UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Marcos Luís Weiss

USO DA ARQUEOLOGIA DE SOFTWARE PARA CONSTRUÇÃO DE CASOS DE TESTE EM SISTEMAS LEGADOS

São Leopoldo

2017

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Marcos Luís Weiss

USO DA ARQUEOLOGIA DE SOFTWARE PARA CONSTRUÇÃO DE CASOS DE TESTE EM SISTEMAS LEGADOS

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Especialista em Engenharia de Software, pelo curso de Pós-Graduação Lato Sensu em Engenharia de Software da Universidade do Vale do Rio dos Sinos – UNISINOS.
Orientador: Profa. Dra. Margrit Reni Krug

São Leopoldo

2017

Uso da arqueologia de software para construção de casos de teste em sistemas legados

Marcos Luís Weiss

Unidade Acadêmica de Pesquisa e Pós-Graduação
Universidade do Vale do Rio dos Sinos (UNISINOS) – São Leopoldo – RS – Brasil

marcoslweiss@gmail.com

Resumo: *Sistemas legados são, em geral, softwares muito antigos e mal documentados, mas que precisam ser mantidos por serem vitais para o negócio da organização. A arqueologia de software pode ajudar nesse processo que envolve a compreensão do sistema legado e o provimento de informações para a produção de casos de teste. Neste artigo é apresentado um estudo de caso em que se verifica a viabilidade de se gerar casos de teste a partir das informações disponibilizadas pela arqueologia de software. São apresentadas também técnicas e ferramentas que fazem a análise de código-fonte que auxiliam no processo de extração de informações.*

Abstract: *Legacy systems are usually very old and poorly documented software, but they need to be maintained because they are vital to the organization's business. Software archeology can help in this process that involves understanding the legacy system and providing information for the production of test cases. This paper presents a case study in which the feasibility of generating test cases from the information provided by software archeology is verified. Techniques and tools that do source code analysis are also presented and help in the process of extracting information.*

1. Introdução

Quando organizações investem em software, esperam alongar ao máximo o seu ciclo de vida. Durante a sua existência, o software passa por fases evolutivas até o ponto em que a organização, por questões estratégicas, considera parar de investir na sua evolução. No entanto, esses softwares, em geral, permanecem ativos e necessitam ser mantidos para continuarem funcionais, passando a ser reconhecidos dentro da organização como sistema legado. Com o passar do tempo esses softwares sofrem alterações por mudanças de requisitos ou por questões tecnológicas e acabam se tornando muito diferentes da concepção original, muitas vezes tornando-se difíceis de serem compreendidos e mudados (SOMMERVILLE, 2011).

Nesse contexto, a arqueologia de software surge como um conjunto de técnicas e ferramentas para auxiliar na extração de informações nos sistemas legados, permitindo assim uma melhor compreensão do código produzido e facilitando a manutenção do mesmo (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2011). Hunt e Thomas (2002) afirmam que os pesquisadores que trabalham neste campo lidam, como arqueólogos, com artefatos restantes do passado. A principal base da arqueologia de software é a necessidade de os desenvolvedores, ao manter um produto de software, terem de lidar com o código que foi modificado em possivelmente muitos pontos ao longo do tempo por muitos desenvolvedores diferentes (HUNT; THOMAS, 2002).

Manter um sistema legado funcional implica em um processo de teste eficaz e que possibilite entregar as alterações realizadas com um mínimo de defeitos. No entanto, realizar testes em sistemas legados pode ser tornar inviável uma vez que não é uma tarefa fácil identificar requisitos em sistemas com pouca ou nenhuma documentação disponível. Segundo Bastos (2007), os requisitos de negócio se constituem nos documentos básicos para definir objetivos de teste. Molinari (2008), por sua vez, afirma que os testes são importantes para "ajudar na otimização da tecnologia e das aplicações que suportam o negócio da empresa". Assim, a arqueologia de software pode se tornar uma alternativa para identificar os requisitos em sistemas legados e viabilizar a criação de casos de teste que, segundo Molinari (2008), representam aquilo que deve ser testado em termos de detalhe.

A partir do contexto apresentado, este trabalho teve como objetivo propor a melhoria da qualidade da manutenção em sistemas legados através do uso da arqueologia de software, respondendo à seguinte questão de pesquisa: Como obter informações consistentes para a construção de casos de teste em sistemas legados? Para atingir esse objetivo, este trabalho propôs os seguintes objetivos específicos:

1. Identificar os principais problemas e dificuldades na apuração de informações em sistemas legados e as estratégias de obtê-las a partir dos artefatos produzidos.
2. Analisar ferramentas para extração de informações em sistemas legados.
3. Identificar técnicas e ferramentas para a construção de casos de teste em sistemas legados.
4. Validar, através de um estudo de caso, se as informações obtidas possibilitaram a construção de casos de teste.

Na próxima seção são apresentados os fundamentos teóricos relacionados a sistemas legados, arqueologia de software e testes de software, e que servirão de subsídio para o desenvolvimento do estudo de caso. Na sequência, serão detalhados os métodos e procedimentos de pesquisa utilizados para a realização deste trabalho, seguido pela apresentação e análise do estudo de caso. Por fim, serão apresentadas as considerações finais.

2. Fundamentação teórica

2.1. Sistemas legados

De acordo com Nascimento (2014), muitos dos processos de negócio das organizações estão implementados em sistemas legados. São sistemas de software que geralmente custaram muito dinheiro e muitas vezes tem uma vida útil longa para que as organizações tenham um retorno do seu investimento, necessitando que sejam ampliados e adaptados às mudanças inerentes ao negócio e ao ambiente em que está inserido para se manterem funcionais (SOMMERVILLE, 2011). Para Nascimento (2014), os sistemas legados possuem pouca ou nenhuma documentação, foram desenvolvidos com tecnologias obsoletas e os processos de negócios foram programados implicitamente no seu código fonte. Paradauskas e Laurikaitis (2006 apud PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2011, p. 2) afirmam que "um sistema de informação legado é qualquer sistema de informação que resista significativamente à

modificação e evolução para atender aos novos requisitos de negócios em constante mudança". Hunt e Thomas (2002), por sua vez, afirmam que "o código torna-se legado logo após ter sido escrito".

Para Birchall (2016), os sistemas legados têm algumas caraterísticas comuns:

- Antigos: É necessário que o sistema exista por alguns anos para se tornar de fato difícil de ser mantido. Nesse período várias gerações de mantenedores contribuíram para o seu crescimento e o conhecimento sobre a ideia original do sistema e as intenções do mantenedor anterior são perdidos.
- Grandes: Quanto maior o sistema, mais difícil de manter. Isso porque há um maior volume de código e, consequentemente, o sistema se torna mais propenso a erros. Além disso, aumenta a possibilidade de partes do código serem afetadas por alterações. Sistemas grandes também são mais difíceis e arriscados para substituir, por isso tendem a se tornarem sistemas legados.
- Herdados**:** Esses sistemas geralmente são herdados de um desenvolvedor ou equipe anterior que já não se encontra mais na organização. Isso significa que mantenedores atuais não tem como saber a razão do código funcionar de uma determinada maneira e, muitas vezes precisam adivinhar as intenções e suposições das pessoas que escreveram o código.
- Pouco documentados: Manter uma documentação atualizada deveria ser um processo vital para um sistema. No entanto, o que se observa é que ao longo do tempo as várias gerações de desenvolvedores vão deixando a atualização em segundo plano até o ponto que a documentação está totalmente desatualizada ou, em casos mais graves, nem se encontra mais disponível.

Birchall (2016) ressalta que o fato de um determinado sistema atender a um dos critérios não significa que ele deva ser tratado como sistema legado, citando como exemplo o *kernel* do Linux. Mesmo sendo muito antigo e também muito grande, ele consegue manter uma qualidade alta.

Conforme Birchall (2016), muitos sistemas legados são tão complexos e mal documentados que até mesmo as equipes responsáveis em os manter tem dificuldade em compreender eles. Em função disso e ponderando os riscos envolvidos em uma mudança, evitam fazer alterações desnecessárias e consideram fortemente o estado atual como uma opção mais segura. Osborne e Chikofsky (1990 apud PRESSMAN, 2011, p. 663), por sua vez, relatam que muitos dos softwares atuais têm em média de 10 a 15 anos e durante esse período passaram por migrações de plataformas, ajustes para se adequarem a novos hardwares e aperfeiçoados para atender a novas necessidades de usuários, sem a devida atenção na arquitetura geral. Isso resulta em estruturas mal projetadas, mal codificadas, de lógica pobre e mal documentadas que equipes atuais precisam manter. Pressman (2011) conclui que, além dos problemas relatados, a mobilidade de profissionais se constitui num problema adicional uma vez que é provável que equipes ou profissionais responsáveis pelo trabalho original não estejam mais na organização.

Muitas vezes um sistema legado não possui testes e é difícil testar (BIRCHALL, 2016). Além disso, o código em geral é inflexível, significando que para realizar mudanças simples é

preciso muito trabalho. Pressman (2011) complementa dizendo que, apesar de alguns sistemas legados terem uma arquitetura razoavelmente sólida, seus módulos individuais foram codificados de um modo que se torna difícil de entender, testar e manter tais sistemas.

Mesmo num cenário problemático em que um sistema legado possa existir, a organização busca manter esse sistema funcional e isso geralmente implica em custos. Pressman (2011) afirma que a organização se dá conta de que, ao longo do tempo, a manutenção dos programas está exigindo mais tempo e dinheiro do que a criação de novas aplicações, implicando muitas vezes em gastos que consomem de 60% a 70% de todos os recursos com a manutenção de software. Nesse cenário, Sommerville (2011) sustenta que a organização precisa fazer uma avaliação dos seus sistemas legados a fim de obter um melhor retorno de seus investimentos e decidir entre quatro opções estratégicas aquela que seja mais adequada:

1. Descartar completamente o sistema;
2. Deixar o sistema inalterado e continuar com a manutenção regular;
3. Reestruturar o sistema para melhorar a sua manutenibilidade;
4. Substituir a totalidade ou parte do sistema por um novo.

