existem limites para um processo de conversão, visto que nem todas as linguagens possuem conceitos compatíveis para que uma tradução seja realizada de forma simples, como por exemplo paradigma funcional para paradigma orientado a objetos (SOMMERVILLE, 2011).

Para softwares com uma alta incidência de manutenções, sejam correções de bugs ou melhorias, um sistema procedural terá sua dificuldade de manutenção multiplicada pelos anos em que está em uso, então é aceitável avaliar um processo para conversão destes programas.

A transição entre um sistema desenvolvido no paradigma procedural para um orientado a objetos é algo especialmente complexo para sistemas com anos de desenvolvimento e melhoria contínua, especialmente se o desenvolvimento não é apoiado em um planejamento restrito de requisitos a serem desenvolvidos. O processo seria basicamente inviável sem um forte apoio de um padrão de arquitetura consolidado (SANTOS, 2007).

1.1 PROBLEMA DE PESQUISA

Em tempos em que existe uma crescente preocupação das grandes empresas com a manutenção de sistemas legados, faz-se necessário analisar soluções para minimizar custos de manutenção e incidências de problemas derivados da alta manutenibilidade de sistemas legados.

Segundo Sommerville (2011, p.497) a definição de sistemas legados é:

> *Algumas organizações ainda dependem de sistemas de software que têm mais de vinte anos de existência. Muitos desses antigos sistemas ainda são fundamentais para as empresas, isto é, as empresas dependem dos serviços fornecidos pelo software, e qualquer falha desses serviços teria um sério efeito em seu dia-a-dia. A esses sistemas antigos foi dado o nome de Sistemas Legados.*

Em sua maioria, as empresas de software, quando escolhem adotar uma nova tecnologia ou querem otimizar o tempo de manutenção de seus desenvolvedores, possuem basicamente duas escolhas: desenvolver um novo produto a partir da concepção ou refatorar o sistema atual aplicando as melhorias necessárias em sua estrutura interna de código.

No caso de algumas empresas, é inviável o desenvolvimento de um produto inédito utilizando uma nova tecnologia, já que isso acarretaria custos com treinamento tanto de colaboradores quanto de clientes, além de todo o custo que a aquisição de novas licenças de software agregaria ao cenário atual da empresa. Então, a refatoração é uma saída para estas instituições.

Segundo Marin Fowler e Kent Beck (2001, p.16, tradução nossa), refatorar é

> *Uma alteração feita na estrutura interna do software para torná-lo mais fácil de entender e mais barato para modificar sem alterar seu comportamento observável…. É uma forma disciplinada de limpar código que minimiza as chances de introduzir bugs.*

A mesma lógica pode ser utilizada em um processo de conversão de sistemas legados procedurais em orientados a objetos. A linha que separa um processo de desenvolvimento de um novo produto e um de conversão, é basicamente que as rotinas já escritas serão reaproveitadas apenas adaptando à linguagem escolhida. Em alguns casos, existem ambientes de software que já suportam utilização tanto de paradigmas procedurais quanto orientados a objetos, mas, por um motivo de cultura ou do momento histórico em que o software foi desenvolvido, nunca foi utilizado o paradigma mais atual.

Dessa forma, o problema em questão, e que o trabalho objetiva resolver, é como realizar a modernização destes sistemas legados, através de paradigmas orientados a objetos já estabelecidos no mercado, mantendo seu funcionamento de uma forma segura e eficiente.

1.1.1 Questão de Pesquisa

É possível desenvolver um processo de conversão de sistemas legados escritos proceduralmente em orientado a objetos, utilizando conceitos da arquitetura MVC, mantendo a funcionalidade e estabilidade do software?

1.2 JUSTIFICATIVA

Devido à inquietude do mercado de software, diversos sistemas estabelecidos no mercado já estão ultrapassados. Por causa de sua complexidade, falta de conhecimento da empresa ou até atrasos no ciclo de desenvolvimento, mantém-se estagnados com tecnologias não mais funcionais para um mundo em constante mudança.

> *Somente através de mudança um software pode fornecer suporte para futuros esforços de desenvolvimento, assim nos esforçamos para aumentar o valor dos ativos do software tornando-o capaz de aceitar mudanças substantivas de estrutura. (WEIDERMAN, 1997, p.5, tradução nossa)*

O processo de modernização pode ser tão perigoso quanto a estagnação: após anos e anos de manutenção que basicamente são remendos, um software legado atinge uma complexidade desnecessária para seu funcionamento. Devido a isso, muitas empresas desenvolvedoras de software têm receio de alterar seus códigos-fonte, acarretar problemas para o usuário final e perder relevância no mercado.

O desenvolvimento desta monografia se justifica exatamente por esse receio de modernizar um software, por meio de um processo que espera padronizar a conversão de sistemas legados procedurais em orientados a objetos, através de conceitos de MVC, assim minimizando os riscos para a empresa e mantendo a funcionalidade e eficácia do sistema atual. Também será possível analisar o comportamento deste processo em um cenário real, em um software que já é utilizado

por centenas de usuários, avaliando a assertividade do processo e suas ramificações para quem decidir adotá-lo.

1.3 OBJETIVOS

A seguir serão apresentados os objetivos dessa pesquisa, os quais estão divididos em objetivo geral e específicos.

1.3.1 Objetivo geral

Desenvolver um processo de conversão, focado em requisitos, de sistemas legados construídos sob o paradigma procedural para o paradigma orientado a objetos, guiado pelos conceitos da arquitetura MVC e, posteriormente, validá-lo através de uma implementação em funcionalidade candidata.

1.3.2 Objetivos específicos

Este trabalho tem como objetivos específicos:

a) Realizar o levantamento da fundamentação teórica para apoiar o desenvolvimento do processo de conversão;
b) Desenvolver um processo de conversão de sistemas legados desenvolvidos proceduralmente para orientados a objetos através dos conceitos de MVC;
c) Aplicar o processo proposto em um cenário real e, baseado na experiência, realizar uma aplicação em funcionalidade candidata;
d) Analisar os resultados da aplicação e verificar a viabilidade do processo;
e) Realizar os ajustes necessários ao processo baseado nas experiências obtidas na aplicação do mesmo.

1.4 ESTRUTURA DO TRABALHO

A estrutura deste trabalho está dividida em seis capítulos, sendo que o primeiro introduz o assunto a ser discutido; nos capítulos dois, três e quatro, é

apresentado o embasamento teórico utilizado para suportar o processo de conversão. É importante salientar que todo referencial apresentado está direcionado a fornecer maior entendimento dos artefatos utilizados para construir o processo de conversão apresentado no capítulo cinco.

Por fim, no capítulo sete, realiza-se o encerramento e conclusão do assunto, explanando o fechamento do trabalho e o que foi possível aprender.

2 LINGUAGENS DE PROGRAMAÇÃO

Uma linguagem de programação – tal como a língua natural que utilizamos para falar – é constituída de palavras e frases, que nada mais são do que agrupamentos de palavras em uma ordem específica para produzir algum sentido. A diferença para as linguagens de programação está no fato de que as frases se tornam estruturas de programação e as palavras se tornam palavras-chave. (PUGA, RISSETTI, 2004).

Seguindo o paralelo com a língua natural, as linguagens de programação também seguem regras semânticas para construir frases que possuam sentido: isso recebe o nome de sintaxe. (PUGA, RISSETTI, 2004).

Existem diversos tipos de linguagens de programação que seguem diversos padrões diferentes: uma linguagem não pode ser considerada superior à outra, cada uma foi criada com um objetivo e função em mente. Por exemplo, uma linguagem pode ser superior na integração com serviços externos, outra pode ter melhor integração com o sistema operacional; tudo depende da adequação da tarefa à língua escolhida. (PUGA, RISSETTI, 2004).

2.1 CONTEXTO HISTÓRICO

A programação de computadores é um produto de anos de estudos em diversos campos por vários especialistas. Desde a lógica de Aristóteles, os fundamentos da lógica de Boole, até Turing na segunda guerra mundial, a história da programação é extensa e complexa: este trabalho não entrará no mérito da fundação da linguagem, mas sim no que compreende a segunda geração dos computadores (1964 – 1970), marcada pela introdução das linguagens de mais alto nível FORTRAN e LISP.

Durante a década de 50, antes do advento dos circuitos integrados, existiam diversas linguagens de alto nível sendo desenvolvidas, mas muito poucas obtiveram a aceitação que o FORTRAN (FORmula TRANslator) teve. Desenvolvida pela IBM Corporation, o FORTRAN surgiu com o objetivo de desenvolver aplicativos científicos que exigiam cálculos complexos e ainda é amplamente utilizada até os dias de hoje. (DEITEL, 2011).

Posteriormente, durante a década de 50, através das mãos do governo norte-americano, surgiu o COBOL (Common Business Oriented Language). Ao contrário do

FORTRAN, o foco desta linguagem era a utilização em softwares comerciais que exigiam manipulação eficiente de quantidades muito grandes de dados (DEITEL, 2011). Até 2014, instituições como o Banco do Brasil ainda utilizavam COBOL, mas com uma iniciativa de uma empresa privada chamada ELPaaS foram convertidas para Java devido à dificuldade de mão de obra (GUSMÃO, 2014).

No início da década de 70, os desenvolvedores de software encontraram uma barreira: as entregas estavam constantemente atrasadas e os custos de desenvolver softwares ultrapassavam constantemente o orçamento designado. Derivada de pesquisas e constantes buscas para resolver os problemas apresentados, nasceu a linguagem Pascal, criada em 1971 pelo professor Niklaus Wirth. Embora a linguagem Pascal fosse uma das primeiras na evolução da programação estruturada, não foi muito bem aceita nos círculos comerciais devido à falta de recursos necessários para uma implementação real e seu uso foi confinado basicamente a instituições de ensino (DEITEL, 2011).

Devido a necessidades do Departamento de Defesa (DOD) dos Estados Unidos, uma linguagem robusta era necessária para controlar os sistemas de software do departamento. Até o momento, eram utilizadas centenas de linguagens para gerenciar todos os requisitos. Assim, em 1980, nasceu a linguagem ADA, que apresentou um recurso muito importante para sua adoção pelo DOD: a multitarefa, um recurso que permite que o programador designe atividades que devem ser executadas paralelamente (DEITEL, 2011).

Mas foi na década de 90 que surgiu um dos maiores avanços na programação de computadores, a programação orientada a objetos (OO). Embora a tecnologia existisse desde 1960 com o C++, ela só foi amplamente aceita após a introdução de linguagens como Java, C# e retroativamente o C++. A característica fundamental da programação OO é a aproximação do ambiente de desenvolvimento ao mundo real do desenvolvedor. As linguagens estruturadas, ou até mesmo anteriores a elas, focavam no que seria realizado, ou seja, ações. Já a programação OO foca seus esforços no objeto, na entidade (DEITEL, 2011).

Segundo Charles Scalfani (2016), nos últimos anos o mercado está lentamente sendo direcionado à programação funcional, na qual dados e estados mutáveis são evitados. O enfoque, como o próprio nome sugere, são aplicações de funções. Embora algumas empresas já estejam alterando o foco do desenvolvimento de seu software para um modelo de desenvolvimento orientado a serviços (SOA),

como a gigante do entretenimento Netflix (HEMEL, 2013). Este tipo de arquitetura fornece uma base centralizada e uma arquitetura robusta suportando uma pesada carga de requisições através de serviços expostos (SOBRINHO, 2011).

2.2 PARADIGMAS DE PROGRAMAÇÃO

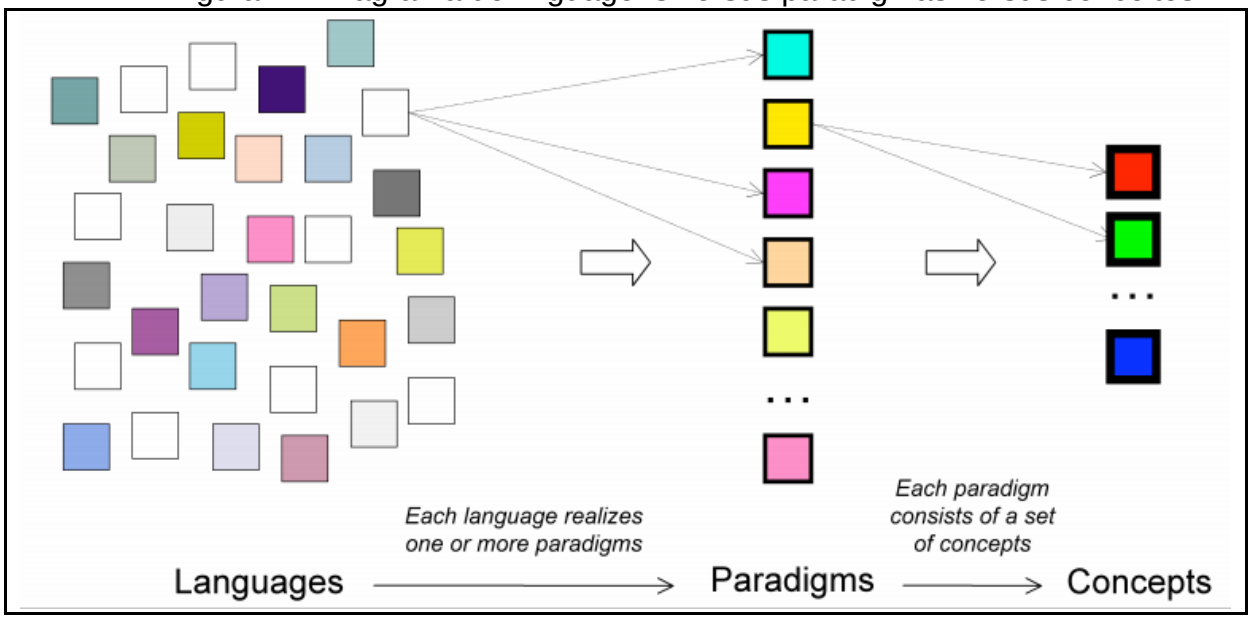
> *"Mais não é melhor (ou pior) do que menos, apenas diferente. " – Peter Van Roy*

Paradigmas de programação são definidos através de um conjunto de conceitos. Esses conceitos são organizados em um núcleo de linguagem que pode ser chamado de linguagem de *kernel* do paradigma (VAN ROY, 2009).

Segundo Van Roy (2009), quando são analisadas linguagens de programação versus paradigmas, é mais interessante focar-se nos paradigmas, visto que existem muito menos deles do que existem linguagens de programação. Tomando-se esse ponto de vista, linguagens como Java, Javascript, C#, Ruby e Python são basicamente idênticas, já que todas implementam o paradigma orientado a objetos.

Na Figura 1, é possível analisar uma abstração de como as linguagens de programação se comportam perante os paradigmas e seus conceitos. Paradigmas não são isolados uns dos outros; devido a uma mescla de conceitos que formam os paradigmas, uma linguagem de programação pode atender a um ou mais paradigmas, e por sua vez os paradigmas podem ser formados por um ou mais conceitos.

Figura 1 – Diagrama de linguagens versus paradigmas versus conceitos



Fonte: Peter Van Roy (2009).

2.2.1 Programação Procedural

Programação procedural, ou estruturada, é um paradigma de programação que defende que independentemente da complexidade do algoritmo a ser escrito ou a linguagem escolhida, o programa pode ser dividido em três partes interligadas: sequência, condição e repetição (BERTOL, 2012).

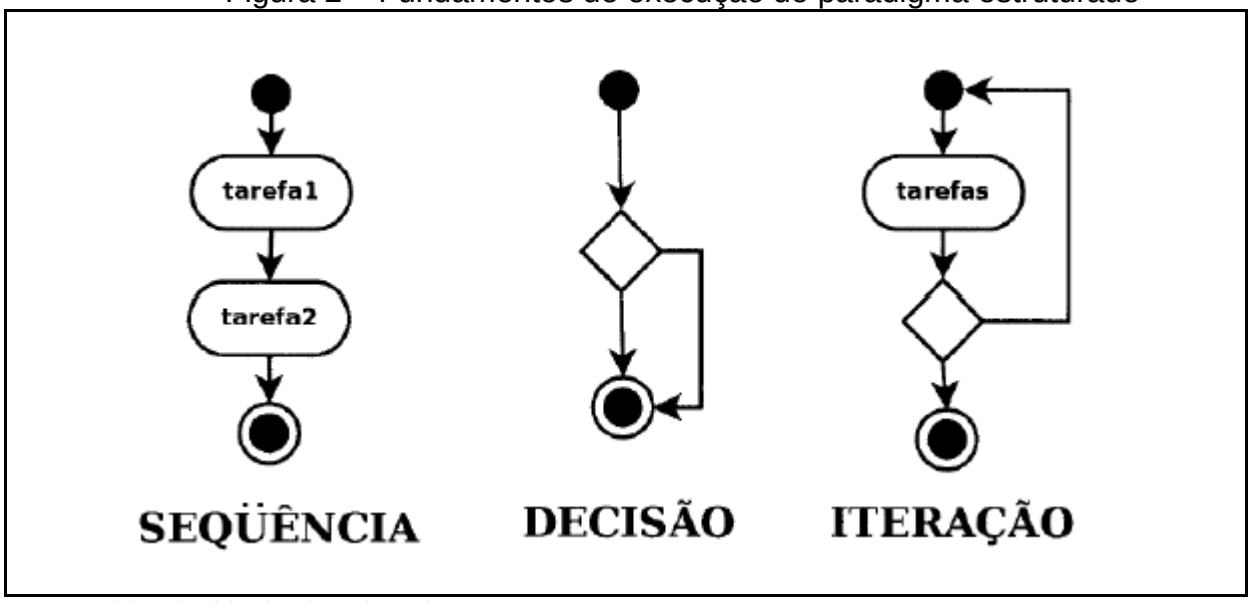
Através de uma forma de raciocino relativamente óbvia, o programa é executado sequencialmente, ou seja, quando a etapa X for concluída, a etapa Y poderá iniciar e assim por diante, enquanto existirem procedimentos a serem chamados (SANTOS, 2015).

Durante a interpretação do fluxo de execução, podem existir modificadores condicionais, os quais podem alterar o fluxo de execução de um algoritmo. Por exemplo: utilizando os modificadores condicionais *if* (se) e *else if* (senão), é possível chamar procedimentos únicos para cada linha de intepretação. Se a condição X for verdadeira, o fluxo é desviado para o procedimento A; caso falsa, o procedimento B é executado (SANTOS, 2015).

Finalmente, cada um dos procedimentos pode ser chamado infinitas vezes ou até que uma condição seja atingida. Utilizando os modificadores de repetição *while* (enquanto) e *for* (para), é possível executar um procedimento diversas vezes. Por exemplo, enquanto X for maior que 1, chamar procedimento B (SANTOS, 2015).

Na Figura 2, é possível verificar os fluxos apresentados, decisão e iteração correspondem à condição e repetição, apresentadas anteriormente.

Figura 2 – Fundamentos de execução do paradigma estruturado



Fonte: Erick Ednaldo de Lima (2013).

2.2.1.1 Procedimentos e Modularidade

Conforme o crescimento em complexidade e tamanho físico dos algoritmos aumenta exponencialmente, faz-se necessária uma forma de simplificar e tornar o programa mais legível. Isto é atingido através da divisão do programa em blocos menores, que podem ser chamados de subprogramas ou procedimento na programação estruturada (BERTOL, 2012).

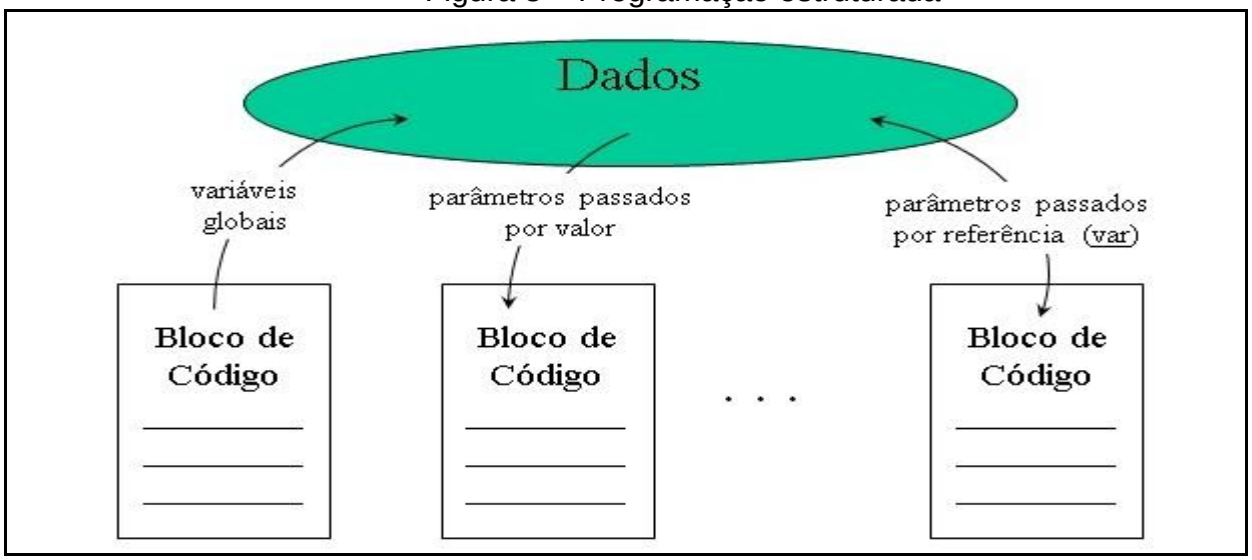
Procedimentos são pedaços da solução de software desenhados para atingir um objetivo. Esses procedimentos derivados normalmente estão vinculados ou subordinados a uma rotina maior. Uma característica deste tipo de recurso é que a solução é encerrada dentro dele mesmo (BERTOL, 2012).

Seguindo conceitos de Maquiavel, na programação de computadores um problema deve ser atacado em várias frontes (dividir para conquistar). Quando um problema é complexo demais para ser resolvido, ele é quebrado em soluções menores. Algumas dessas soluções terão retorno imediato através de procedimentos isolados, mas outras continuarão complexas demais para serem resolvidas, então devem ser novamente decompostas em soluções menores. Ao final do processo de

decomposição, o problema será resolvido através de diversas rotinas menores. Esse processo é chamado de modularidade (BERTOL, 2012).

Na Figura 3, podemos ver a interação dos blocos de códigos isolados com o contexto maior na forma de dados.