Dependendo da estratégia adotada, algumas considerações são necessárias e estão resumidas na Tabela 1.

Tabela 1. Mudanças, benefícios e riscos para um sistema legado (Birchall, 2016)

Mudanças	Benefícios	Riscos	Ações requeridas
Remover uma funcionalidade antiga	● Desenvolvimento fácil ● Melhor desempenho	● Alguém ainda está usando a funcionalidade	● Checar logs de acesso ● Questionar usuários
Refatoração	● Desenvolvimento fácil	● Regressão acidental	● Revisão de código ● Testes
Atualizar uma biblioteca	● Correção de *bugs* ● Melhorias no desempenho	● Regressão devido a alteração no comportamento da biblioteca	● Revisar os registros de mudanças ● Revisão do código da biblioteca ● Testar manualmente as principais funcionalidades

Sommerville (2011) destaca que "a manutenção de software é o processo geral de mudança em um sistema depois que ele é liberado para uso". Essas mudanças podem ser simples correções de erros de codificação até mudanças mais extensas para correção de erros de projeto. Segundo Sommerville (2011) existem três tipos diferentes de manutenção de software: correção de defeitos, adaptação ambiental e adição de funcionalidade. Feathers (2013), por sua vez, relaciona quatro razões principais que levam à necessidade de alteração em um software:

1. Inclusão de uma funcionalidade;
2. Correção de um bug;
3. Melhoria do projeto;
4. Otimização do uso de recursos.

Um aspecto importante abordado por Feathers (2013) é como o comportamento do software é afetado pelas alterações. A inclusão de uma funcionalidade, por exemplo, pode implicar na maneira como o sistema se comporta. Quando se muda algo num sistema, geralmente queremos manter seu comportamento intacto. E isso é mais crítico em sistemas legados, em que muitas vezes se procura melhorar a sua estrutura enquanto que a sua funcionalidade deve permanecer igual. Feathers (2013) frisa que "é preciso assegurar que a pequena quantidade de itens que mudarmos seja alterada corretamente". No entanto, geralmente não é possível saber o quanto do comportamento está correndo risco quando um pedaço do software está sendo alterado.

Uma das maneiras de melhorar um software sem alterar seu comportamento é chamada de refatoração (FEATHERS, 2013). A refatoração consiste na ideia de tornar um software mais fácil de ser mantido através de uma série de pequenas modificações estruturais. Essas modificações são suportadas por testes a fim de garantir que não haja nenhuma mudança funcional. A otimização, segundo Feathers (2013), é semelhante à refatoração. No entanto, o seu objetivo é diferente. Ela visa a melhoria de algum recurso, geralmente tempo ou memória.

2.2. Arqueologia de software

Muitas vezes desenvolvedores herdam sistemas que não conhecem ou então são alocados em algum projeto para continuar com a sua manutenção e desenvolvimento. Esses sistemas geralmente são grandes, com milhares de linhas de código, documentação defasada ou inexistente e os desenvolvedores originais talvez nem estejam mais presentes na organização. E é nesse cenário, facilmente encontrado nas organizações, que a arqueologia de software entra como um processo para obter informações de um sistema antigo a partir de artefatos como documentos, diagramas e códigos-fonte.

Na arqueologia real, em que são realizadas pesquisas, escavações e análises para investigar alguma situação, busca-se entender o que se está olhando e como tudo se encaixa (HUNT; THOMAS, 2002). Hunt e Thomas (2002) também esclarecem que, para fazer isso, deve-se ter o cuidado para preservar os artefatos encontrados e respeitar e compreender as forças culturais que os produziram.

A ideia de relacionar a arqueologia com a manutenção de software se deu num workshop sobre Arqueologia de Software ocorrido na OOPSLA 2001 (*Conference on Object-Oriented Programming, Systems, Languages, and Applications*). Esse workshop foi organizado por Andy Hunt, Dave Thomas, Brian Marick e Ward Cunningham e foram os primeiros a fundamentar o uso do conceito de arqueologia (IZQUIERDO-CORTAZAR, 2009, p. 3),

> Arqueologia de software é uma metáfora útil: os programadores tentam entender o que estava na mente de outros desenvolvedores usando apenas os artefatos deixados para trás. Eles são prejudicados porque os artefatos não

> foram criados para se comunicar com o futuro, porque apenas uma parte do que foi originalmente criado foi preservada e porque as relíquias de diferentes épocas estão misturadas.

Quando se lida com código antigo, é comum encontrar várias adversidades que dificultam a compreensão do código. Hunt e Thomas (2002), em seu artigo clássico sobre arqueologia de software, comentam que o problema inicia quando desenvolvedores adiam consertos, o que geralmente leva esse problema para que gerações posteriores de desenvolvedores resolvam. Além disso, nomes e comentários enganosos ou incorretos contribuem para entender mal o código que está sendo lido. Apesar do cenário pouco favorável à compreensão do código, ainda é possível extrair informações importantes a partir dele. Segundo Hunt e Thomas (2002, p. 23),

> Escavando abaixo dessas camadas de código bagunçado, mal concebido, desnecessariamente complicado e remendo em cima de remendo, você ainda pode ver a forma do sistema original e obter informações sobre as mudanças que foram necessárias ao longo dos anos.

De acordo com Hunt e Thomas (2002), os pesquisadores que trabalham neste campo lidam, como arqueólogos, com artefatos restantes do passado. Eles tentam compreender a partir dos artefatos encontrados o que estão olhando, ou seja, precisam compreender as forças culturais e de civilização que produziram tais artefatos (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2017). Os arqueólogos, em geral, não questionam o quão estúpida era uma cultura particular. Por isso, decisões relativas ao código que hoje nos seriam estranhas, podem parecer perfeitamente razoáveis para os desenvolvedores da época. Assim, compreender o que eles estavam pensando é fundamental para entender como e porque eles escreveram o código da maneira que eles fizeram (HUNT; THOMAS, 2002).

Apesar do conceito de arqueologia de software ter sido concebido tendo em mente grandes sistemas, ele é válido para qualquer tipo de software, independente da idade e tamanho (IZQUIERDO-CORTAZAR, 2009). A principal ideia desse conceito é que, para manter um produto de software, os desenvolvedores precisam lidar com um código que foi modificado em vários pontos ao longo do tempo por muitos desenvolvedores diferentes. No entanto, apesar deste código ainda estar em uso, ele pode ser diferente daquele originalmente concebido e os processos de negócio e sistemas de informação podem não estar mais alinhados (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2017). Pérez-Castillo, Guzmán e Piattini (2017) afirmam também que modificações no código-fonte em geral não tem qualquer efeito sobre os processos de negócio originais visto que sistemas legados embutem um montante significativo de conhecimento de negócios que não é definido no conjunto original de processos de negócio. E esse conhecimento, segundo Paradauskas e Laurikaitis (2006 apud PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2011, p. 4), pode não existir em nenhum outro lugar. Isso pode dificultar o gerenciamento de processos de negócio devido ao fato dos sistemas legados terem evoluído de forma independente daqueles processos originalmente definidos na organização (HEUVEL, 1990 apud PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2017, p. 2).

Segundo Nascimento (2014), “a principal fonte de informações sobre um sistema legado é o código fonte”. O autor afirma também que, ao analisar o código-fonte de um sistema,

é possível extrair algumas informações mais básicas sobre ele como rotinas, estruturas de dados, estruturas de controle de fluxo e as relações existentes entre seus componentes.

A literatura aborda duas técnicas de análise do código-fonte que podem ser utilizadas para recuperar informações e processos de negócios incorporados em sistemas legados: análise dinâmica e análise estática.

De acordo com Nascimento (2014), a análise dinâmica foca na execução do sistema e ela contempla um conjunto de técnicas e algoritmos de mineração de processos de negócio que são utilizados a fim de extrair o fluxo de informações executados na organização. Isso é realizado através de análises de arquivos de *log* no sistema legado (NASCIMENTO, 2014; PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2017).

Algumas propostas de análise dinâmica são encontradas na literatura. Difrancescomarino, Marchetto e Tonella (2009 apud PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2017, p. 6) apresentam uma proposta que aborda a descoberta de processos de negócio através da execução de interfaces gráficas em aplicações Web. Outra proposta é apresentada por Cai, Yang e Wang (2009 apud PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2017, p. 6). Nesta, os autores sugerem uma abordagem que combina a análise de requisitos (para prover um conjunto de casos de uso), com mineração de arquivos de *log* que são disparados por atores do sistema legado. Nascimento (2014) salienta que, de uma maneira geral, estas técnicas não são adequadas para sistemas legados que não estão preparados para gerar arquivos de *log*.

A análise estática, por sua vez, foca na análise do código-fonte. Devido ao fato de que os sistemas legados em geral não possuem documentação atualizada e foram desenvolvidas com tecnologias obsoletas, as técnicas de análise estática tem sido mais estudadas e empregadas na extração de conhecimento destes sistemas (NASCIMENTO, 2014). Para analisar estaticamente um código fonte são utilizadas ferramentas chamadas de "*parsers*", que analisam as instruções numa determinada linguagem de programação em busca de estruturas que forneçam informações relevantes sobre o negócio da organização (ERNST, 2003 apud NASCIMENTO, 2014, p. 22).