Figura 3 – Programação estruturada



Fonte: DevMedia (2015).

2.2.1.2 Vantagens e Desvantagens

Diversos autores, como Van Roy, defendem que não existe um paradigma ou linguagem de programação superior ao outro: cada um deles surgiu como uma proposta de solução a um problema. O estado ideal da linguagem seria aquele que suportasse múltiplos paradigmas, visto que nem sempre um programa resolve apenas um tipo de problema. Independente do fato de existir ou não superioridade, todos os paradigmas possuem um conjunto específico de vantagens e desvantagens que impulsionam a troca de paradigmas para resolver um problema.

Uma das principais vantagens da programação estruturada é o controle eficaz em cima do fluxo do que vai ser executado. O programador sabe o ponto de entrada e todas as suas paradas até um destino ser finalmente atingido; isso provém principalmente da boa utilização da modularidade (SANTANA SANTOS, 2015).

Além do controle, esse é um paradigma de fácil compreensão, o que o torna amplamente utilizado academicamente (MULLER, 2016).

A facilidade da linguagem em ser modularizada e o fluxo de execução é o que cria uma das suas principais desvantagens. Tanto Muller (2016) quanto Santana (2015) afirmam em seus artigos que o tratamento dos dados julgados juntamente com

o comportamento do programa, cria códigos de difícil interpretação e manutenção, dificultando a manutenibilidade do software e a reutilização de código-fonte.

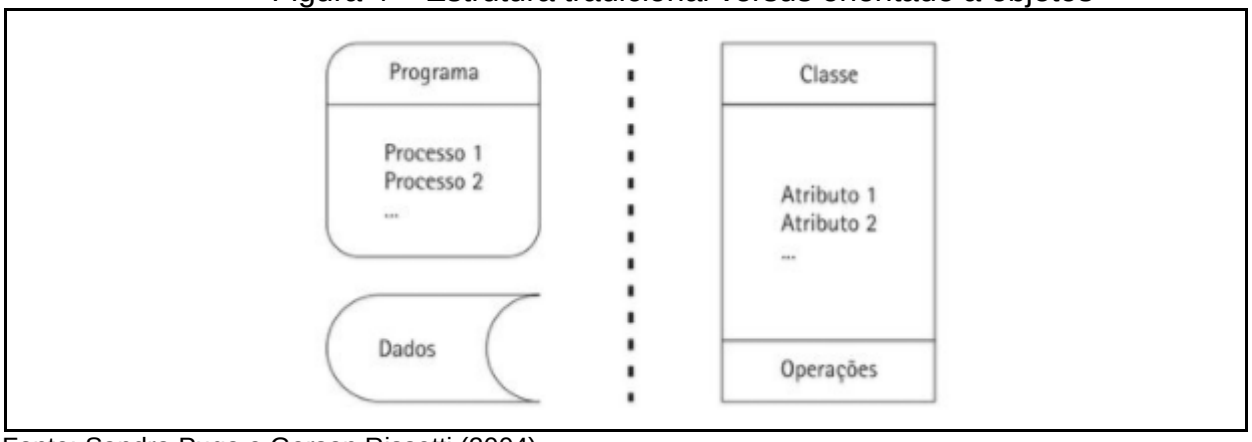
2.2.2 Programação Orientada a Objetos

Com o surgimento da programação orientada a objetos, o foco dos engenheiros de software foi alterado: não mais os programas eram tratados como um conjunto de procedimentos inter-relacionados para obter um resultado desejado, mas sim como um mundo real em que objetos existem, cada qual com suas características e atributos em constante interação (PUGA, RISSETTI, 2004).

O enfoque torna-se claro quando é analisado o panorama geral da natureza: os objetos já existem previamente à inserção de qualquer regra de negócio. Por exemplo: um cadastro simples de cliente, o objeto e seus atributos, nome, cidade, razão social, entre outros, podem já ser modelados sem ainda nem conceber métodos de inserção, alteração ou deleção (PUGA, RISSETTI, 2004).

Conforme pode ser percebido na Figura 4, o enfoque tradicional de programação (utilizado no paradigma procedural) se compara ao enfoque do objeto no paradigma OO.

Figura 4 – Estrutura tradicional versus orientado a objetos



Fonte: Sandra Puga e Gerson Rissetti (2004).

Existem diversos artefatos que separam a programação orientada a objetos da programação estruturada. Esta seção focará na explicação e exemplificação de: objetos, classes, herança e polimorfismo.

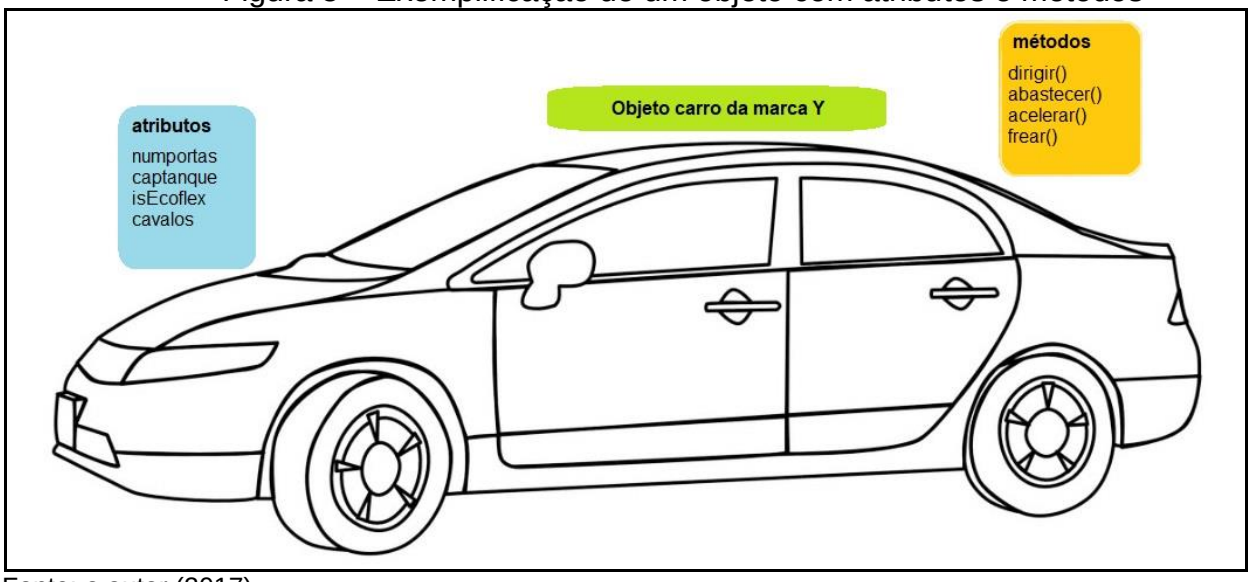
2.2.2.1 Objeto

Segundo Pugna e Rissetti (2004), um objeto é uma extensão de um conceito do mundo natural, classificado como: tangíveis, incidentais e objetos de interação. Não existe um sistema ou regra para o nível de abstração que uma modelagem de objeto deve atingir: tudo depende do desenvolvedor que está realizando a modelagem do software.

> *Abstração consiste em se concentrar nos aspectos essenciais, próprios, de uma entidade e em ignorar suas propriedades acidentais. Isso significa concentrar-se no que um objeto é e faz, antes de decidir como ele deve ser implementado em uma linguagem de programação. (PUGNA, RISSETTI, 2004, p.37)*

Como é possível perceber através da Figura 5, um “carro da marca Y” é um objeto da classe “carro”, que possui atributos e métodos específicos vinculados a ele. No caso abaixo, os atributos são: número de portas, capacidade do tanque, se o carro é ecoflex e o número de cavalos. Já os métodos passíveis de chamada no objeto são: dirigir, abastecer, acelerar e frear.

Figura 5 – Exemplificação de um objeto com atributos e métodos



Fonte: o autor (2017).

Os dados do objeto, ou seja, suas características, são chamados de **atributos.** Esses atributos são vinculados à classe do objeto que definem suas características. Essas características devem ser alteradas apenas através de funções específicas chamadas **métodos**, como por exemplo: caso invocado o método dirigir,

o atributo cavalos será utilizado para calcular a velocidade do carro (PUGA, RISSETTI, 2004).

2.2.2.2 Classe

Juntamente com objetos, as classes são um dos pilares fundamentais da orientação a objetos. Classes são basicamente uma coleção de objetos similares que podem ser definidos através de um conjunto de atributos e operações (métodos) de natureza similar (PUGA, RISSETTI, 2004). Como por exemplo, carro, ônibus e caminhão podem ser todos considerados classes herdadas de veículos, mas diferenciam-se em alguns atributos como número de rodas, combustível e número de assentos, podendo assim serem classificados como parte de uma mesma classe superior.

A ação de agrupar objetos de especificações similares em classes é chamada de **generalização**. Já a derivação do objeto, como por exemplo: carro, derivado de veículo, é chamada de **especialização** (PUGA, RISSETTI, 2004).

Alguns dos erros mais comuns na modelagem e desenvolvimento de software nascem de classes mal definidas e generalizações executadas incorretamente, o que resulta em duplicidades no código-fonte (DEITEL, 2011).

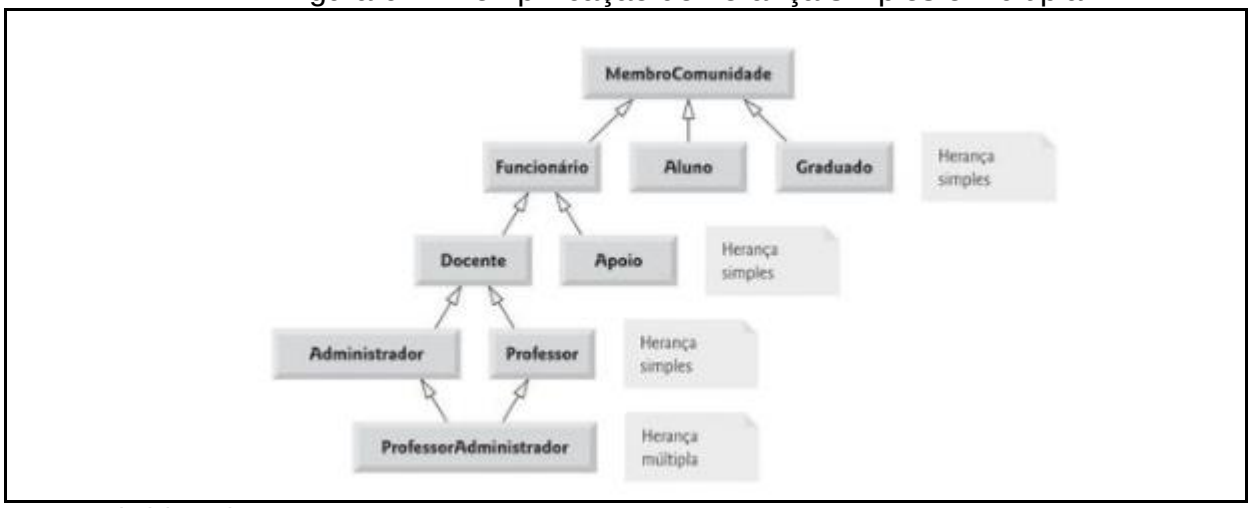
2.2.2.3 Herança

Um dos principais recursos da orientação a objetos é a capacidade de reutilizar fonte através de classes. Esta reutilização é feita através de heranças entre classes e subclasses, nas quais os atributos e métodos da classe original, (também conhecidas como superclasses em Java), são passadas a suas “filhas” através da herança. Normalmente, uma classe deve ter derivações apenas se, além das características (métodos e atributos) da classe pai, a subclasse deva ter especializações próprias (DEITEL, 2011).

A herança de uma classe para outra pode ser de forma singular, o que é chamado de herança simples, ou de forma a uma subclasse herdar atributos e métodos de uma ou mais classes, cuja nomenclatura é conhecida como herança múltipla (DEITEL, 2011).

É possível observar essa atribuição de heranças através da Figura 6, em que a classe professor-administrador é herdada de administrador e professor simultaneamente, causando com que a nova classe tenha os atributos e métodos combinados de duas superclasses.

Figura 6 – Exemplificação de herança simples e múltipla



Fonte: Deitel (2011).

2.2.2.4 Polimorfismo

Diretamente ligado com o conceito de herança, o polimorfismo é uma ferramenta utilizada para alterar o comportamento de subclasses em relação às superclasses. Muitas vezes, embora a subclasse deva executar o mesmo método da sua superclasse no mesmo momento, ela deve executar este método de uma forma diferente, podendo ser exemplificado através da relação veículo – carro/ônibus, quando chamado o método “ligar”, que está alocado na superclasse veículo. No caso do carro, um procedimento deve ser feito e, no ônibus, outro inteiramente diferente será executado. A isso é dado o nome de polimorfismo. (PUGA, RISSETTI, 2004).

2.2.2.5 Vantagens e Desvantagens

Tanto Muller (2016) quanto Santos (2015), afirmam que uma das principais vantagens da programação orientada a objetos é a reutilização e organização do código-fonte através da organização por classes e a utilização da herança, visto que, caso seja necessário realizar a manutenção em uma superclasse com diversas

classes derivadas, apenas um lugar deverá ser alterado, facilitando assim a manutenção do código-fonte e fornecendo um funcionamento singular ao sistema.

Essa mesma vantagem é foco da crítica ao paradigma de Scalfani (2016). Citando uma experiência pessoal, o autor descreve a dificuldade de reutilizar o código-fonte de um projeto anterior em seu novo projeto, problema derivado do fato de que diversas superclasses deveriam ser também consumidas para utilização da subclasse necessária para o novo projeto.

Uma citação de Joe Armstrong, criador do Earlang (uma linguagem de programação desenvolvida para suportar aplicações distribuídas e tolerantes a falhas), sumariza o ponto de Scalfani:

> *O problema com as linguagens orientadas a objeto é que elas têm todo um ambiente implícito que é carregado com ela. Você queria uma banana, mas o que você conseguiu foi um gorila segurando a banana e toda selva junto com ele. (Coders at Work, 2009, tradução nossa)*

2.3 PADRÕES DE ARQUITETURA DE SOFTWARE

Padrões de arquitetura nada mais são do que uma forma de compartilhar experiências de desenvolvimento a partir de outros desenvolvedores para disseminar o reuso de sistemas de software. O início da utilização de padrões foi marcado pela publicação de um livro por Gamma, em 1995. O objetivo do livro era fornecer padrões de projeto para linguagens orientadas a objeto, que haviam apenas recentemente começado a serem utilizadas no mercado (SOMMERVILLE, 2011).

Existem diversos padrões de arquitetura no mercado, como por exemplo: arquitetura de repositório, cliente-servidor e duto e filtro. Este trabalho será direcionado na arquitetura MVC (*Model-view-controller)* devido a sua estrutura de organização de código. Embora os conceitos do MVC sejam utilizados para o desenvolvimento do processo de conversão, não será realizada uma implantação cega: o padrão foi criado para ser utilizado com Smalltalk-76 e nem sempre é possível uma utilização completa, mas seus conceitos fundamentais, que serão apresentados a seguir, serão utilizados.

2.3.1 MVC

O padrão de arquitetura MVC foi inicialmente criado com o nome de *thing-model-view-editor* em 1979, desenvolvido por Trygve Reenskaug para ser utilizado em conjunto com o Smalltalk-76, uma linguagem de programação orientada a objetos e influenciadora de linguagens como Java e Objective-C. (GALLOWAY,2014).

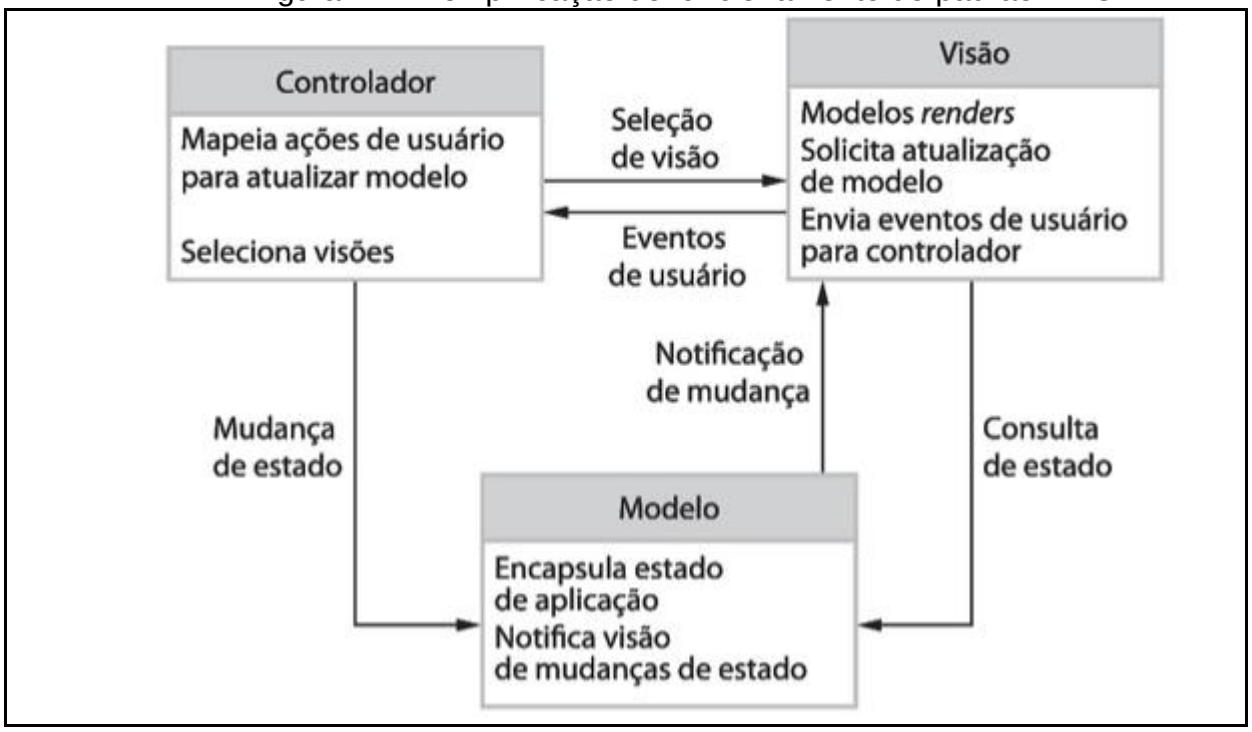
A fundamentação do MVC está em separar o que está sendo apresentado para o usuário através da GUI (*Graphical User Interface)* e a manipulação dos dados do sistema em si. Esta separação é feita através de três componentes com interações diretas entre eles. Os componentes são modelo (*model)*, visão (*view)* e controlador (*controller)* (SOMMERVILLE, 2011).

Trygve Reenskaug (2009), criador do MVC, afirma que o modelo é “[...] sobre pessoas e seus modelos mentais, não o padrão do observador”, ou seja, o *framework* existe para que a representação da informação seja separada da interação do usuário, o que pode ser chamado de MVC-U (*Model-View-Controller-User*).

O mais importante que deve ser considerado no padrão MVC é chamado de **apresentação separada**. O conceito fundamental é criar uma divisão clara entre objetos que modelam nossa percepção e os objetos que são apresentados na interface. Os objetos de domínio, que no MVC correspondem à camada de modelo, devem ser criados de uma forma que sejam autossuficientes e que, apesar da existência de uma interface, funcionem normalmente. Este conceito é como o *Unix*, que permite que várias aplicações sejam manipuladas através de interfaces gráficas ou linhas de comando (FOWLER, 2006).

Podemos verificar esta interação em camadas através da Figura 7, onde as camadas estão interagindo diretamente para a apresentação de dados para o usuário.

Figura 7 – Exemplificação do funcionamento do padrão MVC



Fonte: Sommerville (2011).

2.3.1.1 Modelo

A camada de modelo é a responsável por encapsular os dados e as funcionalidades do software. O componente age como um gerente, interceptando todas as operações associadas à manipulação destes dados. Também é chamada esta camada de modelo de domínio; normalmente o modelo não tem noção nenhuma das interações do usuário com a interface. (FOWLER, 2006).

2.3.1.2 Visão

O componente de visão é responsável por exibir as informações e decidir como estas informações serão exibidas ao usuário (SOMMERVILLE, 2011).

2.3.1.3 Controlador

O componente do controlador é responsável por gerenciar a interação do usuário com a interface, como por exemplo: clicar em teclas, mouse ou interagir com os elementos visíveis e repassar as informações para a visão e o modelo para que os dados sejam persistidos (SOMMERVILLE, 2011).

2.3.1.4 Utilização e Derivações

O MVC, como um dos pioneiros em padrões de arquitetura para software orientado a objetos, possui diversas derivações de sua aplicação. Provavelmente, a mais relevante ao mercado é a ASP.NET MVC: a base continua sendo utilizada através da divisão em três componentes, mas como esta derivação é focada em desenvolvimento web, os componentes também assumem tarefas diferentes. Do modelo, virá uma camada de acesso a dados, utilizando ferramentas como NHibernate. A visão torna-se um *template* HTML gerado dinamicamente e o controlador é o que sofre menos modificações da sua criação original: continua controlando interação do usuário com a interface (GALLOWAY, 2014).

Embora existam diversas aplicações do padrão, o fluxo de funcionamento do MVC geralmente continua o mesmo. Segue o fluxo apresentado por Jonathan Lamim (2012):

a) Interação do usuário com a interface;
b) Controlador manipula o evento através de uma rotina pré-estabelecida;
c) Controlador acessa o modelo, manipulando dados conforme a rotina solicitou, alterando, inserido ou excluindo dados.
d) Algumas interfaces mostram para o usuário o que será alterado, solicitando uma confirmação da ação. Esses dados utilizados pela visão são obtidos diretamente do modelo, embora a camada de modelo não tenha conhecimento da existência da camada de visão.
e) A interface aguarda novas interações em que o ciclo se repetirá.

Sommerville (2011) indica a utilização do padrão MVC quando os requisitos de software para interação e apresentação de dados sejam desconhecidos em um futuro, fornecendo um modelo que terá seu funcionamento da mesma forma independentemente da interface utilizada.

Embora focada em sistemas web, o que deve ser levado deste padrão é sua organização, chamada de apresentação separada por Fowler (2006), anteriormente. Desenvolver um sistema utilizando uma derivação de MVC pode fornecer diversas vantagens, como por exemplo permitir que os dados sejam alterados de qualquer lugar do sistema de uma forma padrão. Após a atualização, todas as outras representações terão os dados atualizados (SOMMERVILLE,2011).