Existem várias propostas baseadas na análise estática do código-fonte. Uma dessas propostas é apresentada por Zou e Hung (2006 apud PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2017, p. 5), em que o código-fonte é analisado estaticamente e, então, é aplicado um conjunto de regras heurísticas para descobrir regras de negócio. Paradauskas e Laurikaitis (2006 apud PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2017, p. 5) apresentam um *framework* para recuperar conhecimento de negócios através da análise estática de dados armazenados em banco de dados. Ghose et al (2007 apud PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2017, p. 5), por sua vez, propõem a extração de informações a partir de pesquisas baseadas em texto no código-fonte e documentação. Pérez-Castillo, Guzmán e Piattini (2011 apud NASCIMENTO, 2014, p.24) apresentam técnicas que mudam o foco da análise estática para a identificação de processos de negócio que implementam as regras de negócio. Isso se dá por meio da descoberta de trechos de código-fonte que representam partes de um processo de negócio executado por um sistema legado.

Uma desvantagem dessas propostas que envolvem técnicas de análise estática é que elas ignoram a informação gerada em tempo de execução. Pérez-Castillo, Guzmán e Piattini (2017) salientam que "a principal fraqueza dessas técnicas é que a maior parte dos esforços são propostas *ad hoc*, que são desenvolvidas para plataformas particulares, tecnologias e contextos específicos". Segundo os autores, isso leva a uma falta de padronização e formalização que tem como consequência a dificuldade em automatizar tais técnicas em projetos de larga escala. Isso fica evidenciado, conforme estudo citado pelos autores, na quantidade expressiva de projetos baseados em técnicas de engenharia reversa que falham devido à falta de padronização e automação, podendo muitas vezes ultrapassar os custos previstos no projeto.

Para contornar isso, Pérez-Castillo, Guzmán e Piattini (2017) propõem um modelo baseado na modernização orientada à arquitetura (ADM - Architecture-Driven Modernization), uma iniciativa do Object Management Group (OMG) e que que defende a realização de processos de reengenharia, considerando princípios de desenvolvimento orientados por modelo ou seja, separando a lógica de negócios e aplicativos da tecnologia e plataforma em que está inserida (OMG, 2017). Através da criação de uma ferramenta de análise estática (MARBLE) que usa padrões de negócio para descobrir fragmentos de processos codificados em sistemas legados, o código-fonte é mapeado para modelos KDM (*Knowledge Discovery Metamodel)* que, num momento posterior, são analisados para descobrir padrões de negócio que podem ser modelados com BPMN (NASCIMENTO, 2014). Um exemplo é a transformação de comandos condicionais do tipo *if-then-else* em gateways XOR-Split da notação BPMN *(Business Process Modeling Notation)*. O padrão KDM também foi proposto pela iniciativa ADM e ela permite a representação e o gerenciamento do conhecimento extraído de todos os diferentes artefatos de software de sistemas legados (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2011). Dessa forma, o conhecimento legado é transformado aos poucos em processos de negócios.

A ferramenta MARBLE, proposta por Pérez-Castillo, Guzmán e Piattini (2011) foi implementada como um *plugin* do Eclipse e, através de um analisador Java, obtém o modelo do código que é transformado e integrado num repositório de acordo com o padrão KDM. Na sequência, o modelo KDM é transformado em modelos de processos de negócio. E, para isso, o MARBLE é dividido em quatro níveis de abstração com três transformações entre eles, conforme pode ser visto na Figura 1.

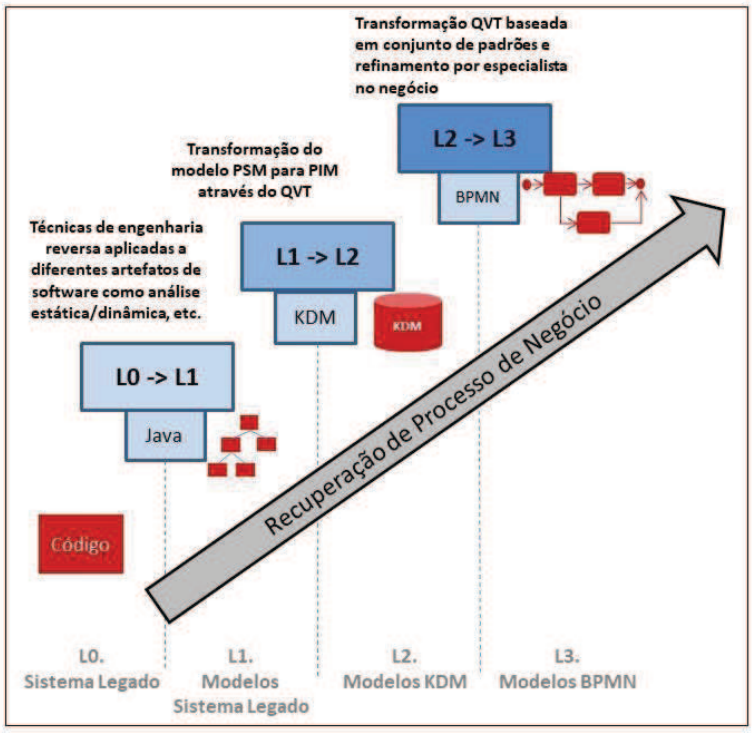

Figura 1. Transformações do MARBLE, segundo o modelo KDM (adaptado de PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2011).

Os quatro níveis genéricos de abstração propostos no MARBLE são descritos a seguir (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2011):

- Nível L0: É o menor nível de abstração e representa o sistema de informação legado no mundo real como uma coleção de artefatos de software diferentes (por exemplo, código fonte, banco de dados, documentação).
- Nível L1: Este nível consiste em vários modelos específicos, ou seja, um modelo para cada artefato de software diferente envolvido no processo de arqueologia (por exemplo, código fonte, banco de dados, interfaces de usuário). Esses modelos são considerados Modelos Específicos da Plataforma (*Platform-Specific Models* - PSM), pois representam os artefatos de software de acordo com suas especificações ou plataformas.
- Nível L2: Consiste em um Modelo Independente de Plataforma (*Platform-Independent Model* - PIM) que representa a visão integrada do conjunto de modelos PSM em L1. O metamodelo KDM padrão é usado para este propósito, pois permite modelar todos os artefatos do sistema herdado de forma integrada e independente da tecnologia.
- Nível L3: É o nível mais alto de abstração e representa um modelo computacional independente do sistema. Ele retrata os processos de negócios recuperados do conhecimento sobre sistemas de informação legados representados no repositório KDM do nível L2. Os modelos de processos de negócio em L3 são representados de acordo com o modelo BPMN. Neste nível ocorre uma última transformação que é baseada num conjunto de padrões e cada padrão indica quais elementos devem ser construídos e como eles estão inter-relacionados no modelo de processo de negócios.

Segundo Pérez-Castillo, Guzmán e Piattini (2011), os modelos obtidos a partir dessas transformações representam um primeiro esboço do processo de negócios. Eles representam uma solução mais eficiente e menos propensa a erros para obter a arqueologia de processos de negócios do que redesenhar todos os processos a partir do zero.

2.3. Testes de software

Segundo Rios (2006), uma parcela significativa do orçamento de TI é dedicada à manutenção dos sistemas logo após os mesmos entrarem em produção. Em função disso, os testes passam a ter um papel importante nesse processo. Sommerville (2011) destaca que uma das finalidades do teste é "mostrar que um programa faz o que é proposto fazer e para descobrir os defeitos do programa antes do uso". O autor frisa ainda que o teste é parte de um processo que visa encontrar erros, anomalias ou informações sobre os atributos não funcionais de um software e que tem dois objetivos distintos:

1. Demonstrar que o software atende aos requisitos.
2. Identificar defeitos de software ocasionados por situações em que o software se comporta de maneira incorreta, indesejável ou de forma diferente das especificações.

O teste de software faz parte de um processo mais amplo que consiste na verificação e validação e que "objetivam verificar se o software em desenvolvimento satisfaz suas especificações e oferece a funcionalidade esperada pelas pessoas que estão pagando pelo software" (SOMMERVILLE, 2011). Essa afirmação é ratificada por Molinari (2008) ao declarar que "o maior propósito das atividades de verificação e validação é garantir que o projeto do software, a codificação e a documentação 'encontrem' todos os requerimentos que foram definidos".

Um software é testado a partir de duas perspectivas diferentes (PRESSMAN, 2011):

1. Teste caixa branca: visam garantir que as operações internas estão sendo realizadas de acordo com as especificações e que todos os componentes internos tenham sido adequadamente exercitados.
2. Teste caixa preta: Não se preocupa com a lógica interna, focando os testes mais na interface do software e no atendimento às especificações dos requisitos.

Ao testar um software, uma das primeiras atividades a ser realizada é a análise do software e definir quais itens deverão ser testados. A partir disso, para cada item devem ser criados casos de teste. Segundo Sommerville (2011), casos de teste são especificações das entradas para o teste e da saída esperada. Molinari (2008) e Pezzè e Young (2008) acrescentam que casos de teste são um conjunto de entradas, condições de execução e resultados esperados para um objeto em particular, com critério de sucesso/falha. Rios (2006) complementa afirmando que o caso de teste desce ao nível de detalhe de campos de formulário, arquivos, telas, páginas e outros, e que cada item de teste deve ter um critério de avaliação e de teste.

Segundo Pezzè e Young (2008), "a especificação de casos de teste é um requisito que deve ser satisfeito por um ou mais casos de teste reais". Quanto à documentação do caso de

teste, ela deve compreender uma série de dados (BASTOS, 2007). São eles: um identificador do caso de teste, um estado inicial, uma sequência de operações de teste, dados de entrada, ambiente necessário para execução do teste, procedimentos especiais, dependências – requisitos e/ou outros casos de teste e os resultados esperados. Pezzè e Young (2008) afirmam que um caso de teste pode incluir, além dos dados de entrada, outros elementos relevantes para a execução e que podem contribuir para o comportamento do programa como, por exemplo, uma interrupção, corte de energia, injeção de falhas, entre outros.