2.4 CONSIDERAÇÕES FINAIS

Embora a fundamentação exposta sobre as linguagens de programação possa parecer primária, é de fundamental importância para o entendimento e aplicação do processo de conversão.

Devido ao aumento exponencial de complexidade ao organizar e desenvolver um código-fonte em classes e decompor os antigos procedimentos do paradigma procedural em métodos para estas classes, é necessária uma forte base teórica para que os costumes carregados de paradigmas anteriores não tornem o sistema refatorado em uma versão mais complexa e de maior dificuldade de manutenção do que o sistema anterior.

O conhecimento de linguagens de programação é a base para a compreensão e aplicação de métodos e processos de engenharia de software, abordados a partir do próximo capítulo.

3 ENGENHARIA DE SOFTWARE

O campo da engenharia de software é composto por vários fundamentos e conceitos importantes para o desenvolvimento de software. Desde o estabelecimento de conceitos básicos de software até padrões avançados de arquitetura de software, o objetivo da engenharia de software é fornecer produtos de software de alta qualidade atendendo prazos e custos propostos (MEDEIROS, 2013).

> *"Engenharia de Software é o estabelecimento e o emprego de sólidos princípios de engenharia de modo a obter software de maneira econômica, que seja confiável e funcione de forma eficiente em máquinas reais" (Fritz Bauer, apud MEDEIROS, 2013)*

3.1 METODOLOGIAS ÁGEIS DE DESENVOLVIMENTO DE SOFTWARE

A partir de derivações dos processos de produção instituídos pela Toyota no final da Segunda Guerra Mundial, a indústria de software acreditava que, para um produto ser desenvolvido com qualidade, era necessário instituir um planejamento extremamente detalhado e bem documentado. A justificativa da necessidade de tanta documentação era atribuída à vastidão dos projetos e variedade da equipe de desenvolvimento. Para projetos maiores, esse modelo funcionava relativamente bem, mas para equipes menores com produtos pequenos o tempo gasto com planejamento era desnecessário (FOGGETTI, 2015).

Em 2001, como uma contraproposta ao modelo tradicional de desenvolver softwares, nasceu o gerenciamento ágil de projetos. O ponto principal de mudança era a ênfase na execução de tarefas e não mais no planejamento. Através de uma reunião com 17 especialistas em processos de desenvolvimento, criou-se a Aliança Ágil, cujo produto, o Manifesto Ágil, criou as linhas a serem seguidas para a modelagem de processo (FOGGETTI, 2015).

Os pilares da metodologia ágil são provocar a interação entre indivíduos e não mais entre documentos, focar no software em execução e não mais em documentação detalhada, considerar o cliente como parceiro e não mais apenas um contrato, e pensar no processo de forma ágil e não mais em planejamentos intermináveis. (FOGGETTI, 2015).

É importante salientar que o manifesto ágil não descarta os processos e artefatos dos métodos de desenvolvimento tradicional, mas atribui sua importância

como secundária no quadro de prioridades, atrás da interação com o indivíduo e de entregas ágeis (SOARES, 2004)

Embora o processo de conversão aqui proposto não seja baseado inteiramente na metodologia ágil, alguns artefatos serão extraídos dela, mais especificamente da aplicação do *Scrum*, através da utilização da construção de histórias e *backlog*. Na seção abaixo, são detalhadas funcionalidades e aplicações desse *framework*.

3.1.1 Scrum

O *Scrum* foi criado em 2002 através dos conhecimentos de Ken Schwaber e Jeff Sutherland. É uma metodologia direcionada a desenvolvimentos em que os requisitos são incertos e estão em constante mudança. É característico do *Scrum* a ausência de técnicas específicas para desenvolvimento de software, apresentando apenas um conjunto de práticas indicadas para gestão de projetos (FOGGETTI, 2015).

A parte fundamental da implementação do *Scrum* é a repetição de processos através de *Sprints*, na qual cada novo fim de ciclo deve estar bem escrito e testado (SCHWABER,2002).

3.1.1.1 Artefatos

O principal artefato que será utilizado do *Scrum* é o que é chamado de *Product Backlog*. Este artefato é uma lista de tudo que deve ser feito em questão de funcionalidades juntamente com uma estimativa. Este artefato é mutável, ou seja, caso seja identificada uma nova necessidade para o produto final, ele pode ser modificado para que esta nova funcionalidade seja atendida (SCHWABER,2002).

As necessidades são inseridas no *backlog* através de um *template* chamado de história de usuário *(user story)*. Estas histórias são funcionalidades que o usuário gostaria de realizar no sistema, através do formato: “Como <um sujeito> eu gostaria de <ação>”. Por exemplo: como um fornecedor “eu” gostaria de fornecer a minha lista de produtos por anexo (LIBARDI; BARBOSA, 2010).

A história de usuário, embora simples, pode acompanhar um detalhamento maior através de um texto ou gráfico de como algo (ator) irá interagir com o sistema.

O ator não necessariamente precisa ser o usuário final, mas o objeto de interação com o sistema (FOGGETTI, 2015).

3.2 FERRAMENTAS DE ENGENHARIA DE SOFTWARE

Todo o conteúdo apresentado até agora faz parte, quase que na sua integridade, de artefatos e ferramentas da engenharia de software. O motivo da divisão desta seção é para que sejam apresentadas ferramentas da engenharia de software que serão utilizadas futuramente para o desenvolvimento do processo de conversão. Os principais pontos que serão apresentados são: requisitos de software, extraídos diretamente do modelo de engenharia direcionado a requisitos, e uma explanação sobre engenharia reversa e reengenharia.

3.2.1 Engenharia Reversa

É desconhecida a data específica da concepção do conceito de engenharia reversa, mas é de conhecimento comum que sua aplicação inicial foi bélica, na espionagem de inimigos para obter vantagem militar (HAUTSCH, 2009).

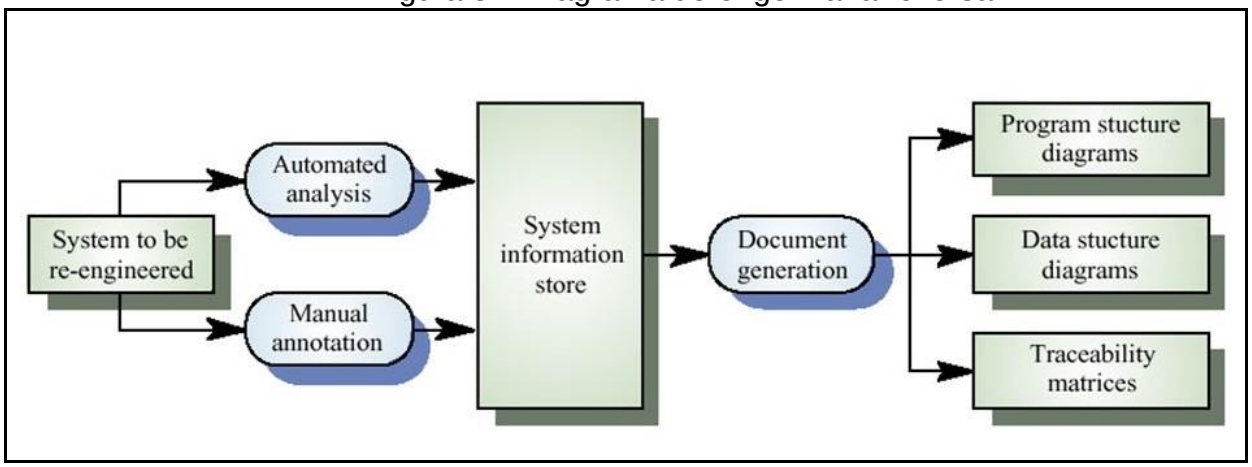
O funcionamento da engenharia reversa é analisar e estudar algo: um processador, uma peça de hardware ou um produto de software. Através da verificação do comportamento e comandos, são analisados os métodos de fabricação e de como podem ser melhorados e, em alguns casos, copiados (HAUTSCH, 2009).

Na Figura 8, é apresentado um fluxo de engenharia reversa por Sommerville (2011). O fluxo consiste em:

a) Entrada do software para sofrer a reengenharia;
b) Análise automatizada através de software ou análise manual;
c) Armazenamento das informações do sistema;
d) Geração da documentação;
e) Geração de estrutura de diagramas dos programas, diagramas de estrutura de dados e matrizes de rastreabilidade;

Conforme apresentado, o foco da engenharia reversa é a análise de informações: nada é reescrito, mas os dados são utilizados para prosseguir com o processo durante a reengenharia.

Figura 8 – Diagrama de engenharia reversa



Fonte: Sommerville (2011).

3.2.2 Reengenharia

Quando um sistema de informação está enraizado em uma empresa há anos, o software tende a perder sua legibilidade, seja por manutenção buscando otimizar desempenho ou por mudanças radicais de requisitos feitas de forma incorreta, causando danos permanente à estrutura do programa (SOMMERVILLE, 2011).

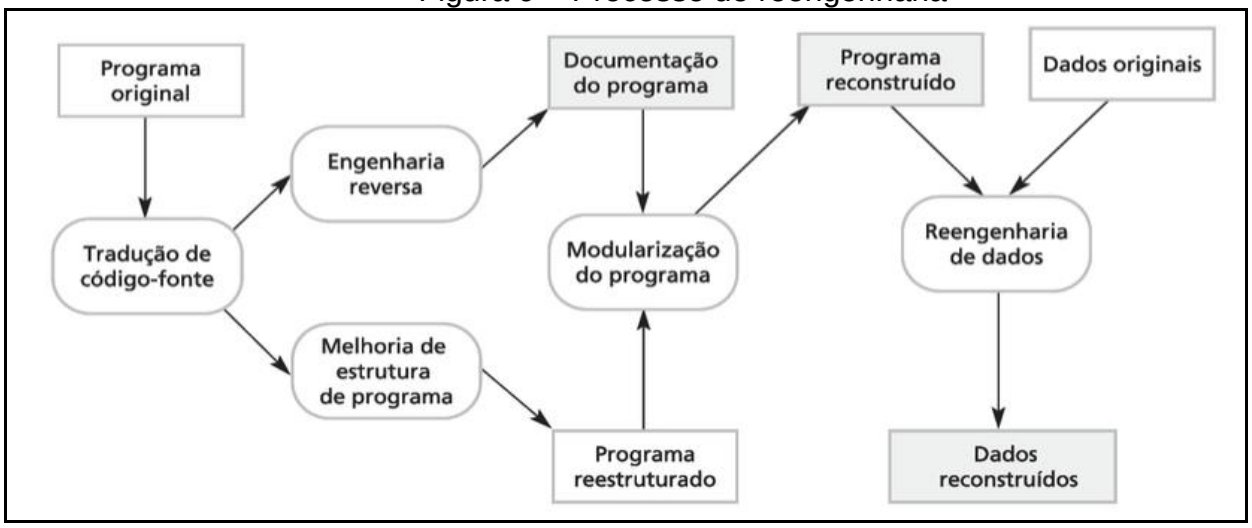
Para facilitar a manutenção destes sistemas legados, é necessário que eles passem por uma reengenharia, com o intuito de otimizar a estrutura através da melhoria de legibilidade e estrutura. Este processo pode envolver diversas modificações como: refatoração e nova linguagem ou geração de documentação para o software (SOMMERVILLE, 2011).

No capítulo sobre reengenharia de software, Sommerville aponta dois benefícios consideráveis para apoiar o processo de reengenharia contrário à alternativa que seria a substituição do produto de software. Os fatores que apoiam a reengenharia são: risco reduzido, levando-se em conta a alta complexidade de desenvolver um novo produto para substituir um software crítico aos negócios da empresa, devido ao fato de que durante este processo podem ocorrer perda de funcionalidades ou comportamentos não esperados. E o custo é consideravelmente menor, em parte pelo número de pessoas que serão necessárias para o processo e em parte pelos custos de manutenção e perdas de negócios em caso de falhas no novo software.

Na Figura 9, Sommerville sugere um processo de reengenharia no qual a entrada é um software legado e a saída é um novo software mais legível. O autor

afirma que nem todos os passos podem ser necessários, dependendo da necessidade de troca de ambiente de desenvolvimento ou esquemas de banco de dados.

Figura 9 – Processo de reengenharia



Fonte: Sommerville (2011).

3.2.3 Requisitos de Software

Requisitos de software, assunto fundamental de estudo da engenharia de requisitos, são: *"[...] as descrições do que o sistema deve fazer, os serviços que oferecem e as restrições a seu funcionamento".* (SOMMERVILLE, 2011).

Requisitos podem ser separados em duas categorias que requerem um maior ou menor nível de abstração: requisitos de usuário, que são basicamente o que o software deve atender a nível de utilizador; e requisitos de sistema, que é uma descrição detalhada do que o sistema deve fazer sem que necessariamente o usuário tenha conhecimento de sua existência (SOMMERVILLE, 2011).

A classificação tradicional dos requisitos é funcional e não funcional. Embora pareçam isolados, os requisitos são normalmente ligados. Ou seja, um requisito não-funcional pode gerar a necessidade de um requisito funcional (SOMMERVILLE, 2011).

3.2.3.1 Requisitos Funcionais

É considerado como requisito funcional tudo que um sistema deve fazer. Existem variações dependendo do tipo de produto a ser desenvolvido e quem será o público alvo. Quando direcionado a usuários, os programas devem ter uma abstração maior para que seja possível o entendimento do usuário, e quando escritos para o

sistema, devem contemplar todas entradas e saídas possíveis incluindo suas exceções (SOMMERVILLE, 2011).

É de fundamental importância que os requisitos do software sejam bem definidos desde sua concepção. Se for permitida a interpretação ambígua de algum requisito, o produto está em risco de sofrer mudanças acarretando atrasos e custos de implementação.

3.2.3.2 Requisitos Não Funcionais

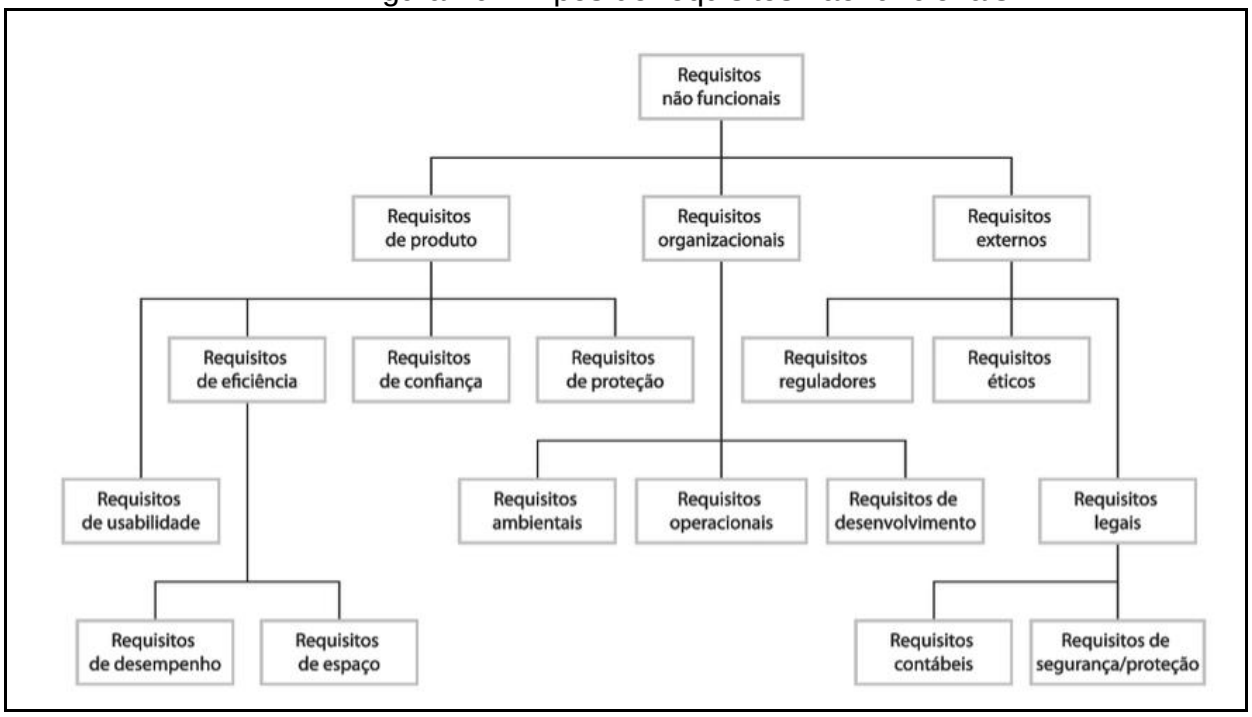
São classificados como requisitos não funcionais aqueles que não estarão diretamente relacionados com o que o produto ou serviço oferecerá aos usuários. Eles podem fazer referências a conceitos abstratos como segurança, confiabilidade e tempo de resposta. Normalmente, este tipo de requisito não é limitado a módulos ou pedaços de software, mas sim ao sistema como um todo (SOMMERVILLE, 2011).

É importante salientar que os requisitos não funcionais, embora sejam mais fáceis de serem ignorados, podem inutilizar um sistema por completo, devido ao fato de que eles estão interligados com os requisitos funcionais de um sistema. Por exemplo, o requisito não funcional de um sistema é ser seguro: esse simples requisito irá criar diversas contrapartes funcionais – será necessária funcionalidade de *login,* criptografia para as senhas, formas de recuperação de senha, verificação em dois passos, etc. (SOMMERVILLE, 2011).

Normalmente, o surgimento de requisitos não funcionais é atribuído à necessidade dos usuários. Estes requisitos podem surgir de diversas fontes e, em muitos casos, devido a restrições orçamentais, não é possível atingir todos os requisitos necessários e uma priorização deve ser feita (SOMMERVILLE, 2011).

Através da análise de um sistema médico MHC-PMS, Sommerville explanou as diversas fontes dos requisitos não funcionais de um sistema, conforme pode ser percebido na Figura 10.

Figura 10 – Tipos de requisitos não funcionais



Fonte: Sommerville (2011).

3.3 CONSIDERAÇÕES FINAIS

O estabelecimento de padrões e técnicas para extrações de requisitos é fundamental para a aplicação do processo de conversão, visto que todo o trabalho a ser realizado no software partirá de requisitos e casos de uso em funcionamento no sistema.

4 TRABALHOS RELACIONADOS

No decorrer da pesquisa e desenvolvimento desta monografia, foram encontrados alguns trabalhos que retroalimentaram o processo. Embora o assunto de alguma forma careça de artigos e publicações, duas destas foram importantes para a concepção e desenvolvimento do processo apresentado.

O primeiro deles é um artigo chamado “Um processo de transformação de arquitetura de sistema legado funcional para orientado a objetos, direcionado por indicadores de qualidade”, de autoria de Wagner Leal dos Santos, publicado no 3º Congresso Internacional de Gestão da Tecnologia e Sistemas de Informação em 2006. Durante o decorrer do artigo, o autor descreve um processo focado na transformação de um sistema legado funcional desenvolvido em ambiente funcional apoiado por indicadores de qualidade baseado nas normas da ISO 9001, ISO 9000-3 e NBR ISO/IEC 14598 (SANTOS, 2007).

A principal contribuição do artigo, é fornecer uma base que pode ser trabalhada para desenvolver novos processos de conversão, devido ao fato de que o autor não foca o trabalho no estabelecimento de uma arquitetura para desenvolvimento, levantamento de requisitos e boas práticas para realizar o processo de reengenharia. Através do estabelecimento desta base, foi possível desenvolver o processo apresentado nesta monografia, com o estabelecimento de um padrão de arquitetura sugerido e removido o foco dos indicadores de qualidade para o estabelecimento de requisitos.

Com a conclusão do artigo, Santos acredita que a migração dos sistemas legados é uma alternativa a ser considerada pelas empresas, visto que a maioria dos artigos abordam conversão de código e não a obtenção de uma arquitetura.

A segunda publicação que norteou o desenvolvimento dos passos do processo de conversão foi o capítulo 9.3.2 “Reengenharia de Software”, do livro “Engenharia de Software”, escrito por Ian Sommerville, autor britânico considerado referência no campo de engenharia de software (SOMMERVILLE, 2011).

O capítulo é focado na otimização da estrutura e inteligibilidade de sistemas legados. Sommerville levanta dois pontos fundamentais para o apoio do processo de reengenharia em relação à alternativa de criar um novo software, o primeiro é a redução do risco de problemas e o segundo a redução dos custos com o desenvolvimento.

Embora o processo exposto por Sommerville tenha passos que envolvam a interoperabilidade do sistema legado com o sistema que sofreu a reengenharia, o processo apresentado nesta monografia não se utiliza destes passos, visto que a premissa é uma substituição completa e não gradual.