2.3.1. Testes em sistemas legados

Rios (2006) ressalta que os testes realizados durante o desenvolvimento do software tendem a serem os mesmos usados durante a manutenção. No entanto, quando se fala em sistemas legados a realidade é um pouco diferente. Muitos sistemas legados não possuem testes e geralmente foram escritos sem ter isso em mente, o que torna mais difícil a inclusão de testes nestes sistemas (BIRCHALL, 2016). Além disso, implementar novos recursos ou alterações no comportamento desses sistemas torna-se mais penoso pois, mesmo pequenas mudanças, podem envolver a edição do código em muitos lugares. Dessa forma, segundo Birchall (2016), os testes passam a ter uma importância ainda maior, já que cada uma dessas edições precisa ser testada. E isso, às vezes, precisa ser feito manualmente.

Escrever testes para sistemas legados geralmente implica em preservar comportamento ao realizar alterações (FEATHERS, 2013). O autor ressalta que buscar informações em documentos de requisitos para a partir daí escrever testes pode não ser uma boa abordagem. Isso porque em quase todo sistema legado o que o sistema faz é mais importante do que o que deveria fazer. Dessa forma, Feathers (2013) indica escrever testes que ajudem a fazer alterações de maneira mais determinística. Outro ponto importante que facilita as alterações é a eliminação das dependências, considerado por Feathers (2013) como um dos problemas mais críticos encontrados em sistemas legados.

Birchall (2016) enfatiza que o código legado em geral é inflexível e que mudanças simples podem resultar em muito trabalho. Essa situação pode ser melhorada com o uso da refatoração. Feathers (2013), por sua vez, sugere um algoritmo que pode ser usado quando uma alteração deve ser feita no código legado:

1. Identificar os pontos de alteração;
2. Encontrar os pontos de teste;
3. Eliminar dependências;
4. Escrever testes, e;
5. Fazer alterações e refatorar.

Os testes unitários são, na visão de Feathers (2013), um dos componentes mais importantes no trabalho com código legado, pois fornecem um *feedback* durante o desenvolvimento e permitem a refatoração com mais segurança. Além disso, preenchem lacunas que os testes maiores deixam passar e possibilitam o teste de trechos de código de forma

independente, além de proporcionar a localização de erros mais rapidamente. Testes maiores que envolvem áreas mais amplas de código também são importantes. No entanto, Feathers (2013) reitera que testes mais amplos em sistemas legados apresentam alguns problemas devidos, principalmente, ao tamanho desses sistemas: localização de erros, tempo de execução e cobertura.

3. Metodologia

Este capítulo apresenta a metodologia de estudo utilizada para o desenvolvimento do trabalho. A sua realização compreendeu as seguintes etapas:

1. Levantamento bibliográfico;
2. Análise de ferramentas para extração de informações em sistemas legados;
3. Seleção de sistema legado para aplicação das técnicas abordadas;
4. Definição dos participantes do estudo de caso;
5. Realização do estudo de caso, e;
6. Análise dos dados coletados.

O estudo de caso foi o instrumento utilizado para o desenvolvimento do trabalho e os conceitos apresentados na revisão bibliográfica forneceram os subsídios necessários para a sua condução. A realização desse estudo de caso permitiu ter, ao seu término, os elementos necessários para avaliar a viabilidade de uso da arqueologia de software como instrumento para construir casos de teste em sistemas legados.

3.1. Delineamento da pesquisa

Segundo Flick (2009), a pesquisa qualitativa é um tipo de pesquisa que tem por objetivo abordar o mundo fora dos contextos especializados de pesquisa, como os laboratórios, a fim de entender, descrever e, às vezes, explicar os fenômenos sociais a partir da análise de experiências de indivíduos ou grupos, do exame das interações e comunicações que estejam se desenvolvendo e da investigação de documentos. Flick (2009) complementa afirmando que "uma grande quantidade de pesquisa qualitativa se baseia em estudos de caso ou em séries desses estudos, e, com frequência, o caso (sua história e complexidade) é importante para entender o que está sendo estudado". Dessa forma, a pesquisa objeto desse trabalho teve um enfoque qualitativo e se deu em duas etapas: revisão bibliográfica e estudo de caso para validação das técnicas e ferramentas apresentadas.

A revisão bibliográfica abordou aspectos relevantes sobre a arqueologia de software e o processo de extração de informações em sistemas legados em que há pouca ou nenhuma documentação disponível. Além disso, ela forneceu subsídios importantes para a construção de casos de teste a partir dos elementos encontrados.

No estudo de caso, as técnicas e ferramentas apresentadas foram avaliadas através da sua aplicação em situações reais. A realização do estudo de caso se deu em dois cenários específicos. No primeiro, foram aplicadas técnicas de extração de informações com a posterior construção de casos de teste no ERP da empresa Sesc-RS. No segundo cenário, foi analisado

o processo utilizado pela empresa Meta3, situada em Belo Horizonte, e que faz uso da arqueologia de software como um dos instrumentos para mapear código-fonte e processos de negócio em sistemas legados.

3.2. Definição da unidade-caso

O estudo de caso foi realizado na empresa Meta3 Services & Technologies, situada em Belo Horizonte/MG, e que atua como integradora de TI. Um dos critérios que favoreceram a escolha desta empresa é o uso que a mesma faz da arqueologia de software para extrair informações de sistemas legados. Além disso, a empresa desenvolveu uma ferramenta própria que auxilia seus analistas nesse processo.

3.3. Técnicas de coleta de dados

Para cumprir os objetivos deste trabalho, a técnica de coleta de dados empregada foi a realização de entrevistas. Conforme Yin (2001), as entrevistas são uma das mais importantes fontes de informações para um estudo de caso. Neste trabalho, foram utilizadas entrevistas semiestruturadas que, segundo Yin (2001), são aquelas que partem de alguns questionamentos básicos, apoiados em teorias e hipóteses que interessam ao estudo.

O estudo de caso relacionado à Meta3 se concentrou na entrevista de um dos diretores da Meta3 e de um analista com grande envolvimento no processo de migração de sistemas legados. O objetivo dessa entrevista era caracterizar o processo e a ferramenta que a empresa utiliza para recuperar informações em sistema legados e se, a partir do que foi extraído pela ferramenta ela consegue construir casos de teste.

3.3. Técnicas de análise dos dados

O passo seguinte à realização das entrevistas consistiu na organização das informações e dos dados coletados para, em seguida, proceder com a sua análise. De acordo com Yin (2001), "a análise de dados consiste em examinar, categorizar, classificar em tabelas ou, do contrário, recombinar as evidências tendo em vista proposições iniciais de um estudo".

Essa análise foi fundamental para o andamento das demais etapas do trabalho e tinha por objetivo reunir elementos que pudessem ser considerados relevantes no processo de identificação de informações em sistemas legados.

4. Análise e Apresentação de Resultados

4.1. Seleção de ferramentas

Para a realização do estudo de caso, foram selecionadas ferramentas disponíveis no mercado que oferecessem recursos de análise de código-fonte para extração de informações. Um dos critérios que levaram a seleção das ferramentas foi a disponibilidade de uma versão completa, sem restrições de funcionalidades, para a realização dos estudos de caso. Nesse sentido, a escolha da ferramenta Understand, desenvolvido pela Scientific Toolworks se deu em função da gama de recursos que ela oferece e pelo fato da empresa disponibilizar uma versão para uso acadêmico com as mesmas funcionalidades da versão comercial (SCITOOLS, 2017a).

Apesar da ferramenta MARBLE, referenciada na fundamentação teórica, oferecer recursos que viabilizam a extração de informações, ela não foi objeto deste estudo de caso. Isso decorreu do fato da ferramenta não estar mais disponível para *download*, mesmo sendo um *plug-in* gratuito para o Eclipse e ser referência em arqueologia de software. Alia-se a isso o fato dela suportar somente código desenvolvido em Java e não ter havido novas implementações da ferramenta. A última versão foi disponibilizada em 2010.

A ferramenta Meta3-AI (*Application Inspector*) se tornou objeto deste estudo de caso em função da mesma ser desenvolvida por uma empresa estabelecida no Brasil e cuja história foi construída em torno de processos envolvendo sistemas legados. Dessa forma, a experiência adquirida ao longo dos anos possibilitou o desenvolvimento de uma ferramenta que, de fato, pudesse auxiliar na extração de informações destes sistemas. Além disso, a disposição da empresa em fornecer informações a respeito da ferramenta e de seus processos favoreceu na seleção da mesma para o estudo de caso.

4.1. Estudo de caso da Meta3

4.1.1. Apresentação da empresa

A Meta3 Services & Technologies é uma empresa de integração de TI situada em Belo Horizonte, estado de Minas Gerais, e conta com clientes das áreas de telecomunicações, bancos, instituições financeiras, organizações governamentais e corporações que possuem computadores de processamento crítico de dados, como mainframes. Ela atua na prestação de serviços (desenvolvimento de sistemas, fábrica de software, consultoria em governança da tecnologia e mapeamento e gerenciamento de processos de negócio, manutenção e arqueologia de sistemas legados), comercialização de produtos próprios (Sistema de Gestão de Vendas, Sistema de Gerenciamento Eletrônico de Documentos, Sistema de análise de sistemas legados) e comercialização e implantação de produtos de terceiros, em que mantém uma parceria com a Parasoft© para testes de software com ferramentas de prevenção de defeitos e análise de códigos-fonte para identificação de problemas (META3, 2017).

4.1.2. Sistemas legados

Entre os serviços prestados pela empresa, a manutenção e a arqueologia de sistemas legados são objetos do estudo de caso. Segundo consta na página da empresa na Internet (META3, 2017), a manutenção de sistemas legados consiste em sustentar e conservar os sistemas atuais existentes no cliente, enquanto que a arqueologia de sistemas legados tem por objetivo criar a documentação de regras de negócio de sistemas existentes nas organizações.

Os clientes com os quais a Meta3 trabalha, em geral, possuem sistemas legados desenvolvidos em Cobol, Natural/Adabas, PL/SQL, Delphi, Visual Basic, entre outras linguagens estruturadas. Estes sistemas, em sua maioria, foram construídos por volta da década de 90, são grandes, geralmente não tem documentação e são difíceis de compreender. Normalmente os desenvolvedores originais já não se encontram mais no cliente e, em outros casos, desenvolvedores retém o conhecimento para si a fim de manterem-se estáveis na organização. E é em função desse cenário que os clientes procuram a Meta3 a fim de minimizar a dependência das pessoas e possibilitar uma migração mais segura e confiável.

Para dimensionar a complexidade de um sistema legado, a Meta3 analisa a quantidade de módulos, programas e o número de linhas. Os tipos de alterações mais comuns são a inclusão de novas funcionalidades e geralmente não visam a otimização de recursos.

Determinadas partes do código-fonte atual dos sistemas legados costumam ser muito diferentes daquele originalmente concebido e, em geral, não existe o registro e histórico dessas alterações. Não há uma certeza se, em função das alterações, o código-fonte continua alinhado com os processos de negócio atuais. Por outro lado, é comum encontrar programas que nunca foram alterados ao longo de toda a vida do sistema.

As principais adversidades encontradas nos sistemas legados são a retenção de informação por parte de determinados funcionários e a confiabilidade com relação ao que se está vendo no código. Existe uma incerteza se o que está no código é o que corresponde ao que de fato deve estar implementado. Além disso, são encontradas muitas dependências entre os diversos componentes e estruturas, muitas vezes derivadas das próprias metodologias de desenvolvimento da época.

A principal fonte de informações sobre o sistema legado é o próprio código-fonte. Raramente são encontrados outros artefatos como documentos e diagrama. A partir do código-fonte são extraídas rotinas, estruturas de dados, estruturas de controle de fluxo, relações de dependência entre componentes e relacionamento entre tabelas.

Como técnica de análise de código-fonte, a Meta3 utiliza a análise estática através de *parsers* baseados na linguagem e, num segundo momento, a análise dinâmica para realizar um refinamento sobre as informações coletadas.

4.1.3. A ferramenta Meta3-Application Inspector (AI)

O Meta3-AI é uma ferramenta para gestão de sistemas legados e migrações de plataforma (META3, 2017). Ela foi construída em Visual Basic e é executada no ambiente Windows. De acordo com Meta3 (2017), essa ferramenta "gera o mapa de relacionamento entre os sistemas legados por intermédio da criação de um repositório único do conhecimento, fornecendo informações em relatórios para os analistas realizarem manutenção com facilidade".

A ferramenta se adapta bem a uma ampla variedade de tipos de software baseados em linguagens estruturadas. No entanto, é necessário realizar uma calibragem através de ajustes em parâmetros a fim de adequá-la a um determinado sistema. Em função dessa adequação ao ambiente do cliente, esse processo não é automatizado. O Meta3-AI não se mostra adequado em cenários que envolvam linguagens orientadas a objeto, como Java, e linguagens Web.

O funcionamento do Meta3-AI se inicia com um processo chamado *scan*, que consiste na leitura dos arquivos fontes do sistema a ser analisado. Nesse processo são realizados ajustes através de parâmetros nas telas a fim de selecionar o que se quer buscar. Todos os objetos são catalogados num repositório de acordo com seu tipo e linguagem. Ao se iniciar o *scan*, são geradas informações relativas à referência cruzada que mostram a relação entre os objetos, permitindo visualizar as referências geradas ou sofridas por um determinado objeto.

É possível criar padrões de reconhecimento (agrupamentos de critérios de pesquisa previamente cadastrados) para a inclusão de termos ou expressões em que se deseja pesquisar

nos códigos-fonte dos objetos do repositório. Estes termos podem ser expressões, nome de variáveis, tabelas, arquivos, constantes, mensagens ou qualquer sequência de caracteres válidos que se queira encontrar em um objeto. Essas pesquisas muitas vezes podem trazer um "falso positivo", ou seja, uma expressão que não faz parte do objetivo do usuário, embora se encaixe nos critérios de pesquisa. Para contornar isso, é possível criar Padrões de Exclusão que visam eliminar tais expressões e refinar os resultados obtidos em conjunto com um determinado Padrão de Reconhecimento.

O Meta3-AI possui uma tabela de símbolos, em que é possível controlar a quantidade de palavras reservadas para Natural ou COBOL que existem em um fonte antes e depois de uma alteração. Este recurso permite observar o tamanho da mudança implementada através do número destas palavras reservadas. A ferramenta possui ainda um módulo de base de conhecimento em que são adicionados e armazenados os termos mais comuns de uma linguagem, facilitando a localização de comandos que devem sofrer manutenção.

Ao se executar o Meta3-AI são disponibilizadas diversas informações relativas aos objetos catalogados:

- Cálculo de pontos por função, nos moldes estabelecidos no IFPUG (*International Function Point Users Group*), tanto para os módulos existentes como para futuras implementações.
- Cálculo de complexidade de programas baseado na métrica de Halstead (HALSTEAD, 1977), permitindo extrair informações como:
    - Cálculo do Volume de Programas: permite verificar se há muitas funcionalidades dentro de um mesmo programa, auxiliando o analista na definição de separar em módulos menores, menos complexos e mais específicos.
    - Quantidade de linhas de código (LOC - *line of code*): Cálculo de linhas úteis excluindo-se os comentários e linhas em branco, permitindo avaliar o tamanho de um programa.
    - Número de *bugs* estimados: Cálculo da métrica esperada de erros tomando por base o tempo de implementação de um programa ou módulo.
- Listas de referência cruzada: um amplo conjunto de telas mostra as diversas relações de dependência entre os objetos. Um exemplo de tela disponibilizada pelo Meta3-AI pode ser visto na Figura 2.
- Árvore de módulos: Esta árvore distingue-se da árvore de referência cruzada, por identificar os relacionamentos em um nível hierarquicamente acima ao dos objetos. As relações são apontadas no nível dos módulos, sendo que os objetos responsáveis pela relação entre os módulos são apontados em detalhe em uma lista a parte.

- Árvore de *Data Set Name*s: permite relacionar arquivos lógicos (*Data Definition Name*) acessados por um programa COBOL, aos seus respectivos arquivos físicos VSAM (*Data Set Name*).
- Relacionamento DSN x Books: Permite fazer uma associação, onde arquivos VSAM são mapeados a layouts de dados, definidos via *copy books* COBOL.

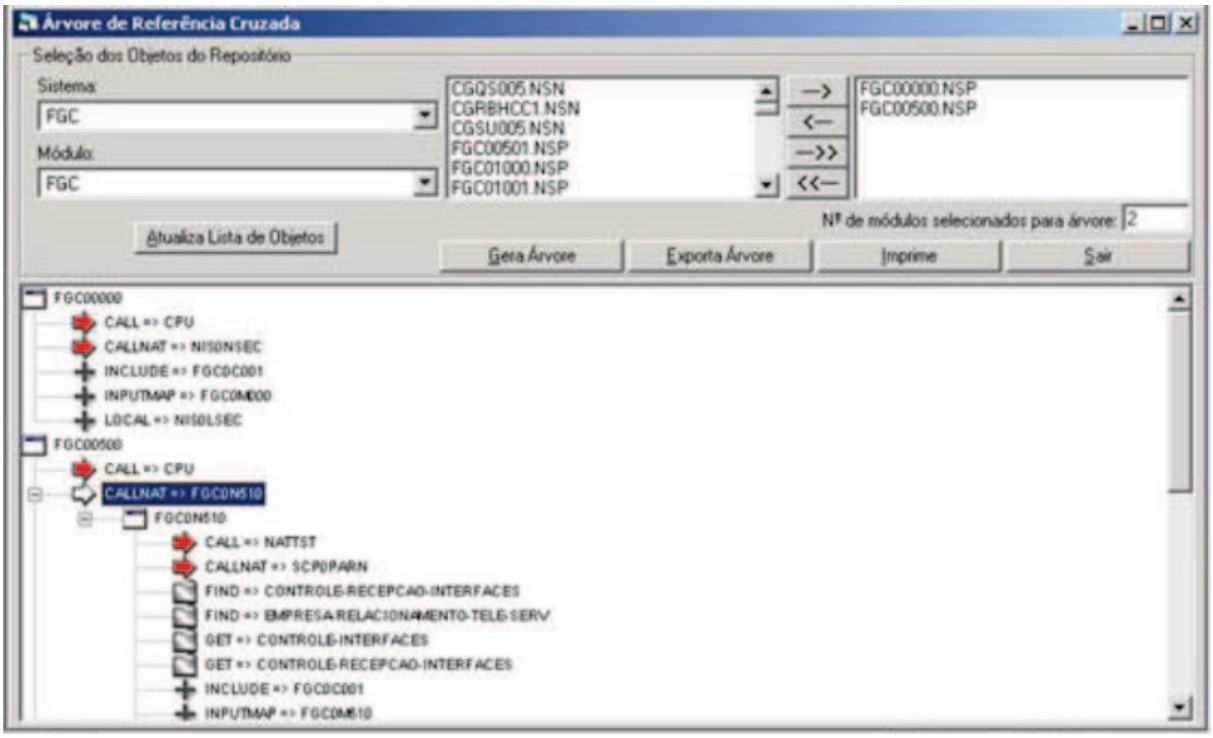

Figura 2. Árvore de referência cruzada (META, 2017)

Além dessas informações, o Meta3-AI apresenta um módulo de relatórios que disponibiliza uma série de informações baseadas na análise do código-fonte:

- Lista hierárquica de Calls: mostra as chamadas que partem dos objetos selecionados em um módulo ou sistema. Permite exibir os comandos referentes a chamadas, referentes a arquivos e/ou referentes a estruturas. É um relatório hierárquico, pois exibe não somente os Calls, mas também todas as suas dependências.
- *Reference List* de Chamadas: relatório que aponta o objeto de origem, o comando de chamada utilizado e os objetos alvo, além do número total de referências para cada objeto.
- *Reference List* de Arquivos: gera um relatório detalhando o objeto de origem e os objetos alvo a que faz referência de leitura e escrita, assim como número total de referências para cada objeto.
- *Reference List* de Estruturas: relatório exclusivo para o ambiente *mainframe*, pois contempla relações entre Copybooks, Mapas, Local, Include, Global e

outras estruturas pertinentes à alta-plataforma, a partir do módulo e sistemas selecionados.