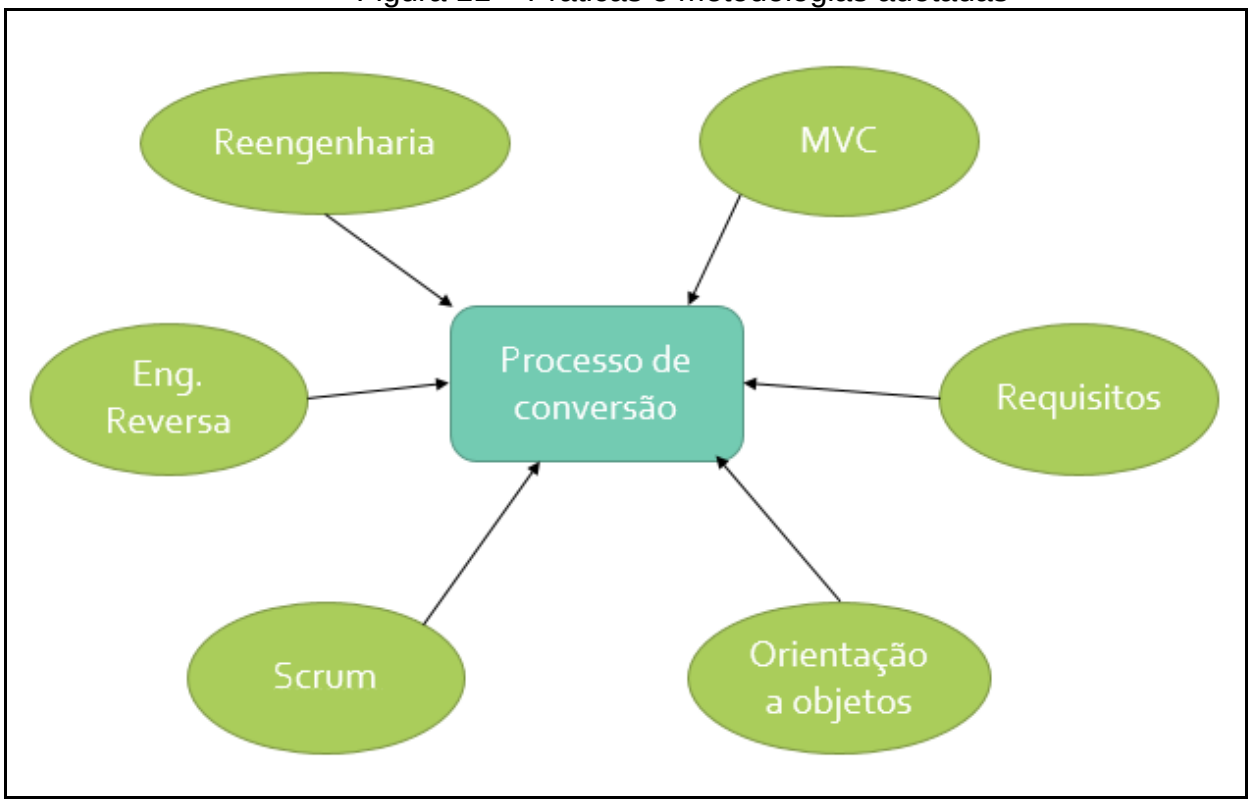
A autor conclui o capítulo apontando as limitações de um processo de reengenharia, que podem estar ligadas ao custo exacerbado ou à inviabilidade técnica do software em se submeter a tal mudança, seja por meios automáticos ou manuais.

5 PROCESSO DE CONVERSÃO DE SISTEMAS LEGADOS

No decorrer deste capítulo, é apresentado o processo de conversão de sistemas legados procedurais para orientado a objetos, direcionado pela arquitetura MVC. O desenvolvimento do processo foi baseado em parte no processo de *Sprint* do Scrum, e em parte no processo de reengenharia proposto por Sommerville. Algumas adaptações foram feitas em fases que exigiam documentação e integração entre o sistema legado e o reconstruído, além de algumas adições ao processo.

Os pilares que fomentaram o desenvolvimento do processo foram elencados para resolução de diversos problemas, como estabelecimento de um padrão de arquitetura, paradigma de desenvolvimento escalável e uma metodologia de trabalho ágil, que podem ser vistos na figura 11.

Figura 11 – Práticas e metodologias adotadas



Fonte: o autor (2017).

O processo está estruturado em dez passos que serão explicados nas seções que seguem. Para exemplificação dos passos a serem tomados será utilizado um *mockup* – modelo em escala para fins de demonstração de uma funcionalidade

simples de um sistema de movimentação de estoque que fará entrada e saída de produtos apenas.

Para as seções que envolvem alguma forma de codificação, o processo será exemplificado em *Visual Dataflex (VDF)* – ambiente de desenvolvimento direcionado à criação rápida de aplicações. Sua sintaxe permite uma maior acessibilidade à interpretação do código-fonte devido ao fato de que as palavras-reservadas são uma transcrição mais literal da operação sendo realizada, ou seja, de mais alto nível. Enquanto no Java, para uma variável receber valor, é necessário realizar a seguinte operação: “int var1 = 10”, em VDF a mesma operação é escrita da seguinte forma: “move 10 to var1”.

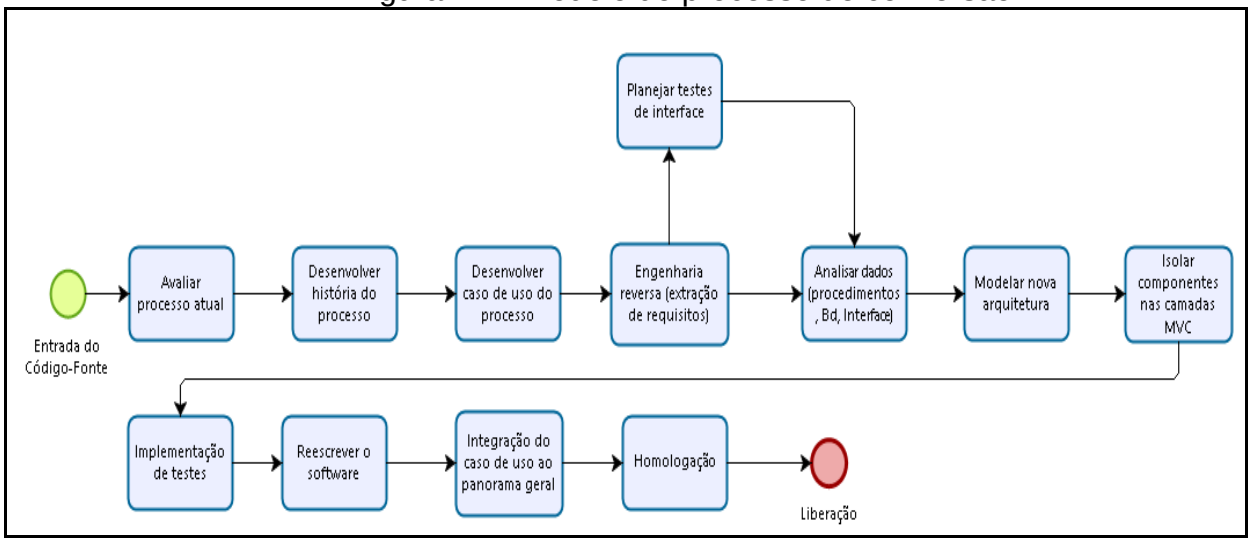
É importante ressaltar que não há dependência da linguagem à aplicação do processo de conversão: a mesma foi escolhida devido a sua fácil interpretação sintática e capacidade de suportar os dois paradigmas: procedural e orientado a objetos.

Embora existam ferramentas que se propõem a realizar a migração automática de programação procedural para orientada a objetos, como por exemplo a empresa mtSystems, que realiza a tradução de código desenvolvido na linguagem C para JAVA, as mesmas não serão utilizadas devido a dois motivos: o primeiro é que nem todas as linguagens de destino são compatíveis, existem ferramentas para Java por exemplo, mas outras linguagens orientadas a objeto não são suportadas. O segundo motivo é que foge à motivação da aplicação do processo, o objetivo é otimizar a arquitetura do software através da migração para um paradigma orientado a objetos, melhorando assim a manutenibilidade e estendendo a vida útil do software legado.

Cada um dos passos está ligado de forma que o produto de uma atividade fomente o desenvolvimento da próxima. O processo tem como entrada o código-fonte, esta entrada seria de uma funcionalidade previamente definida pelo indivíduo ou instituição. Com isso, é realizada uma avaliação da funcionalidade e verificada a viabilidade de aplicação do processo e perguntas a serem respondidas para a continuidade do processo. Esta atividade irá fornecer uma base para que seja iniciado o trabalho na funcionalidade, este trabalho é iniciado através do entendimento e criação das histórias de usuário para a funcionalidade. As histórias criadas irão diretamente fomentar a criação dos casos de uso, que por sua vez irão orientar, de forma em que seja buscado no código-fonte as mecânicas responsáveis pela

realização do caso de uso, a extração do código-fonte, gerando requisitos de desenvolvimento. A sequência extraída será objeto de trabalho para a análise dos procedimentos e estruturas. Neste ponto, é possível verificar o encadeamento de procedimentos responsáveis pelas execuções dos casos de uso desenvolvidos, e a partir disso, iniciar o processo de decomposição de procedimentos e modelagem de classes. Com o modelo de classes definido, o mesmo é adaptado para atender os requisitos do padrão de arquitetura MVC, assim permitindo o início do desenvolvimento. O desenvolvimento do software será com base nos modelos definidos pelos passos anteriores, utilizando-se de boas práticas de desenvolvimento de software e a implementação dos testes de unidade. Ao final do processo, os testes realizados durante o levantamento da engenharia, serão aplicados novamente e a funcionalidade encaminhada para homologação. Com a aprovação da funcionalidade a mesma é liberada para os clientes. Esta interação pode ser vista na figura 12, que apresenta um panorama geral do modelo do processo de conversão proposto.

Figura 12 – Modelo do processo de conversão



Fonte: o autor (2017).

5.1 APLICABILIDADE

Quando um sistema legado tem a necessidade de ser reescrito em um novo paradigma, algumas considerações devem ser feitas. Abaixo são listadas algumas destas considerações antes da aplicação do processo:

a) O ambiente de desenvolvimento suporta ou não diversos paradigmas? Caso não, para qual linguagem será migrado?
b) O banco de dados será compatível com o novo ambiente de desenvolvimento? Caso não, será reestruturado?
c) Será possível manter a aplicação legada em funcionamento simultâneo até o fim a entrega do software refatorado?

As perguntas feitas acima não impõem restrições à aplicação, mas são questões importantes a serem respondidas antes da aplicação do processo. A única limitação imposta é em relação aos paradigmas originário e destinatário. O processo de conversão de programas procedurais para orientados a objetos é possível pois os procedimentos irão se tornar métodos de classes tornando-os compatíveis, mas paradigmas funcionais não são compatíveis com orientados a objetos, por exemplo, como apontado por Sommerville (2011).

É de fundamental importância que, antes da aplicação do processo, todas as perguntas feitas acima tenham sido respondidas. Será assumido que, para o decorrer do processo, uma linguagem que suporte a orientação a objetos tenha sido escolhida, uma definição do banco de dados tenha sido feita e a decisão de entregas realizada.

5.2 AVALIAÇÃO DO PROCESSO ATUAL

A avaliação do processo atual tem como objetivo fundamental a análise da viabilidade do processo de migração ou a permanência do software legado. Durante este processo, deve ser avaliado o número de envolvidos e o custo monetário e de tempo para ser aplicado ao processo de conversão. Caso o mesmo ambiente de desenvolvimento seja mantido, deverá se decidir se ao final de cada entrega o módulo refatorado substituirá o legado. Caso o sistema esteja organizado de forma modular, deverá ser decidida a ordem da aplicação do processo de reengenharia dos módulos, priorizando pela relação custo por tempo ou nível de impacto da funcionalidade no produto.

Caso a empresa possua um registro de reclamações ou solicitações realizadas, durante esta etapa é importante que sejam verificadas as necessidades não atendidas e os problemas apresentados pelos programas, para que seja avaliada a possibilidade de integração dos mesmos durante a reescrita do software.

Ao final deste processo de avaliação, espera-se que o artefato do trabalho produzido seja um documento que possua um estudo de viabilidade de aplicação do processo de conversão, tempo previsto para realização e quais partes, ou módulos, deverão ser convertidos primeiro.

5.3 DESENVOLVIMENTO DE HISTÓRIAS

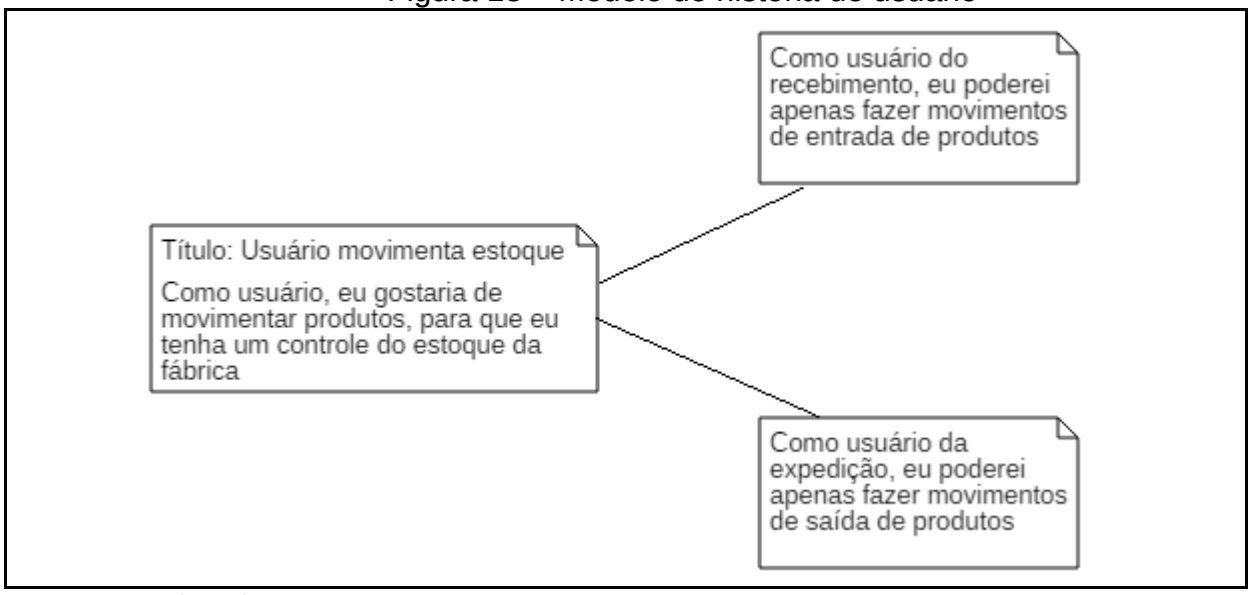
Durante o desenvolvimento de histórias, será assumido que a funcionalidade já tenha sido escolhida anteriormente. Neste momento, serão decompostas as histórias que deverão ser realizadas por funcionalidades, caso aplique-se, organizada por módulos.

Existem diversos formatos e sintaxes para a escrita de uma história de usuário, mas é importante que sejam definidos os **atores** e o que eles querem **fazer** com a funcionalidade e **por que**.

O ponto fundamental deste processo é estabelecer um “ponto de recuperação”, ou seja, o software já provê ao usuário aquela funcionalidade e deverá continuar funcionando da mesma forma após a conclusão do processo.

Na Figura 13, é exemplificada a construção de uma história de usuário com duas histórias dependentes. O ator está definido como “usuário” e a ação é “movimentar produtos” e o por que é “para ter um controle do estoque”.

Figura 13 – Modelo de história de usuário



Fonte: o autor (2017).

Embora normalmente uma história de uso seja acompanhada de um artefato chamado testes de aceitação – que definem os limites que uma história deverá atingir –, a principal funcionalidade é que seja possível identificar quando a história está concluída (COHN, 2004). Os testes foram realocados para que sejam feitos durante a extração dos requisitos no processo de engenharia reversa, para que seja possível ter um planejamento de testes mais assertivo.

A principal utilidade das histórias de usuário na aplicação do processo é definir um parâmetro de comparação e criar um relacionamento de histórias que moldam o sistema.

5.3.1 Considerações sobre casos de uso, histórias e requisitos

Apesar da existência de similaridades entre casos de uso e histórias de usuário, existem diferenças notáveis entre os dois produtos. Ambas partilham dos mesmos dados para sua composição, mas devido à falta de detalhamento que a história de usuário fornece para o panorama geral, não é sempre possível realizar a imersão necessária. A limitação da história de usuário está em seu tamanho, que normalmente é mantida concisa para facilitar entendimento. Ambas técnicas diferem de requisitos, os requisitos são orientados a fornecer informações sobre algo que deve ser realizado, já histórias focam em objetivos a serem cumpridos (LONGO e SILVA, 2014).

Da mesma forma, os três métodos serão utilizados para fins diferentes no processo de conversão, a história será utilizada para fornecer uma base estável para desenvolvimento de um fluxo de caso de uso, que por sua vez, irão estabelecer subsídios para a extração dos requisitos do código-fonte.

5.4 DESENVOLVIMENTO DE CASOS DE USO

A principal finalidade de um diagrama de caso de uso, é através de textos, descrições ou interações, contar a interação de um usuário com o sistema. A personificação de usuários e funcionalidades são realizadas através de distintos papéis (PRESSMAN, 2016).

Para desenvolver um caso de uso completo, é necessário um conjunto de passos. O primeiro é o estabelecimento de "atores", que podem ser softwares, pessoas ou dispositivos que realizem um papel ativo na história. Após, devem ser

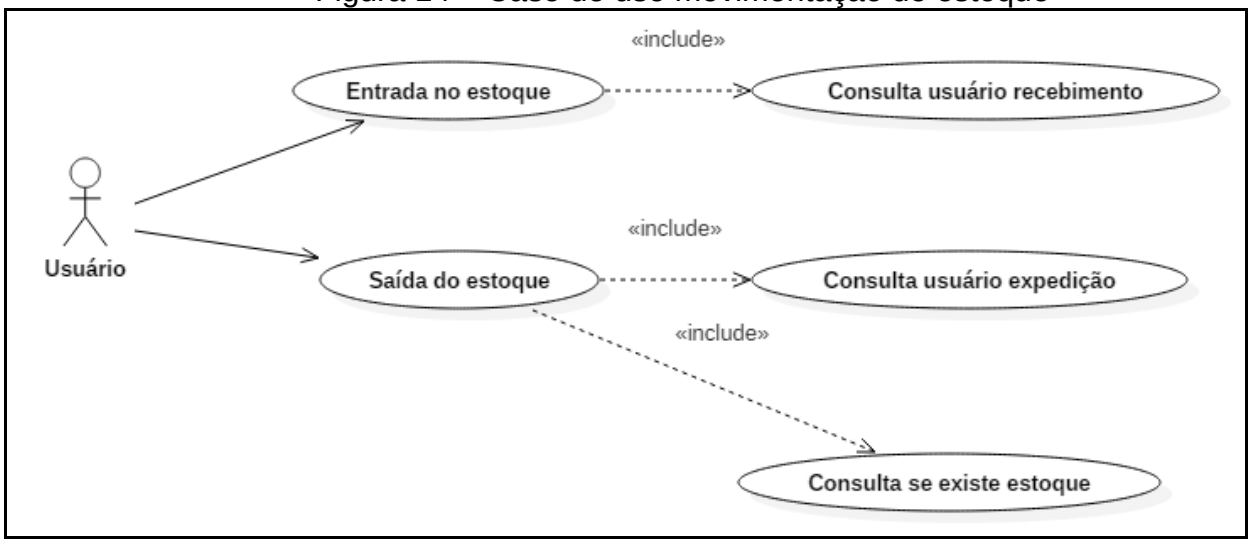
estabelecidos que tarefas os atores irão realizar e como estas tarefas interagem. Por fim, definir qual é a meta do ator durante o período de interação com o caso, ou seja, o que é o produto final da sequência de interações com o caso (PRESSMAN, 2016).

Durante o desenvolvimento do caso de uso, serão utilizadas as histórias definidas durante o processo de desenvolvimento de histórias descrito na seção 5.3. É recomendado que um analista de sistema, com extenso conhecimento do produto, seja consultado para geração tanto dos casos de uso quanto das histórias, visto que o acesso ao código-fonte ainda não será necessário.

Existem dois motivos principais para a utilização de casos de uso no processo de conversão: o primeiro é o auxílio na definição de requisitos do sistema e o segundo é o auxílio no planejamento dos testes, visto que o cenário compreende uma funcionalidade e assim facilita a geração de testes por caso de uso.

A Figura 14 exemplifica a criação do caso de uso da história escrita anteriormente.

Figura 14 – Caso de uso movimentação de estoque



Fonte: o autor (2017).

5.5 PLANEJAMENTO DE TESTES

O objetivo do planejamento de testes, em conjunto com as histórias e casos de uso, é estabelecer um paralelo entre o que o software está fazendo, e deve continuar fazendo, e o que o software irá fazer.

Há empresas de software que não possuem uma estrutura de testes bem formada; isso cria uma oportunidade de estabelecer um plano de testes para facilitar

manutenções futuras, sem o risco de testes manipulados, ou seja, sempre com os mesmos valores.

Existem três tipos de testes que podem ser feitos em um software: unitário, componente e de sistema. Testes unitários, como o nome sugere, testam o código baseado em suas funcionalidades singulares, centrando os esforços em validar métodos e objetos. Testes de componente são utilizados para testar um conjunto de funcionalidades atuando em sincronia, através de interface de componentes compostos. Testes de sistema pretendem verificar como todos os componentes e funcionalidades interagem como um sistema íntegro (SOMMERVILLE, 2011).

Esta unidade será responsável pela criação de testes de sistema, com os quais o fluxo de trabalho do sistema será testado para obter sempre a mesma solução. Isto é importante devido ao fato de que, quando são realizados testes de funcionalidades e componentes isolados, eles operam normalmente, mas, quando a integração entre eles é posta à prova, surgem incompatibilidades que não seriam possíveis de serem identificadas sozinhas (WHITTAKER 2011).

O planejamento é responsável apenas pelo que seu nome diz: planejar casos de testes que serão futuramente utilizados ao final do desenvolvimento para avaliar o sistema como um todo. Não é necessária a implementação em código-fonte de testes de sistema, já que serão realizados testes de unidade.

5.6 EXTRAÇÃO DOS CASOS DE USO DO CÓDIGO-FONTE

O processo de extração de requisitos será feito através da utilização de engenharia reversa. Após as fases anteriores, já é possível ter um conjunto de informações e diagramas que forneçam uma ideia de como o sistema funciona e o que ele está fazendo hoje.

Os casos de uso são parte fundamental da extração dos requisitos, de forma que serão utilizadas as atividades expostas nos casos de uso para orientar a busca do código responsável pela função apresentada. Através desta extração, será possível ter uma sequência de procedimentos e tabelas lógicas utilizadas para cada funcionalidade, que irá retroalimentar a criação de requisitos.

A partir desta etapa, presume-se acesso direto ao código-fonte e ao banco de dados. Embora existam programas que extraiam estes requisitos automaticamente, nem sempre são compatíveis com todas as linguagens de programação, mas, caso

estejam disponíveis, não existem incompatibilidades entre a extração manual dos requisitos e a extração por software.

O objetivo principal é que, ao final da extração de requisitos, uma lista de artefatos seja obtida. Alguns destes artefatos são (SANTOS, 2007):

a) Procedimentos existentes no sistema legado;
b) Estrutura de variáveis;
c) Estrutura do banco de dados;
d) Fluxo de procedimentos chamados;
e) Interfaces com usuário;
f) Requisitos funcionais;
g) Requisitos não funcionais.