- Relação entre Objetos: exibe todas as relações entre objetos dentro de um universo pré-definido (um módulo, um sistema, ou todos os módulos e sistemas).
- Relação entre Módulos: exibe todas as referências entre os módulos, em um sistema ou em todos os sistemas.
- Matriz Arquivos/Programas/Atualizações: gera uma matriz que apresenta a relação de atualizações (*find, read, write, store, update*) entre cada arquivo e cada programa.
- Referências a Objetos Faltantes: disponibiliza uma lista de todos os objetos faltantes que são referenciados por um objeto existente, exibindo o nome e o tipo do objeto faltante, além do nome, o tipo e o tipo de referência do objeto que o referencia.
- Referência a Arquivos: exibe uma lista contendo o nome e o tipo do arquivo referenciado, e o nome, o tipo e o tipo de referência feita pelos objetos que o referenciam.
- Objetos Faltantes: exibe uma lista com o nome e o tipo de todos os objetos faltantes.
- Objetos mais Usados: exibe uma lista com o nome, o tipo e número total de referências dos objetos mais referenciados. A lista é organizada do objeto mais referenciado para o menos referenciado.
- Parâmetros de Programas Chamados: gera um relatório de informações provenientes da área de parâmetros de cada objeto que utiliza parâmetros. Exclusivo para ambiente *mainframe*.
- Programas mais Referenciados – Resumido: gera relatório contendo os objetos referenciados pelo menos duas vezes. Apresenta o nome do programa, LOC (Linhas de Código), RMK (Comentários), Referências Únicas (contabiliza apenas uma referência por objeto distinto, não repetindo a contagem no caso de mais de uma referência em um mesmo objeto) e Referências Totais (número total de referências, ainda que sejam do mesmo objeto).
- Programas mais Referenciados – Detalhado: relatório que apresenta, além dos dados do relatório de programas mais referenciados resumido, quais são os Programas Origem (objetos que fazem referência ao Programa Alvo) e a quantidade de referências que cada um faz no Programa Alvo.
- Programas mais Chamados por Jobs – Resumido: gera um relatório contendo os objetos mais chamados por Job, organizado do mais referenciado para o menos referenciado, incluindo o número de Linhas de Código (LOC), Comentários

(RMK), referências únicas e referências totais. Só são exibidos os objetos que possuem pelo menos duas referências.

Todo o processo envolvendo a análise do código-fonte e a disponibilização de informações está ilustrado na Figura 3.

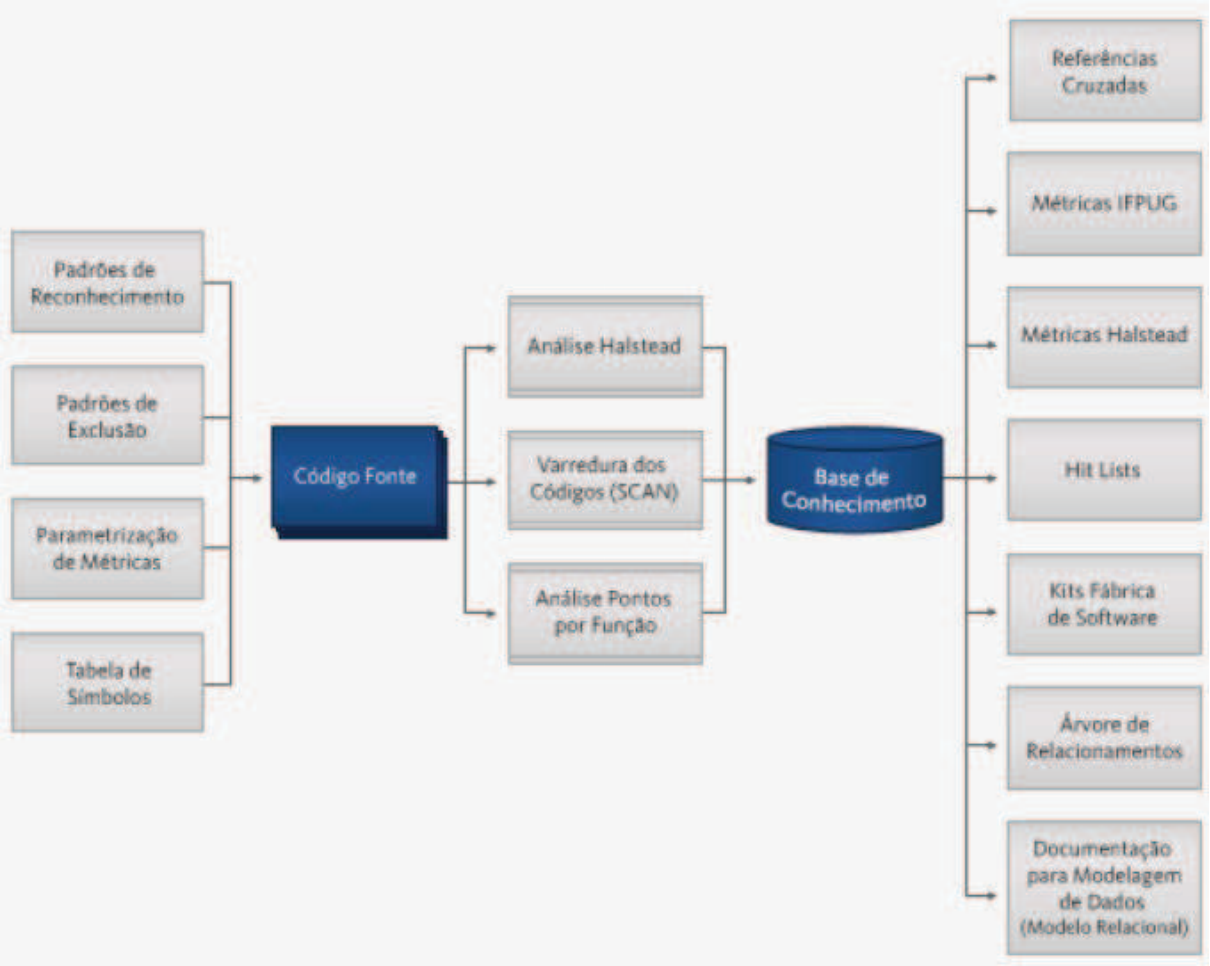

Figura 3. Funcionamento do Meta3-AI (META, 2017)

4.1.4. Construções de casos de teste

A ferramenta Meta3-AI oferece um conjunto muito amplo de recursos que possibilitam uma análise bem detalhada do código-fonte. As informações que o Meta3-AI gera permitem que os analistas tenham uma visão da complexidade envolvendo o sistema legado e que orientem suas atividades em torno dos resultados mostrados pela ferramenta. Ela oferece subsídios importantes para que todo o processo envolvendo o sistema legado seja conduzido de forma segura e prática para os analistas.

Por outro lado, os entrevistados consideram que, mesmo com esse conjunto de informações que a ferramenta oferece, ainda é necessário o envolvimento de um especialista no negócio para aprimorar e extrair informações mais concisas após a execução da ferramenta. Isso se refere principalmente a recuperação de regras de negócio e, consequentemente, na construção de casos de teste. Somente com as informações obtidas na análise do código-fonte não é possível construir casos de teste. Para isso é necessário que se faça antes a construção de

casos de uso. No entanto, conforme documentação fornecida pela Meta3, os recursos de rastreabilidade que a ferramenta oferece possibilitam o planejamento de testes integrados por meio das informações fornecidas na Árvore de Referência Cruzada para visualização dos módulos envolvidos (“chamadores” e “chamados”).

4.2. Estudo de caso do Sesc-RS

4.2.1. Apresentação da empresa

O Sesc – Serviço Social do Comércio é uma entidade de caráter privado, mantida e administrada pelos empresários do Comércio. Possui diversas unidades operacionais espalhadas pelo estado que prestam atendimentos e serviços nas áreas de saúde, educação, cultura, esporte e lazer aos trabalhadores do comércio de bens, serviços e turismo. Devido a isso tem uma dependência muito forte da tecnologia da informação e dos sistemas que dão suporte ao negócio. Atualmente, o negócio do Sesc-RS é mantido por um sistema construído em Genexus.

4.2.2. Apresentação da ferramenta

O segundo estudo de caso abordou o uso da ferramenta Understand, desenvolvido pela empresa Scientific Toolworks (www.scitools.com). Ela é uma ferramenta de análise estática de código que auxilia na compreensão, manutenção e documentação de sistemas, sejam eles legados ou não, através de um conjunto de recursos visuais, documentos, métricas e fluxogramas (SCITOOLS, 2017). O Understand cria um repositório das relações e estruturas contidas num projeto. Esse repositório é então utilizado para aprender sobre o código-fonte e através de funcionalidades de análise permite responder a questões como:

- O que é uma determinada entidade?
- Onde ela é modificada?
- Onde ela é referenciada?
- Quem depende dela?
- Ela depende de quais entidades?

A ferramenta faz a análise de código-fonte escrito nas linguagens C, C++, C#, Objective C/Objective C++, Ada, Assembly, Visual Basic, COBOL, Fortran, Java, JOVIAL, Pascal/Delphi, PL/M, Python, VHDL, e Web (PHP, HTML, CSS, JavaScript, e XML) (SCITOOLS, 2017). Para a realização do estudo de caso foi utilizada a versão 4 (Build 904). A empresa desenvolvedora disponibiliza para uso acadêmico uma versão (usada nesse artigo) com as mesmas funcionalidades da versão comercial, porém com o período de uso restrito a 1 (um) ano.

Para verificar a viabilidade de se usar a ferramenta Understand na extração de informações de sistemas legados e, posteriormente, utilizá-las para construção de casos de teste, foi utilizado um sistema de Matrículas da organização Sesc-RS. Esse sistema foi construído em Visual Basic por volta do ano 2000 e se manteve ativo até meados de 2008, quando foi migrado para uma versão mais moderna. Apesar do sistema não estar mais em uso, a aplicação do Understand sobre ele fornece subsídios importantes sobre como a ferramenta extrai as

informações e contribui para a sua utilização em outros projetos que envolvam sistemas legados.

4.2.3. Processo de extração de informações

O Understand possui uma IDE (*Interactive Development Environment*) que permite a seleção de suas funções de maneira rápida e intuitiva. Através dessa IDE é possível realizar uma série de análises a partir de gráficos e tabelas que são disponibilizados após a execução da análise do código-fonte.

Para a realização do estudo de caso, após a execução do Understand foram selecionados alguns artefatos produzidos pela ferramenta que demonstram o processo de extração de informações. O primeiro artefato selecionado é um diagrama em que é exibida a estrutura da arquitetura do sistema de Matrículas. Essa estrutura é vista na Figura 4, na qual é possível observar as principais camadas do sistema.

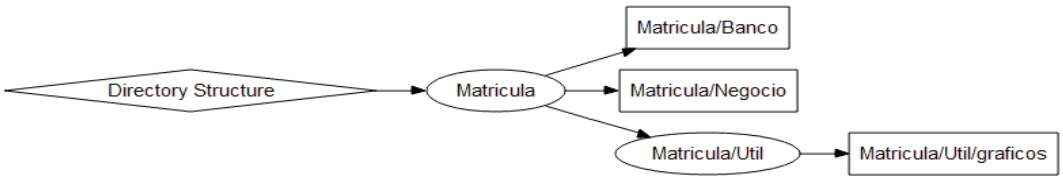

Figura 4. Estrutura da arquitetura do sistema de Matrículas (Fonte: do autor)

Devido às restrições de tempo, a aplicação do estudo de caso deu-se somente sobre a camada de negócio, visto que é nessa camada que se concentram as regras de negócio do sistema de Matrículas.

O próximo diagrama (Figura 5) mostra a estrutura interna do sistema de Matrículas e a quantidade de objetos das camadas Banco e Util que são chamados pela camada de Negócio.

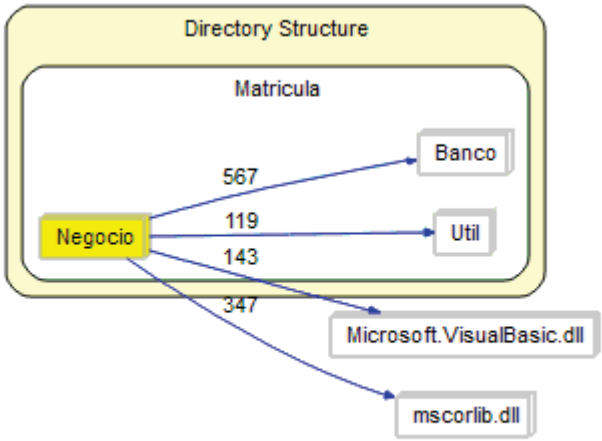

Figura 5. Quantidade de objetos chamados por Negócio (Fonte: do autor)

Ao se aprofundar na camada de negócio, é possível ver os diversos objetos que compõem essa camada e os relacionamentos existentes entre eles, conforme pode ser visto na

Figura 6. Os números sobre as setas indicam a quantidade de chamadas que um objeto x faz sobre um objeto y.

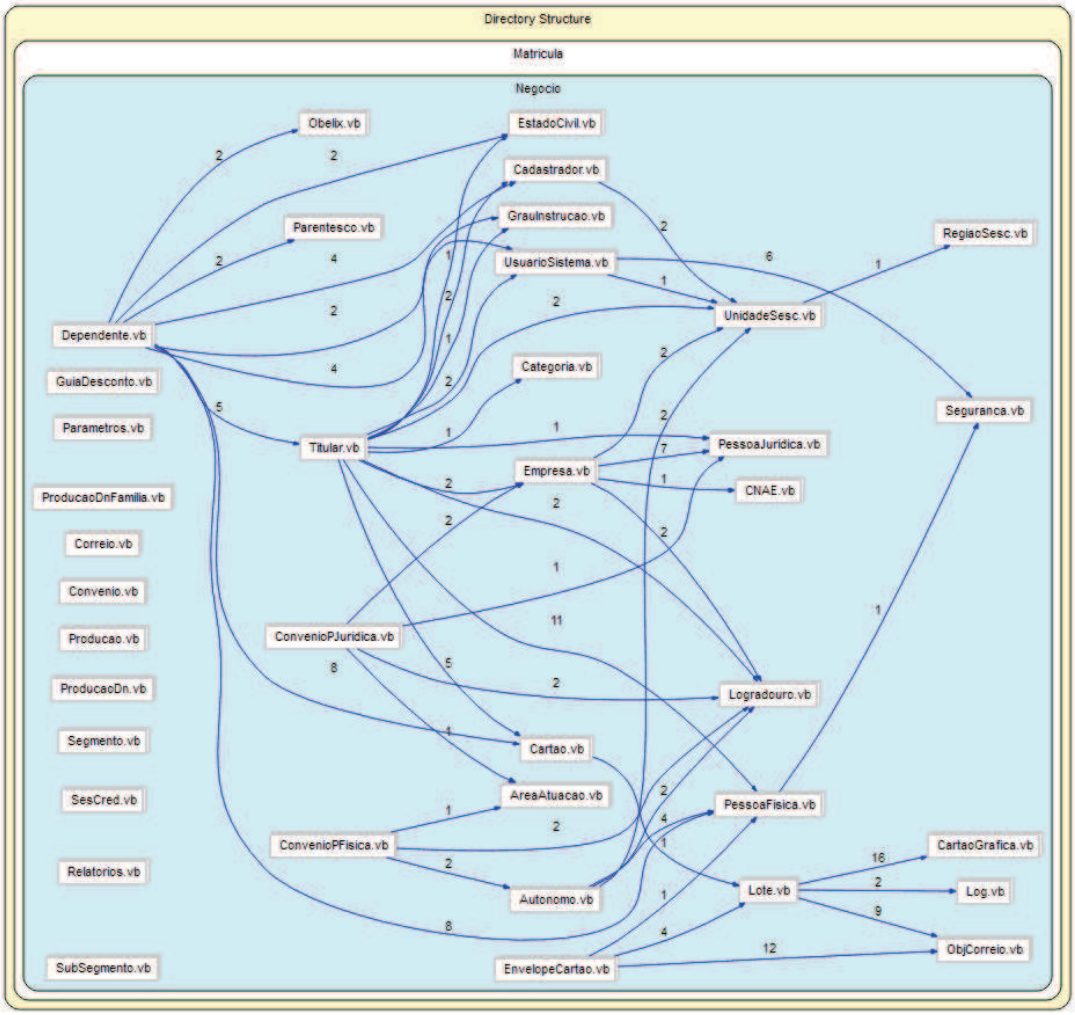

Figura 6. Estrutura interna da camada de Negócio (Fonte: do autor)

Na Figura 6 é possível observar que alguns objetos não possuem nenhum relacionamento com outros objetos. Isso se deve a duas situações. Na primeira são objetos, como “Convenio.vb”, que são classes mãe para outras classes, no caso “ConvenioPJuridica.vb” e “ConvenioPFisica.vb”. Na segunda situação são objetos que não são referenciados por nenhum outro objeto.

O Understand oferece uma diversidade de gráficos que possibilitam avaliar a complexidade do sistema e, a partir disso, decidir pela priorização de objetos que devem ser tratados de acordo com a sua importância dentro do sistema. O gráfico TreeMap, por exemplo, destaca objetos através de blocos que variam de tamanho e cor. Cada bloco representa um arquivo de código e diferentes métricas podem ser observadas nesse tipo de gráfico. Na Figura7, o gráfico TreeMap é baseado no número de linhas de cada arquivo e na métrica de complexidade ciclomática, permitindo assim destacar quais arquivos são grandes e complexos daqueles que são grandes e não são complexos.