Estes artefatos extraídos serão de grande ajuda na modelagem da nova arquitetura e isolamento dos componentes nas camadas do MVC.

A extração dos requisitos deverá ser baseada a nível de interface e a nível de código-fonte. Muitas vezes, em sistemas legados, as regras de negócio estão alocadas no mesmo espaço que as regras de programa. É importante fazer uma separação do que o programa deve fazer (requisitos) e o que deve fornecer apoio a ele através do código-fonte (requisitos não funcionais).

É recomendado que os requisitos possuam um identificador único, data de extração, autor e uma descrição livre de ambiguidades que explique o que é de responsabilidade daquele requisito e suas dependências.

Apresentado na Figura 15 está um quadro que pode ser utilizado para criação dos requisitos, elaborado por Plínio Ventura (2016). O quadro apresenta as informações necessárias para uma boa rastreabilidade de requisitos no sistema.

Figura 15 – Exemplo de quadro de requisitos

Identificador			
Nome			
Módulo			
Data de criação		**Autor**	
Data da última alteração		**Autor**	
Versão		**Prioridade**	
Descrição			

Fonte: Plínio Ventura (2016).

5.7 MODELAGEM DE CLASSES E MÉTODOS

Durante a modelagem da arquitetura, serão realizados quatro grandes processos: criação de classes, transformação de procedimentos em métodos, ligação de métodos a classes e padronização da arquitetura.

Este processo irá modelar tudo que será desenvolvido no processo de reengenharia. É de fundamental importância uma boa definição de qual será o nível de granularidade que o sistema deve atingir com as classes propostas.

5.7.1 Criação de classes

A definição das classes partirá do nível de abstração desejado para o sistema. Existem diversas formas de elencar as classes que serão criadas. Normalmente, são utilizadas as abordagens: *bottom-up* e *top-down*. A abordagem *top-down* analisa o sistema como um todo e, a partir dele, irá desbravando seus componentes e interações com o banco de dados. É um método complexo e, caso não exista um grande conhecimento do sistema que está sendo refatorado propenso a falhas. A abordagem *bottom-up* parte dos níveis mais baixos, normalmente do banco de dados, e vai subindo até o nível mais alto da hierarquia sistemática (ELIAS, 2014). Neste modelo, as tabelas do banco de dados tornam-se classes e os campos, atributos. Já no *top-down*, as classes são elaboradas conforme a visão que o engenheiro de software possui do sistema.

Existe também uma abordagem em que podem ser criadas classes baseadas em módulos, nos quais todos os métodos referentes ao módulo serão alocados em classes. Métodos de uso comum, por sua vez, serão alocados em classes genéricas para consumo.

5.7.2 Transformação de procedimentos em métodos

Com a estrutura de classes elaborada, é dado início ao preenchimento das mesmas com métodos. Através de métodos de decomposição, os procedimentos do sistema legado devem ser quebrados em métodos menores e independentes. Caso, por exemplo, anteriormente existia um procedimento que movimentava o estoque e, dentro deste procedimento era calculado o estoque atual do produto, serão decompostos o cálculo do estoque em um método e a movimentação do estoque em outro método. Quando for utilizado, consumirá o método do cálculo de estoque.

Os métodos criados nesta etapa podem ser apenas planejados. O vínculo às classes candidatas será realizado no passo seguinte, mas é recomendado que os métodos sejam criados com suas respectivas classes já em mente.

A utilização dos requisitos deve servir como um guia para elaboração dos métodos das classes, procurando evitar a criação de métodos em duplicidade ou sem necessidade real.

5.7.3 Ligação de métodos a classes

A ligação de métodos a classes utilizará os subprodutos criados anteriormente: classes e métodos, e fará a ligação das classes aos seus devidos métodos.

Um dos maiores riscos durante este processo é a sobreposição de métodos. Dependendo do nível de abstração tomado para a criação das classes, pode ser necessário cuidado redobrado com heranças e múltiplas heranças.

É recomendado que os métodos estejam sempre alocados ao nível mais alto possível da hierarquia, desde que o mesmo seja utilizado por este nível. Por exemplo, é incorreto alocar no topo da herança um método que será utilizado apenas por uma classe derivada. O método, neste caso, deverá estar no primeiro nível derivado que

utilizará aquela classe. Caso necessário, quando uma herança for criada com esta classe como base, o método será também herdado.

5.7.4 Padronização da arquitetura

Com a conclusão da concepção das classes candidatas, é dado início à padronização das mesmas sob a ótica dos conceitos do MVC. É importante ressaltar que a interação das classes e seu comportamento devem estar de acordo com os princípios do padrão de arquitetura organizado em três camadas: modelo, visão e controlador.

Durante a aplicação desta fase, pode ser necessária a decomposição das classes candidatas em duas ou mais classes devido à necessidade de isolar os componentes das camadas.

As classes responsáveis pela persistência de dados, seja em bancos de dados ou tabelas relacionais, serão criadas como modelos. É imprescindível que seja alocada uma classe correspondente a cada tabela do banco de dados para realizar a manipulação dos dados. O modelo não deve ter conhecimento das classes de visão, sendo programado da forma mais genérica possível.

Os pacotes responsáveis pelo gerenciamento das regras de negócio e captura das ações do modelo serão alocados como classes controladoras. Esse tipo de classe deve gerenciar as entradas de dados provenientes das classes de visão e validar as saídas possíveis; caso seja um conjunto de dados válidos, deverá consumir os métodos expostos das classes de modelo, realizando a persistência das informações e após atualizando as informações da visão.

Finalmente, a interface com o usuário será criada como classe de visão. Estas classes serão responsáveis por criar uma ponte entre o usuário e as informações alocadas no banco de dados. A recomendação é que todos os métodos de validação de entrada de dados, seja através de formulários, grades ou listas, sejam consumidos com base nos métodos criados para as classes controle, assim padronizando o controle sob o fluxo de dados.

5.8 REENGENHARIA

A parte do processo que compreende a reengenharia é responsável, basicamente, por duas tarefas: implementação das classes candidatas seguindo boas práticas e padrões de programação, e implementação de testes de unidade.

O foco de ambas as tarefas é otimizar a legibilidade e facilitar a manutenção do software, fomentando todo o objetivo principal do processo de conversão: otimizar a manutenção, mantendo a eficiência e funcionalidade do software.

5.8.1 Padrões de desenvolvimento

O desenvolvimento do software em si começará após o levantamento de todos os requisitos e construção de classes com os métodos atribuídos a elas. Este desenvolvimento propriamente dito deve seguir algumas linhas gerais que a comunidade de software concorda como sendo um conjunto de boas práticas no desenvolvimento de software. Todas as práticas apresentadas neste capítulo são de caráter opcional e podem variar de acordo com as convenções internas estabelecidas pela fábrica de software.

Algumas das características negativas atribuídas a sistemas legados são: código-fonte desnecessariamente complexo, funcionalidades decentralizadas e com um alto índice de funcionalidades duplicadas. Através dos processos feitos anteriormente para o desenvolvimento de uma arquitetura em classes MVC, o processo, caso implementado corretamente, corrige os problemas de descentralização e duplicidade encontrados no software legado, deixando como pendência o estabelecimento de um padrão de desenvolvimento para criação de um código-fonte de alto padrão, ao qual é atribuído o nome de *clean code*, ou código limpo.

Leandro Tavares, em seu artigo publicado em 2014, estabeleceu algumas características que um código considerado *clean code* deve ter (TAVARES, 2014):

a) Simples: de fácil compreensão;
b) Direto: funcionalidade está implementada para fazer o seu trabalho de forma direta;
c) Eficiente: o código funciona da maneira que foi desenhado;

d) Cuidado: código foi desenvolvido por alguém que teve preocupação de atingir pontos de qualidade, como otimização e elegância de código.

O autor continua o artigo expondo algumas boas práticas que o programador deve ter para atingir os critérios de qualidade apontados acima:

a) Nomenclatura: ao desenvolver métodos ou criação de variáveis, o desenvolvedor deve responder três perguntas: por que, o que e como. Um método que se chama apenas “validaData” não responde nenhuma das perguntas necessárias para que outro desenvolvedor dê manutenção ao código, ao contrário de um método chamado: “validaDataEmissaoNotaFiscalSaida”;
b) Evitar tipificação no nome da variável: também conhecida como notação húngara, esta forma de programação visa identificar a variável sendo declarada através de um sufixo que identifica seu tipo. Como por exemplo: “ldDataValidade”, onde o sufixo “ld” identifica uma variável **l**ocal do tipo **d**ata.
c) Limitar classes e métodos: classes ou métodos extensos demais dificultam o entendimento geral de como eles trabalham. Embora, se uma classe ou método precise de mais linhas, não existe nenhuma regra que os impeça de tê-las.
d) Formatação: um código deve estar desenvolvido estruturalmente. Como a organização de uma monografia agiliza a leitura e interpretação do texto, o mesmo é aplicado ao fluxo e compreensão de um código-fonte, e a indentação é sua base.

5.8.2 Testes automatizados

Conforme o novo software vai tomando forma e gradativamente recuperando sua complexidade, um velho problema volta a surgir: como testar?

Em softwares complexos, cobrir todas as possibilidades de como um software deve se comportar quando sujeito a cenários fora de sua rotina de trabalho é uma tarefa de difícil realização por mãos humanas. Assim, entram os testes automatizados.

Através dos conceitos utilizados no TDD (*Test Driven Development*), tudo que é desenvolvido deve ser testado e nada pode ser desenvolvido até que um teste tenha

falhado. Todo e cada teste criado para a implementação das classes deve estar sujeito a interpretação F.I.R.S.T (TAVARES, 2014).

O teste deve ser executado de forma ágil, mantendo a facilidade de aplicação para os colaboradores responsáveis por eles (Fast). Os testes devem ser realizados de forma unitária, onde não existe dependência de um teste adjacente (Independent). Deve ser possível executar diversas vezes com o retorno inalterado (Reapetable). Sua validação deve estar contida em sua concepção (Self-Validating). Deve sempre ser criado antes da implementação do código fonte (Timely) (TAVARES, 2014).

Existem diversas ferramentas estabelecidas no mercado para auxiliar no gerenciamento e implementação de testes, a implementação da ferramenta irá variar dependendo do ambiente de desenvolvimento escolhido, como por exemplo o J.Unit para desenvolvimento Java.

5.9 ENTREGAS

No ponto em que se trata das entregas do produto de software, o produto se encontra estruturado através de requisitos, casos de uso e classes desenvolvidas através de conceitos de MVC, em padrões de qualidade de desenvolvimento de software.

Embora muito da fase de entregas e homologação de produto parta da cultura estabelecida pela empresa, é recomendado que o desenvolvimento do software, que compreende desde o planejamento dos requisitos até a criação dos testes automatizados, seja feito de formas cíclicas, através da utilização de *sprints* de duração estabelecida de 30 dias. Faltando 5 dias, a funcionalidade deve ser liberada em forma de homologação para um cliente interno ou externo, para que seja avaliado e testado, e as necessidades e problemas encontrados durante a fase de testes retornem como novos requisitos ao *backlog*.

A concepção deste processo de conversão é aplicada a softwares monolíticos, nos quais os programas estão organizados por módulos, caso o ambiente de desenvolvimento tenha sido mantido. Como premissa, o mesmo deve suportar desenvolvimento orientado a objetos. Uma liberação de módulos reconstruídos é possível. Caso seja necessário trocar o ambiente de desenvolvimento, é visto como decisão da empresa de que forma será feita a liberação final.

6 APLICAÇÃO DO PROCESSO DE CONVERSÃO EM EMPRESA PILOTO

Este capítulo está organizado de forma a apresentar uma introdução do cenário de aplicação, detalhando a funcionalidade escolhida e a instituição elencada para aplicação do processo. Após, é apresentado um caminho proposto para obtenção do produto final de cada um dos passos da aplicação do processo. Junto ao caminho, é apresentada uma exemplificação do produto do passo obtido na implementação em cenário real do processo.

6.1 EMPRESA E FUNCIONALIDADE

A empresa de software escolhida age no ramo de sistemas ERP para empresas do ramo moveleiro. Atua no ramo há mais de 20 anos, contando hoje com uma base de 450 clientes ativos e uma estimativa de 4000 usuários ativos utilizando o sistema diariamente.

Para escolha da funcionalidade, foram analisados processos com uso considerável, mas que não apresentassem riscos elevados ao cliente final em caso de erros críticos. Após verificação dos números no portal de solicitações, foi selecionada a funcionalidade que trata das notificações do setor de qualidade das empresas. As notificações geradas no sistema avisam o usuário de inspeções realizadas com problemas no recebimento de matérias-primas ou em processos. Após receber o problema, o usuário pode realizar algumas ações com base nas informações a ele fornecidas.

A escolha da funcionalidade também foi retroalimentada pelo interesse da empresa no ramo metal mecânico, que possui um controle restrito de normas em relação ao setor de qualidade e é de interesse aplicar o processo em todo o módulo de qualidade do software.

6.2 APLICABILIDADE DO PROCESSO

A subfase de análise da aplicabilidade está implícita na avaliação do processo atual. Durante o levantamento de aplicabilidade, é recomendado o envolvimento de diversas áreas da instituição, visto que este processo irá nortear o resto do projeto.

Algumas decisões tomadas na análise da aplicabilidade irão definir acréscimos ou decréscimos no tempo de análise e desenvolvimento do processo de conversão: caso seja necessário reestruturar banco de dados e migrar a linguagem de desenvolvimento, uma grande parcela do tempo será alocada ao treinamento ou contratação de profissionais que dominem a linguagem.

6.2.1 Análise do caso

Foi apresentado à gerência do setor de desenvolvimento da empresa o processo de conversão em resposta à necessidade da instituição de melhorar a manutebilidade do sistema em funcionamento e com isso diminuir as horas gastas com manutenção. Além disto, foram respondidas em conjunto as três questões apresentadas na seção 5.1 – Aplicabilidade:

a) O ambiente de desenvolvimento suporta ou não diversos paradigmas? Caso não, para qual linguagem será migrado?

O ambiente de desenvolvimento suporta paradigmas de linguagem orientada a objetos, mas é utilizado de forma procedural pelos desenvolvedores da empresa. Ou seja: não são criadas classes, mas sim procedimentos sequenciais. Este foi um fator agravante na análise do processo, já que cada indivíduo desenvolve as aplicações da forma que bem entender, devido à falta de um guia de estilo ou arquitetura no desenvolvimento de software. A empresa escolheu não migrar a linguagem, mas sim desenvolver uma estrutura que forneça a possibilidade de usufruir por inteiro do paradigma orientado a objetos.

b) O banco de dados será compatível com o novo ambiente de desenvolvimento? Caso não, será reestruturado?

O banco de dados utilizado pela empresa, IBM DB2, já é um banco relacional com integração completa com a linguagem de desenvolvimento e, por escolha da empresa em razão de licenças, ele foi mantido como banco de dados principal.

c) Será possível manter a aplicação legada em funcionamento simultâneo até o fim da entrega do software refatorado?

Como não foi alterada a linguagem de desenvolvimento, a empresa manteve em funcionamento normal o sistema e foram realizadas liberações modulares das funcionalidades convertidas. Foi alocado apenas um desenvolvedor de software para aplicação do processo em funcionalidade piloto, com acesso à consulta de analistas, mas sem participação direta deles.

6.3 AVALIAÇÃO DO PROCESSO ATUAL

A sequência lógica à análise da aplicabilidade é uma avaliação do cenário atual da empresa em que é identificado se existe quadro disponível para aplicação do processo e uma estimativa de tempo e custo para empresa. Novamente, é fundamental a participação da gerência e análise para estimar tempos e custos.

6.3.1 Análise do processo atual no caso piloto

Conforme mencionado anteriormente, um levantamento de risco foi realizado para escolha da funcionalidade elencada. A empresa, em conjunto com o autor, decidiu aplicar o processo em uma funcionalidade isolada devido à necessidade de desenvolver toda a estrutura para suportar as três camadas da arquitetura MVC no sistema. A estrutura desenvolvida poderá depois ser utilizada em todas aplicações de processos futuros. Foram levantadas algumas estimativas em relação à aplicação do processo e um cálculo estimado de tempo para aplicação:

a) Tempo limite de aplicação: 60 dias, tempo de 4 horas diárias estimadas.
b) Funcionalidades a serem desenvolvidas: aplicar processo de conversão no controle de notificações da qualidade. Desenvolver estrutura para utilização do MVC, especificamente camada controladora e modelo.
c) Envolvidos: um desenvolvedor de sistemas e participação de análise e gerência indiretamente.

d) Custo com licença: não haverá custo com licenciamento devido à escolha de banco de dados e linguagem de programação. Parte integral do custo será a hora trabalhada do colaborador.
e) Data início do projeto: 07/08/2017
f) Data limite para finalização: 06/10/2017

6.4 DESENVOLVIMENTO DE HISTÓRIAS

O início do processo se dá quando as histórias de usuários começam a ser escritas, conforme mencionado na seção 5.3. Os três principais artefatos de documentação: histórias, casos de uso e requisitos, apesar de suas similaridades, irão servir a funcionalidades diferentes. A principal competência de uma história de usuário é segmentar as funcionalidades do software em pequenas histórias que servem como linha base para fomentar o *backlog* durante o *sprint*.

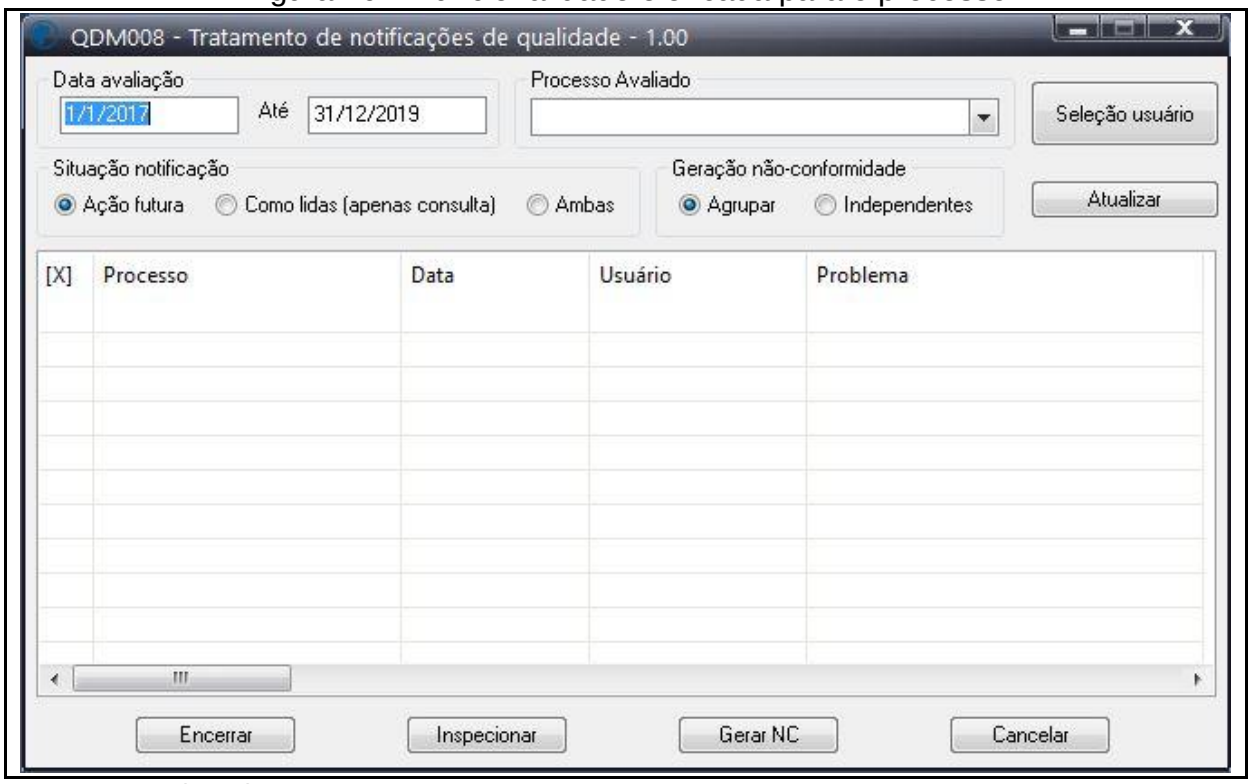
É de grande importância que as histórias sejam escritas sem que haja acesso à fonte do software, tomando apenas o ponto de vista do usuário final e a regra de negócio que originou o software.

Poderá haver divergências entre as histórias criadas e, futuramente no processo, a extração dos requisitos do código-fonte. A principal causa para essas diferenças é que nem sempre o que está desenvolvido foi documentado ou, até mesmo, está de acordo com o funcionamento esperado do software. Caso este cenário venha a ocorrer, é imperativo que a história – e consequentemente os casos de uso e requisitos – sejam adaptados à realidade correta, ou que o software seja modificado para que se comporte da forma esperada.

6.4.1 Análise do desenvolvimento de histórias no caso piloto

A fase de desenvolvimento de histórias iniciou com uma análise do comportamento da funcionalidade de tratamento de notificações durante períodos de utilização comum. Conforme a figura 16, foram realizadas diversas combinações com as seleções disponíveis do software:

Figura 16 – Funcionalidade elencada para o processo



Fonte: o Autor (2017).

Cada funcionalidade testada teve um breve relatório anotado com o comportamento que o programa apresentou após todas as funcionalidades exteriores terem sido utilizadas do ponto de vista de usuário final. Uma reunião foi agendada com o analista da área responsável pela funcionalidade e as anotações realizadas foram comparadas com o que existia de documentação e o que era esperado do software do ponto de vista da empresa.

Com o alinhamento entre o que o programa está fazendo e o que era esperado do mesmo, deu-se início à escrita das histórias de usuário. O objetivo era que fossem pequenas histórias com atores e funcionalidades bem definidas para facilitar a transição para casos de uso. Algumas das histórias escritas para a funcionalidade podem ser vistas abaixo:

a) Funcionalidade **ATUALIZAR**:

SENDO um usuário gestor da qualidade, **QUERO** consultar todos os problemas gerados através de notificações **PARA** que possa, através deles, verificar como melhorar o meu processo. **DEVO**, através dos filtros de interface, conseguir realizar consultas precisas que atendam a minha necessidade.

b) Funcionalidade **ENCERRAR**:

SENDO um usuário gestor da qualidade**, QUERO** marcar um processo na grade e, através dela, quero que seja encerrada **PARA** que possa controlar o que não tomarei ação.

c) Funcionalidade **INSPECIONAR**:

SENDO um usuário gestor da qualidade**, QUERO** marcar um processo na grade e, através dela, gerar uma inspeção mais detalhada **PARA** garantir minha qualidade de produto e processo.

Pode-se perceber que as histórias estão escritas de um ponto de vista de usuário final. Não se entrou no mérito de código-fonte do software ou de arquitetura do mesmo.

6.5 DESENVOLVIMENTO DE CASOS DE USO

Durante o desenvolvimento de casos de uso, será apenas transcrito para uma forma visual o que foi desenvolvido das histórias de usuário. Como as histórias são pequenas e com poucos detalhes, durante o desenvolvimento de um caso de uso, é possível agregar mais fluxos do que seria possível através de frases simples (LONGO e SILVA, 2014).

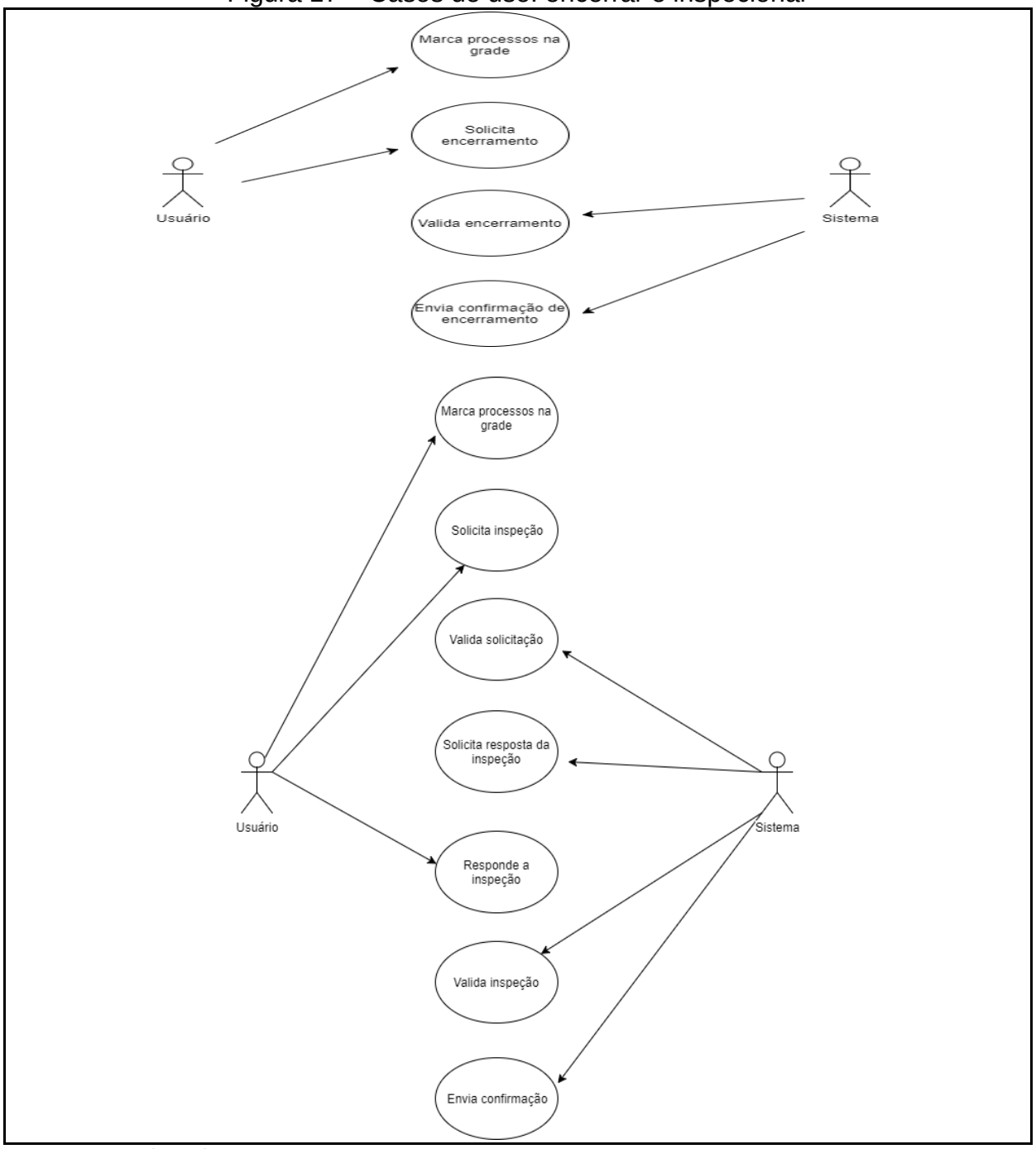
O maior cuidado necessário durante esta fase é que os casos de uso fiquem contidos em suas histórias de usuário. Detalhes adicionais são importantes para o entendimento do fluxo visual, desde que se atenham ao escopo pré-definido pela história.

6.5.1 Análise do desenvolvimento de casos de uso no cenário piloto

Os casos de uso foram criados com base nas histórias desenvolvidas para a funcionalidade. Durante este momento, ainda são casos de uso isolados por histórias, ou seja: não existe interação entre o caso de uso da inspeção e o encerramento. Quando a fase de integração do caso de uso com o panorama geral, descrita na seção 5.9, for atingida, todos estes casos de uso irão formar um panorama maior do software.

Na figura 17, pode-se perceber a transcrição das histórias "encerrar" e "inspecionar" para um fluxo visual, através do caso de uso.

Figura 17 – Casos de uso: encerrar e inspecionar



Fonte: o Autor (2017).

6.6 PLANEJAMENTO DE TESTES

O desenvolvimento dos testes de interface é vital para garantir a assertividade do processo de conversão após o mesmo ter passado pelo processo de reengenharia.

Durante esta fase, será desenvolvido um roteiro de testes externo, para que seja possível aplicar sobre o programa reescrito e obter as mesmas saídas utilizando as mesmas entradas.

Conforme descrito por Craig e Jaskiel (2002), o conjunto de ferramentas caso e procedimento de teste descrevem um cenário de teste através da roteirização do funcionamento do software com entradas e saídas. Esse roteiro irá gerar caminhos que, por sua vez, gerarão casos de testes.

6.6.1 Análise do cenário de planejamento de testes no caso piloto

A forma encontrada para gerar com maior facilidade o roteiro de testes foi mapear o fluxo principal do software. Isso foi atingido isolando a rota em comum que as funcionalidades “consulta”, “encerramento” e “geração de não conformidade” tinham. Conforme pode ser visto abaixo, a rota principal chega até a atualização de dados para consulta do usuário final:

Tabela 1 – Fluxo principal do cenário de testes

Ação	**Descrição da execução do fluxo**
1 - [**IN**]	Usuário informa intervalo de data de avaliação;
2 - [**IN**]	Usuário seleciona o processo a ser consultado;
3 - [**IN**]	Usuário seleciona os usuários responsáveis pelas notificações;
4 - [**IN**]	Usuário seleciona a situação de notificação que gostaria de ver;
5 - [**IN**]	Usuário requisita os dados através da função: Atualizar;
6 - [**OUT**]	Sistema informa as notificações filtradas com base nos filtros;

Fonte: o Autor (2017).

O fluxo especificado é necessário em todos os quatro cenários disponíveis no software. Com a rota estabelecida, as funcionalidades foram transformadas em fluxos variantes que dão continuidade ao processo:

Tabela 2 – Fluxo alternativo do cenário de testes

Ação	**Descrição da execução do fluxo**
7 - [**Variante**]	Usuário encerra notificações;
7.1 - [**IN**]	Usuário seleciona as notificações na grade através da primeira coluna;
7.2 - [**IN**]	Usuário requisita o encerramento das notificações através da função: Encerrar;
7.3 - [**OUT**]	Sistema confirma o encerramento e remove a notificação da grade;
Exceção 7.3a	Não existem notificações selecionadas na grade;
7.3a.1 - [**OUT**]	Sistema notifica que não existem notificações selecionadas na grade;
7.3a.2 - [**OUT**]	Sistema retorna para passo 6;
Exceção 7.3b	Notificação selecionada já está encerrada;
7.3b.1 - [**OUT**]	Sistema notifica que a notificação já se encontra encerrada;
7.3b.2 - [**OUT**]	Sistema retorna para passo 6;
8 - [**Variante**]	Usuário inspeciona notificações;
8.1 - [**IN**]	Usuário seleciona as notificações na grade através da primeira coluna;
8.2 - [**IN**]	Usuário requisita a inspeção das notificações através da função: Inspecionar;
8.3 - [**OUT**]	Sistema solicita resposta do formulário de inspeção;
8.4 - [**IN**]	Usuário informa os dados da inspeção;
8.5 - [**OUT**]	Sistema confirma realização da inspeção e remove a notificação da grade;
Exceção 8.3a	Não existem notificações selecionadas na grade;
8.3a.1 - [**OUT**]	Sistema notifica que não existem notificações selecionadas na grade;
8.3a.2 - [**OUT**]	Sistema retorna para passo 6;
9 - [**Variante**]	Usuário gera não conformidades;
9.1 - [**IN**]	Usuário seleciona as notificações na grade através da primeira coluna;
9.2 - [**IN**]	Usuário seleciona se irá agrupar ou não as não conformidades;
9.3 - [**IN**]	Usuário requisita o encerramento das notificações através da função: Gerar NC;
9.4 - [**OUT**]	Sistema confirma a geração da não conformidade e remove a notificação da grade;
Exceção 9.4a	Não existem notificações selecionadas na grade;

9.4a.1 - [**OUT**]	Sistema notifica que não existem notificações selecionadas na grade;
9.4a.2 - [**OUT**]	Sistema retorna para passo 6;

Fonte: o Autor (2017).

Com o roteiro de funcionamento do programa realizado, é possível, através dos casos de teste, criar roteiros para testes, conforme o exemplo abaixo do fluxo principal com variante 7:

a) **Objetivo**: Fluxo principal com variante 7;
b) **Caminho**: 1,2,3,4,5,6,7,7.1,7.2,7.3;
c) **Roteiro**: Usuário informa o intervalo de ano atual nos campos de data de avaliação, usuário seleciona processo desejado no processo avaliado, usuário marca todos usuários pertinentes na seleção de usuário, usuário seleciona as notificações que deseja ver, usuário atualiza as informações, sistema exibe o retorno da consulta, usuário seleciona a notificação através da grade, usuário seleciona o encerramento, sistema confirma o encerramento;
d) **Resultado**: Notificação encerrada;

Existe a possibilidade de continuar o roteiro de testes com condições de testes sobre entradas e saídas para especializar ainda mais o cenário de testes do software, podendo ser feito através de métodos como decomposição de requisitos.

6.7 EXTRAÇÃO DE REQUISITOS

O objetivo principal da extração de requisitos do código-fonte é utilizar as histórias e casos de uso elaboradas anteriormente e extrair do código-fonte os procedimentos, tabelas e estruturas de variáveis necessárias para que aquela história funcione a nível de código-fonte. Conforme apresentado anteriormente, o paradigma de programação procedural opera através de um encadeamento de funções e procedimentos para atingir um produto final. Um dos subprodutos da extração é mapear de tal forma que seja possível identificar o ponto de entrada e o ponto de saída dos procedimentos.

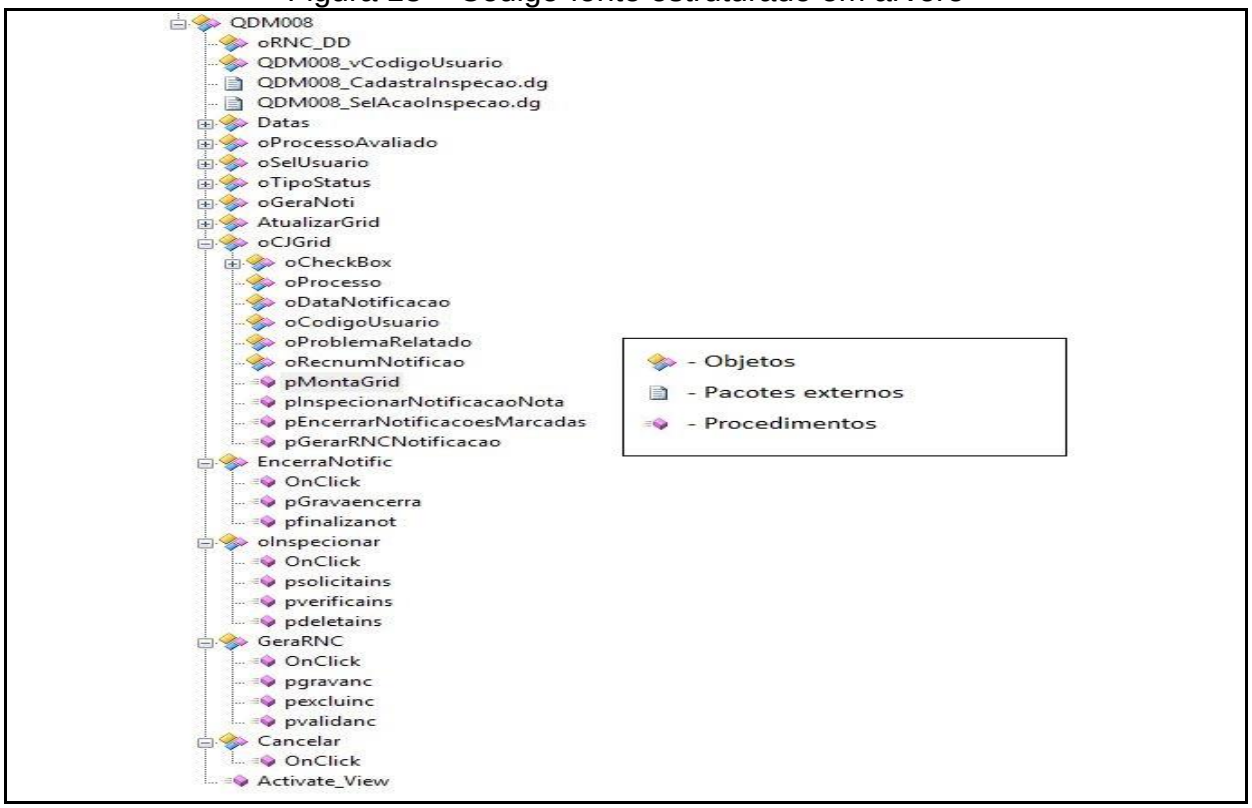
Após a extração dos produtos do código-fonte, estes serão particionados em requisitos menores, passíveis de desenvolvimento através do modelo informado na seção 5.6. Estes requisitos irão fomentar o desenvolvimento de classes e métodos para o funcionamento do software.

6.7.1 Análise da extração de requisitos do cenário piloto

Após analisar e realizar testes, percebeu-se maior facilidade em mapear primeiramente a estrutura geral do código-fonte, após extrair o fluxo de procedimentos para as histórias e, por fim, para cada uma das histórias, extrair a estrutura do banco de dados. Conforme mencionado anteriormente, não é de caráter obrigatório o desenvolvimento de diagramas ER ou de classes para a aplicação do processo: os diagramas foram desenvolvidos com fins acadêmicos para ilustrar (de forma visual) o processo aplicado.

Para visualização da estrutura do software como um todo, foi utilizada uma ferramenta nativa da linguagem de programação – *code explorer.* Através dela, foi exportada em uma visualização em árvore da estrutura do programa. A estrutura pode ser vista através da figura 18.

Figura 18 – Código-fonte estruturado em árvore



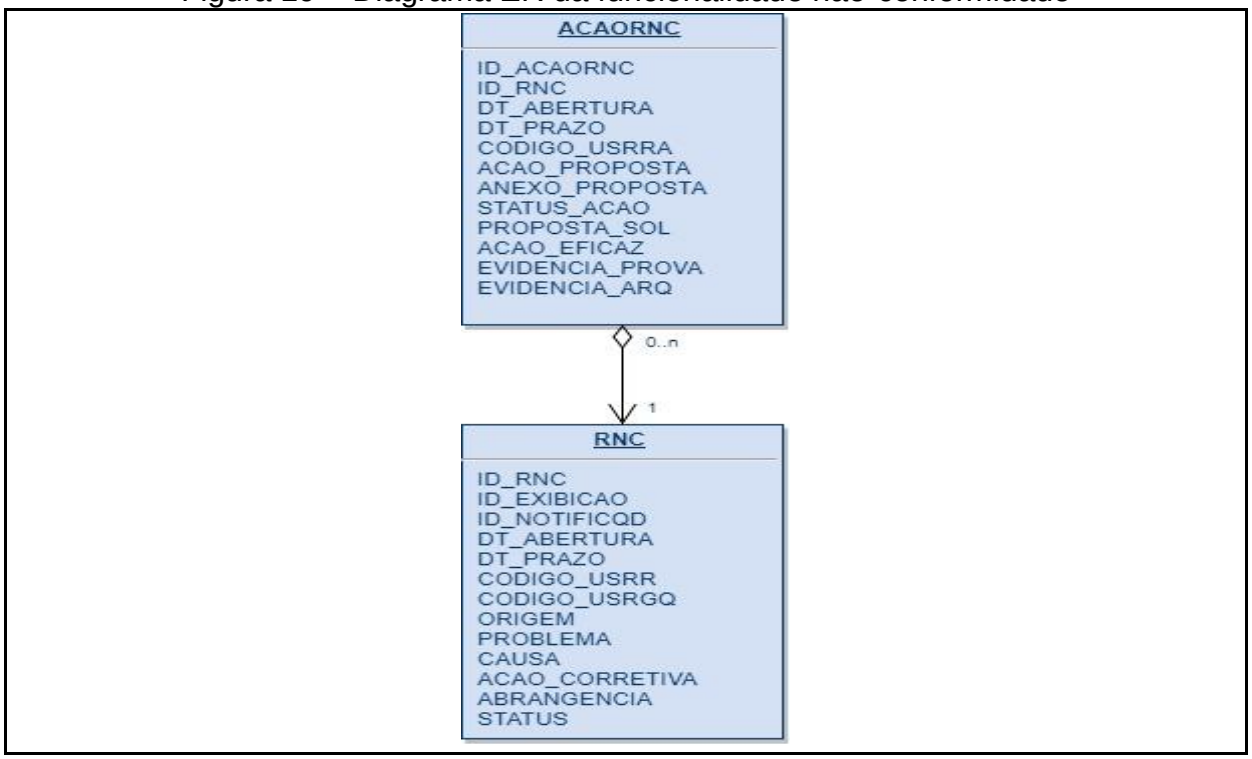
Fonte: o Autor (2017).

Após este levantamento, voltou-se às histórias e casos de uso e deu-se início à análise dos fluxos de procedimentos. Isso foi feito através da aplicação de técnicas de engenharia reversa e análise do código-fonte através de ferramentas de depuração da própria linguagem de programação. Utilizando como exemplo a história de gerar não-conformidades, o fluxo verificado (procedimento e objeto de origem) é o que segue:

- OnClick (**AtualizarGrid**)
- pMontaGrid (**oCJGrid**)
- OnEndEdit (**oCjGrid**)
- OnClick (**GeraRNC**)
- pvalidanc (**GeraRNC**)
- pgravanc (**GeraRNC**)

Finalmente, foi expandida a busca para que fosse possível a extração das tabelas lógicas envolvidas nos procedimentos de manipulação de dados. As tabelas estavam sendo gravadas no procedimento "pgravanc": ACAORNC e RNC, conforme pode ser visto pela figura 19:

Figura 19 – Diagrama ER da funcionalidade não-conformidade



Fonte: o Autor (2017).

Neste momento, foi possível assumir uma estrutura de procedimentos e banco de dados para que se desse início à escrita dos requisitos para o desenvolvimento da arquitetura. A linguagem utilizada não possuía suporte nativo à utilização do MVC, então foi adicionado como requisito não-funcional o desenvolvimento destas funcionalidades.

Conforme mostra a figura 20, o desenvolvimento da camada de controle é um requisito que não está vinculado diretamente à história para gerar não-conformidades, mas é vital para a continuidade do projeto. Já o requisito “PRO0000002” mostra a necessidade de isolar a validação para a gravação dos dados na vindoura camada de controle.

Figura 20 – Exemplificação de requisitos gerados

Identificador	PRO000001		
Nome	Projeto MVC - Camada de controle		
Módulo	Qualidade		
Data de criação	15/08/2017	**Autor**	Giovani Biondo
Data da última alteração		**Autor**	
Versão	1.0	**Prioridade**	Alta
Descrição	Desenvolver camada de controle para derivação futura.		

Identificador	PRO000002		
Nome	Projeto MVC - Validação de não-conformidade		
Módulo	Qualidade		
Data de criação	15/08/2017	**Autor**	Giovani Biondo
Data da última alteração		**Autor**	
Versão	1.0	**Prioridade**	Média
Descrição	Procedimentos de validação para gravação de NC - QDM008		

Fonte: o Autor (2017).

6.8 CRIAÇÃO DE CLASSES

Quando é atingido o ponto de concepção das classes, já existem documentados fluxos de código-fonte e requisitos necessários para o funcionamento do software. Conforme mencionando na seção 5.7.1, existem algumas abordagens

para o desenvolvimento de classes como *top-down* e *bottom-up*, além do desenvolvimento de classes agrupadas por funcionalidades. Um dos maiores riscos que podem ocorrer durante a fase de criação de classes é a granularidade que as mesmas terão.

É importante criar um conceito de uma estrutura de classes que se adaptem ao ambiente em que o software está envolvido. Caso seja um ambiente de alta mudança de requisitos, não seria propriamente adequada uma extensa gama de pacotes auxiliares, visto que estariam sendo sempre especializados. Mas, em contrapartida, os agrupamentos de muitas funcionalidades irão retroceder o software ao lugar comum do procedural.

6.8.1 Análise da criação de classes no cenário piloto

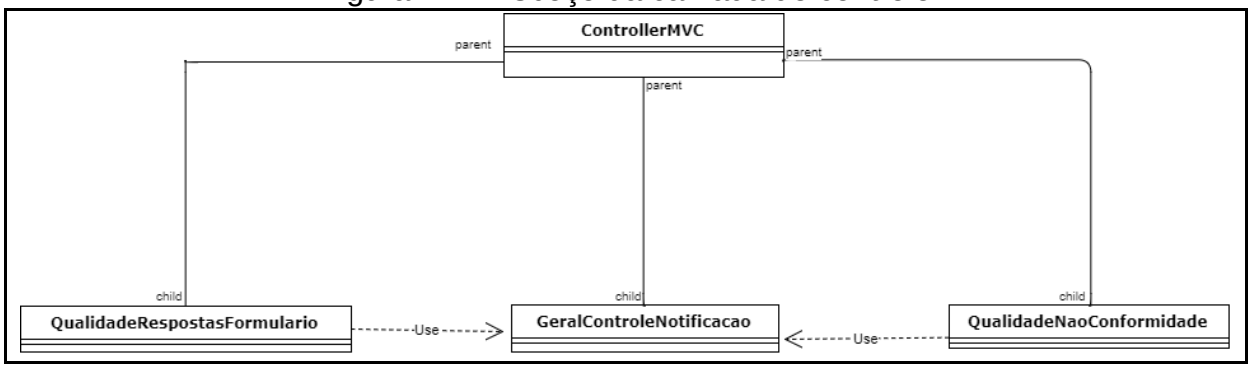
A empresa teste possui um cenário relativamente particular: o sistema possui um núcleo modularizado, mas, caso se faça necessário, especializações são realizadas para clientes – chamadas de específicos. Além desta particularidade, o núcleo do sistema sofre diversas alterações, tanto de melhorias quanto correções de problemas.

Levando em consideração ambas as características, foram adotados dois agrupadores para idealização de classes candidatas: módulo em que ela está inserida e funcionalidade maior. O agrupador modular é para que todas as funcionalidades vinculadas àquele conjunto específico de programas fiquem agrupadas e, caso haja a necessidade de utilização a partir de outros módulos, seja instanciado um objeto e não realizada uma nova especialização da classe. A funcionalidade maior – como é chamada a principal funcionalidade da classe – no caso da funcionalidade candidata, uma das funcionalidades maiores é a geração de uma não-conformidade. Neste caso, será criada uma classe para o módulo de qualidade que faça a gestão das não conformidades.

As classes de persistência serão criadas de forma diferente: cada tabela do banco de dados dará origem a uma classe de persistência específica com suas próprias validações.

Conforme a figura 21, foi realizado um esboço de como a camada de controle ficaria já sob a ótica do padrão MVC.

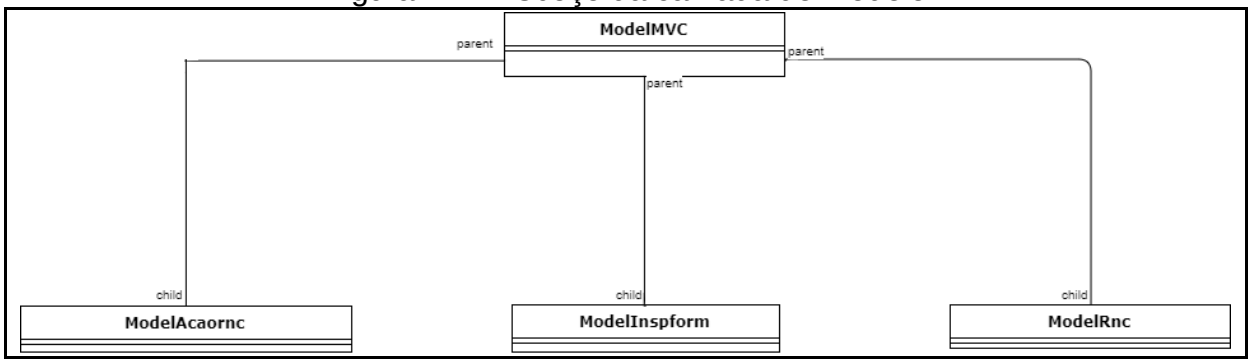
Figura 21 – Esboço da camada de controle



Fonte: o Autor (2017).

Foi realizado um esboço da camada de modelo para a persistência dos dados, conforme pode ser visto na figura 22.

Figura 22 – Esboço da camada de modelo



Fonte: o Autor (2017).

6.9 TRANSFORMAÇÃO DE PROCEDIMENTOS EM MÉTODOS

O principal artefato de trabalho da fase de decomposição de procedimentos é o fluxo extraído da funcionalidade. Deve ser feita uma análise dos procedimentos e decompor em métodos que realizem apenas uma funcionalidade, facilitando a utilização a partir de outras fontes sem a necessidade de especializações.

6.9.1 Análise do processo de transformação de procedimentos no caso

Um dos problemas mais graves da atual situação do software piloto é a dificuldade de discernir onde um procedimento começa e o outro deveria acabar. Existem diversos exemplos de código-fonte nos quais a validação, persistência e manipulação da camada de visão estão mesclados uns aos outros. Tomando como

exemplo o procedimento: "pgravanc", o objetivo do código é verificar quais registros estão marcados na grade, realizar validações sob o usuário que está realizando a geração da não-conformidade, validar o status da não-conformidade e se a data está no prazo. Caso tudo esteja nos de acordo, deve-se fazer a persistência no banco de dados.

Conforme pode ser visto na figura 23, tudo que foi descrito está em apenas um procedimento.

Figura 23 – Código-fonte antes da reengenharia

```
Procedure pgravanc
    Boolean lbResposta
    Handle lhDataSource
    Number lnRecnumNotificacao
    Integer li liTam
    Boolean lbGrava
    String lsProcessoAgrupado lsStatus
    Date ldDataAbertura ldDataPrazo
    tDataSourceRow[] Dados

    If (ppiGeraNotificacao(Self) = 1) Move (True) to lbResposta
    Else If (ppiGeraNotificacao(Self) = 0) Move (False) to lbResposta

    Get phoDataSource to lhDataSource
    Get DataSource of lhDataSource to Dados

    Move (SizeOfArray(Dados)) to liTam
    Move (liTam - 1) to liTam

    For li from 0 to liTam
        If (Dados[li].sValue[0] = "Y") Begin
            Move (fIsUsuarioGestorDaQualidade(Self,Codigo_Usuario)) to lbGrava
            Get fSQLRetorna ("Select STATUS From RNC Where ID_RNC = " + String(piIDRnc)) 1 to lsStatus
            If ((lsStatus = "F") or (lsStatus = "AP")) Move False to lbGrava

            Move RNC.DT_PRAZO to ldDataPrazo
            Move RNC.DT_ABERTURA to ldDataAbertura

            If (ldDataPrazo < ldDataAbertura) Move False to lbGrava

            If (lbGrava) Begin
                Send Clear to (oRNC_DD(Self))
                Set Field_Changed_Value of (oRNC_DD(Self)) Field RNC.PROBLEMA to Dados[li].sValue[4]
                Set Field_Changed_Value of (oRNC_DD(Self)) Field RNC.ORIGEM to "R"
                Set Field_Changed_Value of (oRNC_DD(Self)) Field RNC.ID_NOTIFICQD to Dados[li].sValue[5]
                Send Request_Save to (oRNC_DD(Self))
            End
            Else Begin
                Move (lsProcessoAgrupado + " " + Trim(Dados[li].sValue[4])) to lsProcessoAgrupado
            End
        End
    Loop
End_Procedure
```

Fonte: o Autor (2017).

Este trecho de fonte foi decomposto em cinco funcionalidades diferentes, as quais poderão ser decompostas ou realocadas para atender o padrão de arquitetura MVC proposto:

a) pLeituraGrade - lê os registros marcados na grade;
b) fIsUsuarioGestorDaQualidade - valida se o usuário é gestor da qualidade;
c) fValidarStatusRNC - valida se a RNC pode ter uma não-conformidade vinculada;
d) fValidarPrazoRNC - valida se a data de abertura da RNC ainda está no prazo;
e) pGravarNaoConformidade - persiste os registros no banco de dados;

6.10 LIGAÇÃO DE MÉTODOS A CLASSES

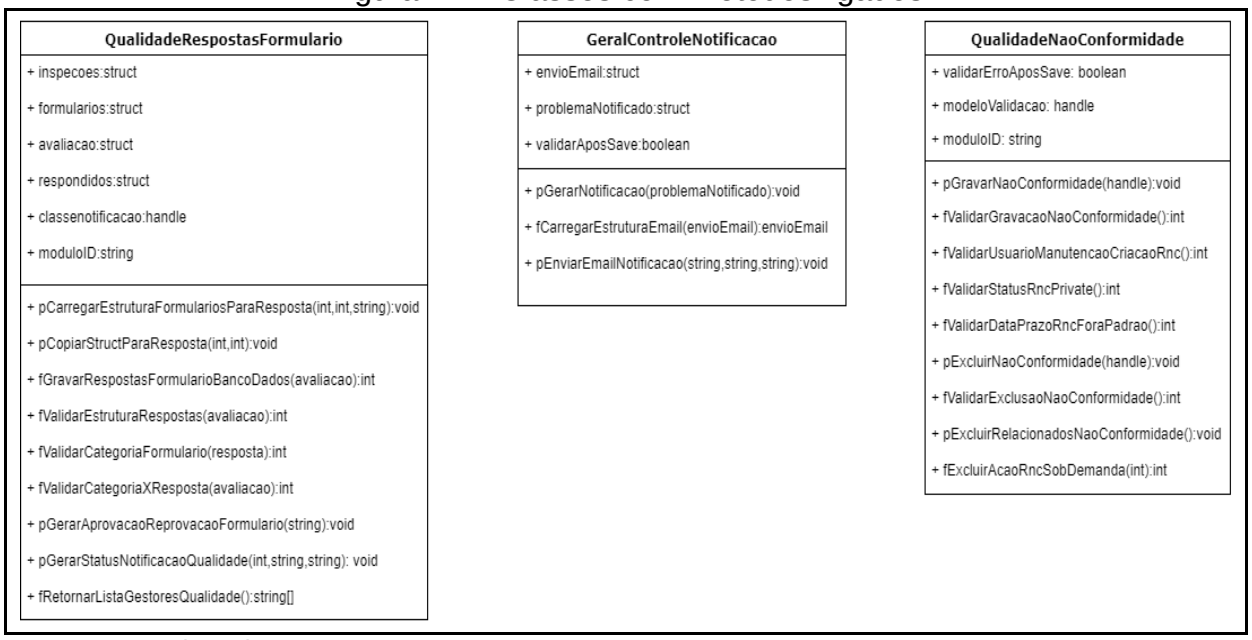
Durante a fase de ligação dos métodos, os produtos gerados a partir da transformação de procedimentos em métodos e criação de classes serão unidos para formar um sistema compreensível. O principal desafio da ligação de métodos é levar em consideração onde os métodos devem ficar especializados e onde devem ser utilizados em nível superior da hierarquia de classes. A forma que foi utilizada para a aplicação do processo foi, quando possível, manter os métodos vinculados ao nível mais baixo da hierarquia, e, caso houver necessidade de reutilização, realocar para os níveis superiores. Desta maneira, irá mitigar a ocorrência de cenários em que as classes de maior nível hierárquico possuem diversos procedimentos que apenas uma de suas classes derivadas utiliza.

Além disso, é importante que a fase seja realizada com o pensamento direcionado à organização do padrão MVC. Métodos de persistência devem ficar na modelo, métodos de validação e procedimentos especializados na camada controle, e métodos responsáveis por exibir informações ao usuário, na camada visão.

6.10.1 Análise da ligação de métodos a classes no caso

Devido à falta de uma integração com a estrutura MVC, não houveram maiores dificuldades durante o processo de ligação de métodos. Além disso, com a organização modular e por funcionalidade, foi uma questão de alocar nas classes candidatas os métodos responsáveis pelas funções que nomeavam as classes, conforme pode ser vistos na figura 24:

Figura 24 – Classes com métodos ligados



Fonte: o Autor (2017).

Foi necessária a conceptualização de alguns novos métodos para atender às novas necessidades que o padrão de arquitetura iria exigir, a abstração do controle de erros (modeloValidacao) para classes superiores e a validação baseada no módulo que está invocando o procedimento (moduloID).

6.11 PADRONIZAÇÃO DA ARQUITETURA

A padronização da arquitetura é realizada nos modelos de classes já com os métodos vinculados. Para utilização do MVC, pelo menos três novas classes deverão surgir para atender o conceito do padrão de arquitetura. A forma de trabalho que mais promoveu tranquilidade durante esta separação foi manter-se fortemente ligado aos conceitos do MVC: persistência, regras de negócio, interface. Todos os métodos e classes que eram responsáveis por algum dos três pilares tornaram-se derivados das classes do MVC.

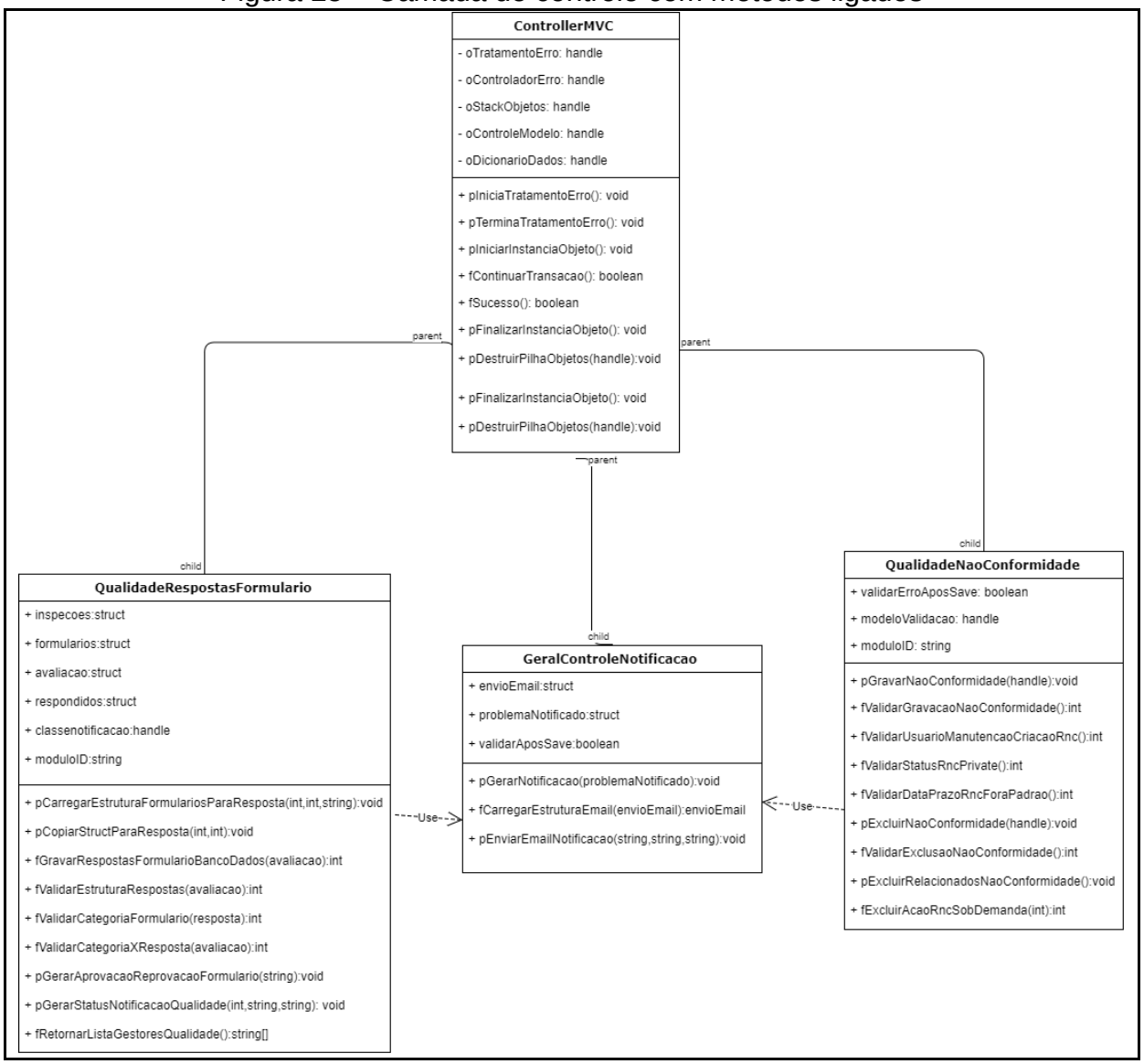
6.11.1 Análise da padronização de arquitetura no caso

A maior mudança realizada nos modelos criados pelos processos anteriores foi a transferência dos procedimentos de persistência para as camadas de modelo. Foram repensados fluxos de trabalho para que as camadas de controle apenas

consumissem métodos das camadas que iriam realizar a manutenção dos dados. Com isso, foi possível conceitualizar uma barreira de validações através dos métodos da camada controladora e, caso a instrução chegasse ao nível de modelo, a camada poderia realizar o procedimento sem a necessidade de validações adicionais.

Conforme pode ser visto através da figura 25, a estrutura foi mantida, mas alguns métodos novos foram atribuídos à camada de controle e as classes: "qualidaderespostasformulario", "geralcontrolenotificacao" e "qualidadenaoconformidade" tornaram-se especializadas da camada de controle.

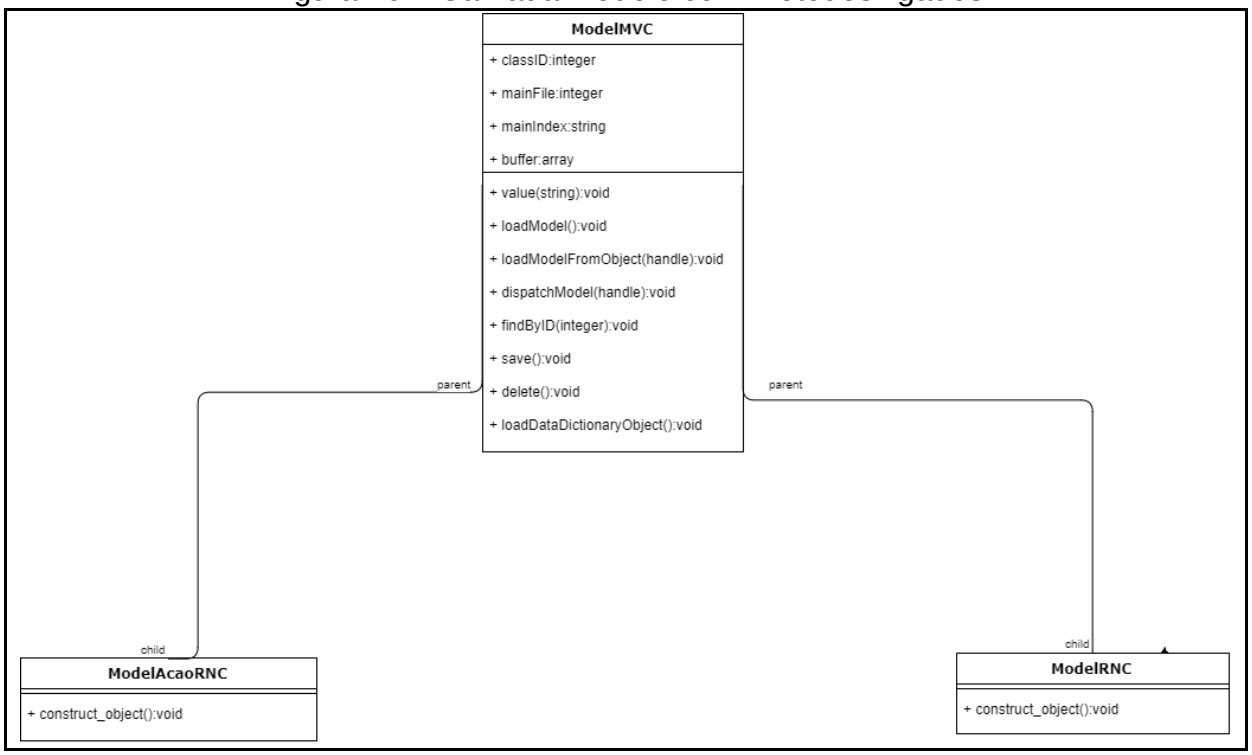
Figura 25 – Camada de controle com métodos ligados



Fonte: o Autor (2017).

Já na camada de modelo, conforme pode ser visto na figura 26, não existem muitas opções para especialização, de modo que a grande maioria dos métodos está ligada à camada modelo e suas duas especializações possuem apenas os métodos construtores.

Figura 26 – Camada modelo com métodos ligados



Fonte: o Autor (2017).

Conforme é possível perceber pelos diagramas, a organização MVC promove uma separação clara entre as camadas, facilitando posteriormente a organização do código-fonte.

6.12 PADRÕES DE DESENVOLVIMENTO

Durante a etapa de reengenharia, foi posto em prática todo o planejamento desenvolvido pelas etapas anteriores. É de grande importância que seja implementado o novo código-fonte através de padrões de código. Um dos principais problemas encontrados durante a aplicação do processo é a falta de padronização do código e o local onde o mesmo deveria estar localizado.

Conforme mencionado na seção 5.8.1, não existem regras ou obrigatoriedades em relação à forma com que o código-fonte é escrito, mas o

estabelecimento de normas e padrões pode facilitar a replicação do processo em outras partes do software, facilitando posteriormente a manutenção. Os pilares considerados importantes durante o desenvolvimento do processo foram: simplicidade, eficiência e clareza.

6.12.1 Análise dos padrões de desenvolvimento na empresa piloto

A dificuldade de implementar um padrão de desenvolvimento é um desafio presente até os dias de hoje na empresa piloto. Os desenvolvedores não têm à sua disposição uma norma de desenvolvimento escrita e disponível. Foi assumida, como uma das responsabilidades do processo, fornecer esta normatização.

É importante ressaltar a presença de notação húngara e a exposição do tipo da variável em sua declaração. Embora seja desencorajada a utilização deste método de declaração de procedimentos e variáveis, no caso da empresa ele foi utilizado para facilitar a utilização do recurso de sugestão da IDE, já que a mesma não consegue encapsular métodos a classes para sugestão durante o desenvolvimento.

Para o desenvolvimento de classes e métodos foram sugeridas as seguintes normas:

a) Padronização de utilização para pacotes externos: todos os pacotes externos devem estar presentes no início do código fonte, antes da declaração da classe;
b) Classes autocontidas: todas as classes desenvolvidas para o sistema devem compilar sem necessidade de ordenação específica ou vínculo a pacotes externos;
c) Nome de métodos: os métodos devem possuir nomenclaturas autoexplicativas, utilizando sempre o tempo infinitivo da língua portuguesa.
d) Modificadores de visibilidade: utilizar os modificadores de visibilidade para encapsular os métodos às classes, identificando uso interno ou externo dos métodos;
e) Métodos simples: sempre que possível, um método deve realizar apenas uma operação. Métodos complexos podem utilizar diversos métodos menores.

f) Formatação: sempre escrever um código-fonte na formatação recomendada, ou seja, métodos e classes devem ser devidamente identados.
g) Herança: sempre que um novo método possa ser reutilizado por outras classes, o mesmo deve ser alocado na classe pai.

A implementação destas normas pode ser vista através da figura 27, o procedimento de encerramento das notificações. O único processo realizado a nível de visão é a leitura dos registros no objeto da grade. As modificações de status são realizadas através de um método exposto na classe "cQualidadeRespostasFormulario":

Figura 27 – Código-fonte de visão reescrito através do processo

```
Procedure pEncerrarNotificacoesMarcadas
    Handle lhQualidade lhDataSource
    Number lnRecnumNotificacao
    Integer li liTam
    tDataSourceRow[] Dados

    Get phoDataSource to lhDataSource
    Get DataSource of lhDataSource to Dados

    Move (SizeOfArray(Dados)) to liTam
    Move (liTam - 1) to liTam

    Get Create (RefClass(cQualidadeRespostasFormulario)) to lhQualidade

    For li from 0 to liTam
        If (Dados[li].sValue[0] = "Y") Begin
            Send pGerarStatusNotificacaoQualidade of lhQualidade Dados[li].sValue[5] "S" "N"
        End
    Loop

    Send Destroy to lhQualidade
End_Procedure
```

Fonte: o Autor (2017).

Por sua vez, conforme pode ser visto pela figura 28, a geração da notificação que está localizada na camada de controle, consome o método "FindByRecordNumber". Este método pertence à classe de modelo, com o objetivo de posicionar o registro que deve ser alterado. Após este posicionamento, o método ainda realiza o consumo de outros dois métodos da camada de modelo: "save" e "value".

Figura 28 – Código-fonte de modelo e controle reescrito através do processo

```
{Visibility = Public}
Procedure pGerarStatusNotificacaoQualidade Integer piRecnum String psStatusLido String psStatusTratar
    Handle lhNotificqdModel
    Integer liRetorno

    Get fEmpilharObjeto (RefClass(cModelNOTIFCQD)) to lhNotificqdModel
    Send FindByRecordNumber of lhNotificqdModel piRecnum
    If (Found) Begin
        Set Value of lhNotificqdModel Field NOTIFCQD.STATUS_LIDO   to psStatusLido
        Set Value of lhNotificqdModel Field NOTIFCQD.STATUS_TRATAR to psStatusTratar
        Send Save of lhNotificqdModel
    End
End_Procedure

{Visibility=Public}
Procedure FindByRecordNumber Integer piRecnum
    Clear (ppiMainFile(Self))
    Set_Field_Value (ppiMainFile(Self)) 0 to piRecnum
    Vfind (ppiMainFile(Self)) 0 Eq
    If (Found) Begin
        Send LoadModel
    End
End_Procedure
```

Fonte: o Autor (2017).

6.13 TESTES AUTOMATIZADOS

Conforme conceituado na seção 5.8.2, a metodologia ágil TDD promove um cenário de testes controlados, onde todo código-fonte desenvolvido possui um teste desenvolvido de antemão para validá-lo. Os testes praticados no TDD funcionam partindo de afirmações onde o retorno de um método é testado contra os parâmetros a ele enviados. Caso esta afirmação se confirme, o teste retorna com sucesso (TAVARES, 2014).

6.13.1 Análise dos testes automatizados aplicados no caso

Em conjunto com a falta de padrões de desenvolvimento, a empresa piloto sofre com a dificuldade em testar o software produzido pela mesma. Contando hoje com mais de quinze mil pacotes de código-fonte em seu repositório, os testes são realizados apenas a nível de interface por analistas de suporte. Além da mudança de cultura para o desenvolvimento orientado a objetos, foi necessária a implementação de uma cultura de testes. Esta implementação foi realizada através do

desenvolvimento de uma classe para testes automatizados, e, através dela, foram aplicados os testes.

Conforme mostra a figura 29, um teste é realizado com base no método de encerrar a notificação. Caso ele tenha sucesso, a afirmação retorna verdadeira.

Figura 29 – Código-fonte para testes de unidade

```
Use cQualidadeFormularios.pkg
Use cTestesUnidade.pkg

Class cTestesQualidade is a cTestesUnidade
    Procedure pTestarEncerramentoNotificacoes
        Handle lhQualidade

        Get Create (RefClass(cQualidadeRespostasFormulario)) to lhQualidade
        Send pGerarStatusNotificacaoQualidade of lhQualidade 25 "S" "N"
        Send AssertBAreEqual (True) (NOTIFCQD.STATUS_TRATAR = "S")
        Send Destroy to lhQualidade
    End_Procedure
End_Class
```

Fonte: o Autor (2017).

6.14 ENTREGAS

Durante o processo de desenvolvimento, foram utilizados artefatos do *Scrum* para orientar o andamento do projeto. Elaborados através de *sprints* semanais, o *backlog* foi retroalimentado pelos requisitos gerados no início da aplicação do processo. Ao final de cada sprint semanal, uma reunião era realizada com um analista de software, para que fossem elencados os requisitos mais importantes para dar continuidade ao processo, mantendo como prioridade o desenvolvimento da estrutura MVC e após a refatoração do código-fonte.

A funcionalidade readequada foi testada por um período de uma semana por um analista de suporte e um analista de software. Após este período, a funcionalidade foi liberada em formato de homologação para dois clientes piloto.

Ao final da segunda semana de setembro de 2017, a funcionalidade foi liberada para todos os clientes da empresa em caráter final.

6.15 RESULTADOS DA APLICAÇÃO

A motivação que promoveu o desenvolvimento deste estudo era encontrar uma maneira de revitalizar sistemas legados através de boas práticas de programação e, assim, facilitar a manutenção do software e aumentar sua vida útil no mercado.

Durante o levantamento do referencial teórico, adotou-se a orientação a objetos e o padrão de arquitetura MVC como etapa de evolução para o software. O objetivo alterou-se para a possibilidade de desenvolvimento de um processo de um sistema legado utilizando o paradigma e o padrão de arquitetura proposto.

Com a conceitualização do processo finalizada, acrescentaram-se à lista alguns objetivos que o processo visava atingir. Durante esta seção, é exposta a completude de cada objetivo levantado para atingimento do processo.

6.15.1 Completude dos objetivos do processo de conversão

A criação de documentação para o software era um objetivo secundário, mas não de menor importância para o processo. A visão que o processo tem sob a documentação é meio para um fim. Ou seja, a documentação é gerada através de histórias de usuário e casos de uso, ambos sendo utilizados para promover a continuidade do processo, mas, além disso, servindo como uma documentação rudimentar do software. Acredita-se que, neste ponto, o software atingiu o objetivo, visto que os casos de uso foram adotados na documentação já existente da empresa.

O segundo ponto era manter a funcionalidade e estabilidade do software, evitando comprometimento da empresa com o mercado, liberando um produto de inferior qualidade. Foram tomados passos tanto na implementação do processo, quanto na entrega do produto final para diminuir ao máximo as chances de liberação problemática. A preparação do roteiro de testes, a utilização de testes automáticos e o ciclo de entregas para homologação interna e externa foram algumas das mecânicas estabelecidas para atingir este objetivo. Com a liberação do produto no dia 11/09/2017, não existem ainda problemas relatados com o cliente final, ou seja, o software continua a se comportar da maneira esperada.

A facilitação da manutenção foi um agente catalisador para o desenvolvimento do processo. Junto a ela, tomou-se como objetivo a melhoria do código-fonte. Com a adoção do paradigma orientado a objetos e o padrão de arquitetura MVC, buscou-se

melhorar a forma de manter o software e, apoiando a aplicação das mesmas, objetivou-se aplicar as melhores práticas de desenvolvimento cabíveis ao cenário da empresa. Devido à característica qualitativa de uma avaliação de código-fonte, não é possível afirmar que o software esteja melhor apenas porque está escrito de acordo com as orientações e boas práticas, mas acredita-se que a aplicação dessas metodologias auxiliou na melhoria da manutenção do produto.

Por fim, foi sugerido um ciclo de trabalho utilizando a metodologia Scrum, utilizando os requisitos gerados pelo processo para retroalimentar o backlog. Como a forma de trabalho de uma corporação é de caráter específico da mesma, o Scrum tornou-se apenas uma orientação, não uma obrigatoriedade. Como a empresa piloto já utilizava uma adaptação do Scrum para gerir o trabalho, não é possível afirmar a assertividade da utilização desta metodologia ágil para aplicação do processo, apenas que ela foi utilizada com sucesso na aplicação candidata.

Acredita-se que este estudo atingiu plenamente o objetivo por ele proposto: o desenvolvimento e aplicação de um processo de conversão de sistemas legados procedurais para orientado a objetos através da utilização da arquitetura MVC. Não é possível, no entanto, afirmar que este processo pode ser utilizado em múltiplos cenários com configurações diversas, devido à existência de vertentes não exploradas que podem negar a efetividade do processo, ou pelo menos, surtir um efeito menor do que o esperado.

7 CONCLUSÃO

Diante de um mundo que trabalha cada vez mais rápido em busca da otimização e diferenciação, dentro de um mercado que fica mais exigente a cada lançamento e opção oferecida, é necessário parar e refletir sobre o que está sendo ofertado e se essa oferta condiz com os altos padrões esperados pelos clientes.

É de conhecimento geral que, desde a concepção do paradigma procedural nos anos 70, diversos produtos de software chegaram ao mercado e diversos deles, graças a uma boa gerência, conseguiram se manter relevantes no ramo em que operam. No entanto, não é possível contentar-se apenas com a manutenção desta relevância para continuar fornecendo um produto competitivo, estável e funcional.

Através da realização de um estudo teórico, foi possível verificar as inúmeras opções que existem no mercado de desenvolvimento de softwares para criação de produtos que atinjam as mais diversas áreas do mercado. Este estudo buscou apresentar um processo para viabilizar a readequação de sistemas legados desenvolvidos proceduralmente em orientados a objetos, por meio de uma aplicação de conceitos da arquitetura MVC.

Além da reengenharia do código-fonte, foi fundamental a anexação de requisitos e casos de uso ao processo, visto que existem diversas empresas no ramo que não possuem documentação do sistema ou, pelo menos, de suas funcionalidades mais antigas. Através destes artefatos, é possível estabelecer um paralelo sobre o que o sistema está programado para fazer e o que ele deve fazer sob uma ótica de negócio.

A proposta de aplicação do processo é que seja possível conhecer o software que se está reescrevendo, facilitando futuramente a manutenção do software através da utilização dos requisitos documentados, casos de uso e roteiros de testes desenvolvidos para a conversão do software legado.

Apesar da existência de limites do que é possível ou não readequar, definidos pelos paradigmas, pelos ambientes de desenvolvimento ou banco de dados, este projeto buscou apresentar uma alternativa à completa reinvenção do produto, que pode apresentar riscos e altos custos à empresa.

Tomou-se por objetivo, então, implementar o processo de conversão em um cenário real, para que fosse possível avaliar como o mesmo se comportaria com problemas e dificuldades reais.

Durante a aplicação do processo, foi percebido que cada cenário, cada empresa, possui um conjunto de particularidades específicas somente a ela. Cada conjunto de particularidades pode afetar diretamente os custos de tempo e dinheiro envolvidos em aplicar o processo de conversão.

No cenário piloto, por já encontrar-se em um ambiente orientado a objetos, não foi necessária a alteração da linguagem de programação, mas, em contrapartida, foi necessário desenvolver pacotes que fornecessem subsídios para a aplicação do padrão MVC. Da mesma maneira ocorreu com os pacotes para testes de unidade, que não existiam na ferramenta.

Com isso, percebeu-se que existe certa dificuldade em estabelecer um framework de trabalho absoluto, com prazos e custos completamente assertivos. Apesar destas dificuldades, foi possível realizar a conversão do sistema legado procedural para um sistema orientados a objetos organizado pela arquitetura MVC.

Assim, ao fim deste estudo, é possível afirmar que aplicar um processo de conversão em sistemas legados é uma alternativa válida para empresas, embora possa existir a necessidade de flexibilizar etapas do processo, removendo partes ou adicionando-as onde necessário.

Embora não seja possível afirmar que o processo proposto neste estudo seja um modelo ideal, devido à alta variabilidade de cenários encontrados, acredita-se ter estabelecido um bom ponto de partida para que, com base neste estudo, seja possível desenvolver processos derivados que se adaptem às situações a ele apresentadas.

A principal confirmação que este estudo promoveu foi da importância de estabelecer uma base escalável para o software legado. Caso a empresa encontre-se em um momento em que é possível desviar pessoal e tempo para a aplicação de um processo de conversão, o produto final é recompensador para a sobrevida do software no mercado.

Além da apresentação do processo de conversão e de uma análise da aplicação do mesmo em um cenário real, este estudo promoveu um movimento na empresa piloto para desenvolver software com um padrão elevado – o padrão de arquitetura MVC – e a orientação a objetos foi adotada como nova forma de desenvolvimento e, atualmente, o processo está sendo aplicado em três módulos do sistema, sob responsabilidade de desenvolvedores diferentes, o que irá fomentar ainda mais a validade do processo.

Contudo, existem vertentes ainda não exploradas por este estudo, que podem ser continuadas em trabalhos futuros, como a aplicação do processo em um cenário onde o ambiente de desenvolvimento e até o banco de dados sejam alterados, e em que seja possível realizar um acompanhamento da adaptação e treinamento da equipe em uma nova linguagem, tal qual o comportamento do processo. Além disso, seria de grande validade a maturidade do processo para outros paradigmas e padrões de arquitetura, expandindo para a cada vez mais presente nuvem.

REFERÊNCIAS

BARANAUSKAS, Maria Cecília Calani. **Procedimento, função, objeto ou lógica? Linguagens de programação vistas pelos seus paradigmas**. Disponível em: <http://200.17.137.109:8081/novobsi/Members/josino/paradigmas-de-programacao/2012.1/(Leitura_e_Resenha)_ArtigoDiscussaoParadigmas.pdf>. Acesso em: 04 abr. 2017.

BECK, M Kent. **Refactoring: Improving the Design of Existing Code** 1 ed., Vol. 1. Addison-Wesley Professional, 1999.

BECK, M Kent; FOWLER, Martin. **Planning Extreme Programming**. Addison-Wesley Professional, 2001.

BERTOL, Omero Francisco. **Introdução à Programação Estruturada**. 2012. Disponível em: <http://www.devmedia.com.br/introducao-a-programacao-estruturada/24951>. Acesso em 16 maio 2017.

COHN, M. **User Stories Applied**: For Agile Software Development, Addison-Wesley Professional, 2004.

DEITEL, Paul; DEITEL, Harvey. **Como programar**. 6. ed. São Paulo: Pearson Education, 2011.

ELIAS, Diogo**. A abordagem Top-Down e Bottom-Up no Data Warehouse**. 2015. Disponível em: <https://corporate.canaltech.com.br/materia/banco-de-dados/a-abordagem-top-down-e-bottom-up-no-data-warehouse-21108/>. Acesso em 4 jun. 2017.

ERLANG PROGRAMMING LANGUAGE. Disponível em: <https://www.erlang.org/>. Acesso em 17 maio 2017.

FOGGETTI, Cristiano. **Gestão ágil de projetos**. 1. ed. São Paulo: Pearson Education. 2015.

FOWLER, Martin. **GUI Architectures**. 2006. Disponível em: <https://martinfowler.com/eaaDev/uiArchs.html>. Acesso em 17 maio 2017.

GALLOWAY, Jon et al. **Professional ASP** – NET MVC 5. Nova Jersey: John Wiley & Sons, Inc, 2014.

GUSMÃO, Gustavo. **Usada por bancos e esquecida por programadores, lingugem ganha fôlego com tecnologia de conversão para Java**. 2014. Disponível em: < http://exame.abril.com.br/tecnologia/usada-por-bancos-e-esquecida-por-programadores-linguagem-de-programacao-ganha-folego-com-tecnologia/>. Acesso em 17 maio 2017.

HAUTSCH, Oliver. **O que é engenharia reversa?** 2009. Disponível em: <https://www.tecmundo.com.br/pirataria/2808-o-que-e-engenharia-reversa-.htm>. Acesso em 16 maio 2017.

HEMEL, Zef. **O modelo de implantação da Netflix: DevOps na veia e resiliência extrema**. 2013. Disponível em: <https://www.infoq.com/br/news/2013/07/modelo-netflix-devops>. Acesso em 16 jun. 2017.

KRASNER, Glenn E.; POPE, Stephen T. **A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System**. Disponível em: <http://heaveneverywhere.com/stp/PostScript/mvc.pdf>. Acesso em 03 abr. 2017.

LAMIM, Jonathan. **MVC** – O padrão de arquitetura de software. 2012. Disponível em: <https://www.oficinadanet.com.br/artigo/1687/mvc_-_o_padrao_de_arquitetura_de_software>. Acesso em 15 maio 2017.

LIBARDI, Paula L. O.; BARBOSA, Vladimir. **Métodos Ágeis**. 2010. Dissertação – Universidade Estadual de Campinas. Disponível em: <http://www.ft.unicamp.br/liag/Gerenciamento/monografias/monogafia_metodos_ageis.pdf>. Acesso em 17 maio 2017.

LIMA, Erick Ednaldo de. **POO x PE**: Vantagens e Diferenças. 2017. Disponível em: <https://pt.linkedin.com/pulse/m%C3%A1gica-da-programa%C3%A7%C3%A3o-extruturada-erick-ednaldo-de-lima>. Acesso em 28 set. 2017.

LONGO, Hugo Estevam Romeu; SILVA, Madalena Pereira. **A utilização de histórias de usuários no levantamento de requisitos ágeis para o desenvolvimento de software**. 2014. Disponível em: <http://incubadora.periodicos.ufsc.br/index.php/IJKEM/article/view/2712>. Acesso em 2 out. 2017.

MEDEIROS, Higor. **Princípios da Engenharia de Software**. Disponível em: <http://www.devmedia.com.br/principios-da-engenharia-de-software/29630>. Acesso em 6 jun. 2017.

MTSYSTEMS. Disponível em: <https://www.mtsystems.com/>. Acesso em 01 out. 2017.

MULLER, Nicolas. **Qual a diferença entre programação estruturada e programação orientada a objetos?** 2016. Disponível em: <https://www.oficinadanet.com.br/post/14463-qual-a-diferenca-entre-programacao-estruturada-e-programacao-orientada-a-objetos>. Acesso em 16 maio 2017.

NOGUEIRA, Antônio Sérgio. **Programando Em Python Do Básico à Web**. 1. ed. Joinville: Clube de Autores, 2008.

PRESSMAN, Roger; MAXIM, Bruce. **Engenharia de Software**. 8. ed. Porto Alegre: AMGH, 2016.

PUGA, Sandra; RISSETTI, Gerson. **Lógica de programação e estrutura de dados com aplicações em Java**. São Paulo: Pearson Education do Brasil, 2004.

REENSKAUG, Trygve; COPLIEN, James O. **The DCI Architecture**: A New Vision of Object-Oriented Programming. 2009. Disponível em: <http://www.artima.com/articles/dci_vision.html>. Acesso em 14 maio 2017.

SANTANA, Vagner Figuerêdo de. **Padrões Arquiteturais de Sistemas**. 2011. Disponível em: <https://pt.slideshare.net/santanavagner/padroes-arquiteturais-de-sistemas>. Acesso em 29 set. 2017.

SANTOS, Igor Abílio Santana. **Programação Orientada a Objetos versus Programação Estruturada**. 2015. Disponível em: <http://www.devmedia.com.br/programacao-orientada-a-objetos-versus-programacao-estruturada/32813>. Acesso em 18 maio 2017.

SANTOS, Wagner Leal dos**. Um processo de migração de sistema legado functional para orientado a objetos direcionado por indicadores de qualidade**. 2007. Dissertação (Mestrado em Sistemas Digitais) – Escola Politécnica, Universidade de São Paulo, São Paulo, 2007.

SCHWABER, Ken. **Agile Project Management with Scrum**. 1. ed. Albuquerque: Microsoft Press. 2004.

SCALFINI, Charles. **Goodbye, Object Oriented Programming**. 2016. Disponível em: <https://medium.com/@cscalfani/goodbye-object-oriented-programming-a59cda4c0e53>. Acesso em 17 maio 2017.

SEIBEL, Peter. **Coders at Work**: Reflections on the Craft of Progamming. 1. ed. New York, Ny.: Springer-Verlag. 2009.

SOARES, Michel dos Santos. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**. 2004. Disponível em: <http://www.dcc.ufla.br/infocomp/index.php/INFOCOMP/article/view/68>. Acesso em 15 maio 2017.

SOBRINHO, Romeu. **O que é SOA (Arquitetura Orientada a Serviços)?**. 2011. Disponível em: <https://www.oficinadanet.com.br/artigo/desenvolvimento/o_que_e_soa_arquitetura_orientada_a_servicos>. Acesso em 16 jun. 2017.

SOMMERVILLE, Ian**. Engenharia de Software**, 497-513, Addison-Wesley. 2011.

TAVARES, Leandro. **Boas práticas de programação**. 2014. Disponível em: < http://www.devmedia.com.br/boas-praticas-de-programacao/31163>. Acesso em 10 jun. 2017.

VAN ROY, Peter. **Programming Paradigms for Dummies**: What Every Programmer Should Know. Disponível em: <https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf>. Acesso em 15 maio 2017.

VENTURA, Plínio. **O que é requisito funcional**. 2016. Disponível em: <http://www.ateomomento.com.br/o-que-e-requisito-funcional/>. Acesso em 5 jun. 2017.

WEIDERMAN, Nelson H.; Bergey, John K.; Smith, Dennis B.; & Tilley, Scott R. **Approaches to Legacy System Evolution** (CMU/SEI-97-TR-014). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University. 1997.

WEGNER, Peter. **Concepts and Paradigms of Object-Oriented Programming**. 1990. Disponível em: <http://www.cs.technion.ac.il/users/yechiel/Hassava/OOP-material/1990--p7-wegner.pdf>. Acesso em 04 abr. 2017.