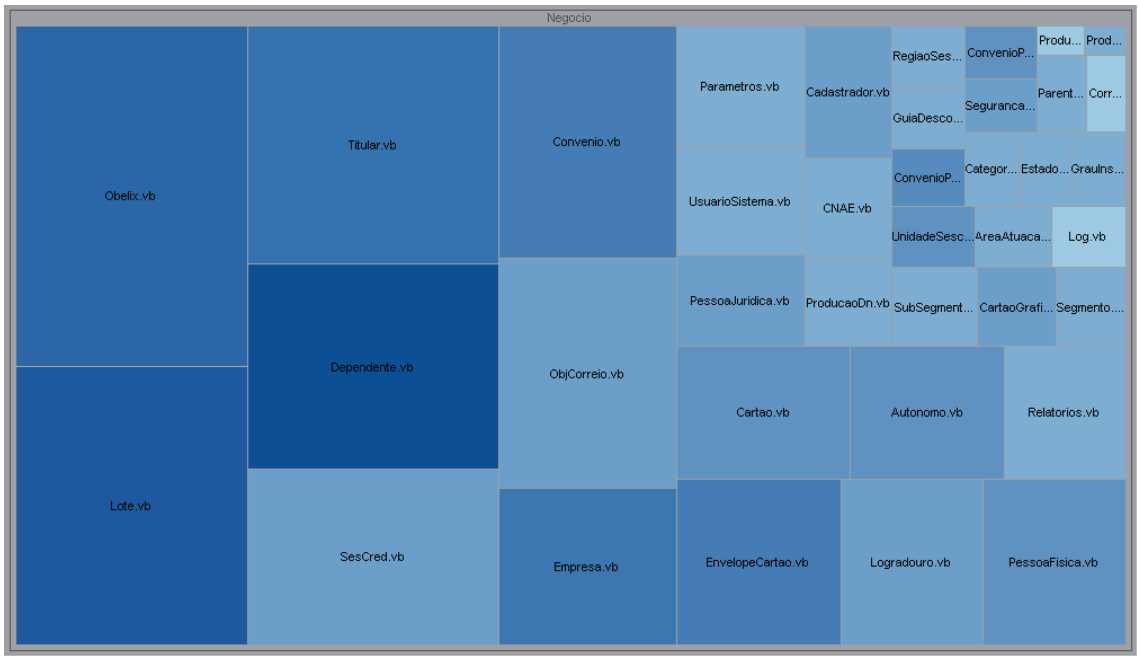

Figura 7. Gráfico TreeMap da camada de Negócio (Fonte: do autor)

Ao analisar o gráfico da Figura 7, é possível observar que o programa Dependente.vb se destaca por possuir maior quantidade de linhas de código e maior complexidade ciclomática. Por outro lado, o programa SesCred.vb apesar de ter tamanho semelhante não é particularmente complexo.

A Figura 8 apresenta um gráfico com métricas relacionadas ao volume de distribuição do código, em que é possível verificar as quantidades de linhas de código, linhas de comentário e linhas em branco dos diversos programas que compõem a camada de Negócio. Nesse exemplo, para efeitos de visualização, foram selecionados apenas 14 programas dessa camada. É possível verificar também que o arquivo de código Titular.vb apresenta a maior quantidade de linhas de código.

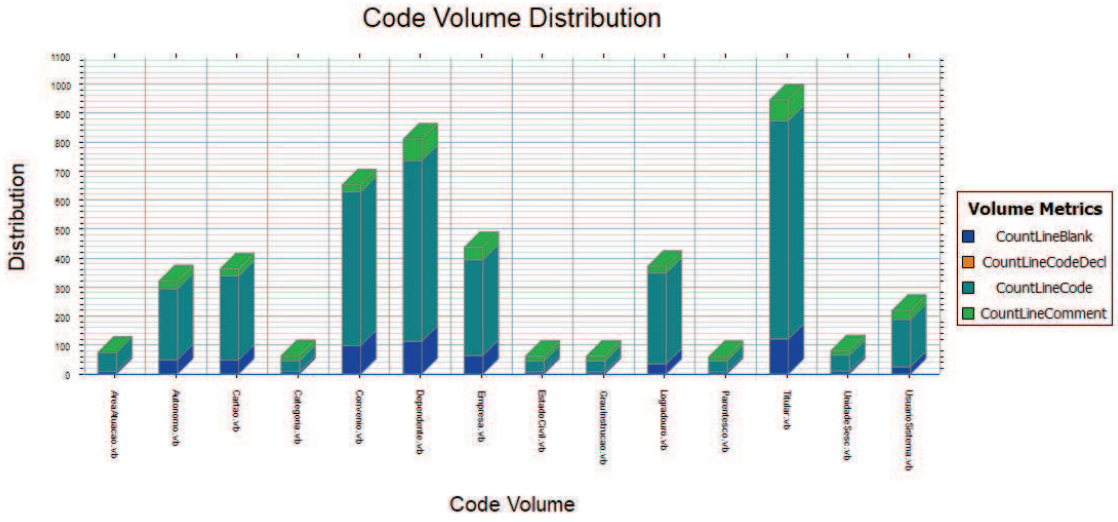

Figura 8. Volume de distribuição do código (Fonte: do autor)

Após analisar as métricas de volume de distribuição do código, o passo seguinte consistiu na análise da complexidade ciclomática, como ilustrado na Figura 9.

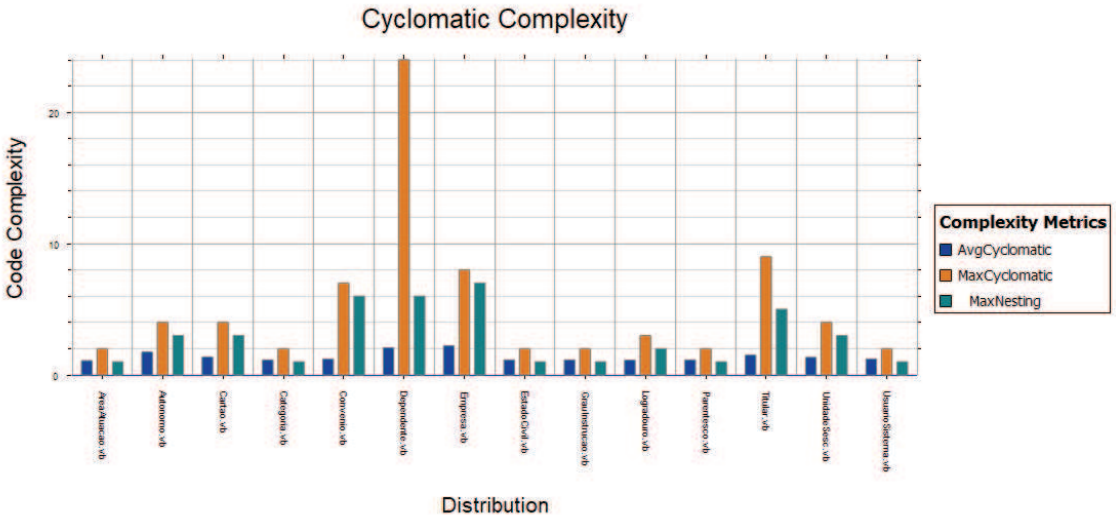

Figura 9. Complexidade ciclomática (Fonte: do autor)

Nesse gráfico da Figura 9 pode-se verificar que o programa Dependente.vb tem uma complexidade ciclomática alta, seguida do programa Titular.vb. A partir desse ponto, o estudo de caso se concentrou no arquivo de código Titular.vb. A escolha se deve ao fato deste arquivo ter um tamanho de código relativamente grande e apresentar um certo grau de complexidade. Além disso, a figura do titular é um dos pontos centrais no negócio do Sesc-RS, visto que ele é o principal usuário dos serviços oferecidos.

Após definido o objeto de análise, o passo seguinte consistiu na verificação dos objetos que dependem de Titular.vb. Essa etapa está ilustrada na Figura 10.

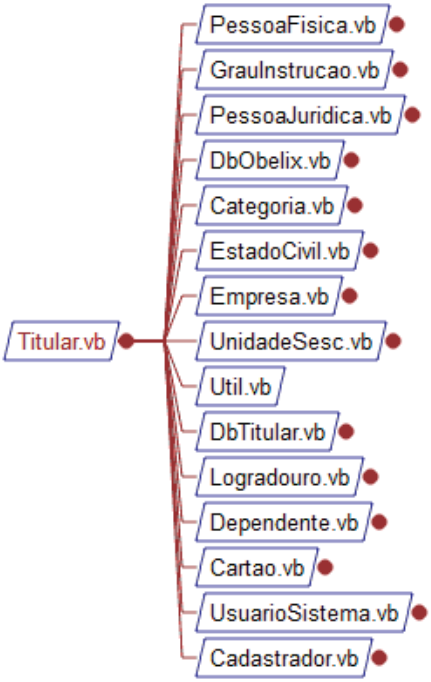

Figura 10. Programas dependentes de Titular.vb (Fonte: do autor)

A identificação dos programas que dependem de um determinado objeto facilita a rastreabilidade, pois indicam que esses programas podem ser passíveis de execução de um teste de regressão caso haja alguma modificação no objeto chamador.

Na sequência, foi gerado no Understand o diagrama UML da classe Titular. Esse diagrama mostra que Titular é uma subclasse de PessoaFisica, conforme pode ser visto na Figura 11.

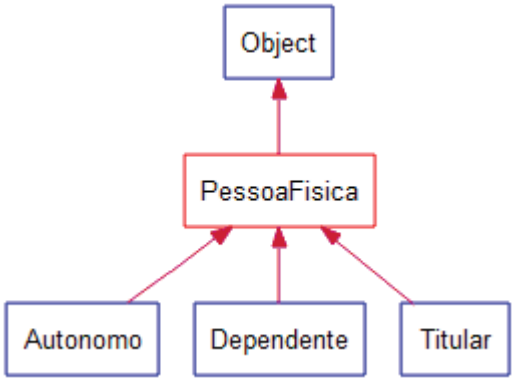

Figura 11. Diagrama UML da classe PessoaFisica e suas subclasses (Fonte: do autor)

Posteriormente o diagrama UML foi expandido para a classe PessoasFisica e a subclasse Titular a fim de verificar a sua estrutura interna, com seus atributos e métodos. O diagrama de classe pode ser conferido na Figura 12. Em função do seu tamanho, optou-se por demonstrar um fragmento desse diagrama.

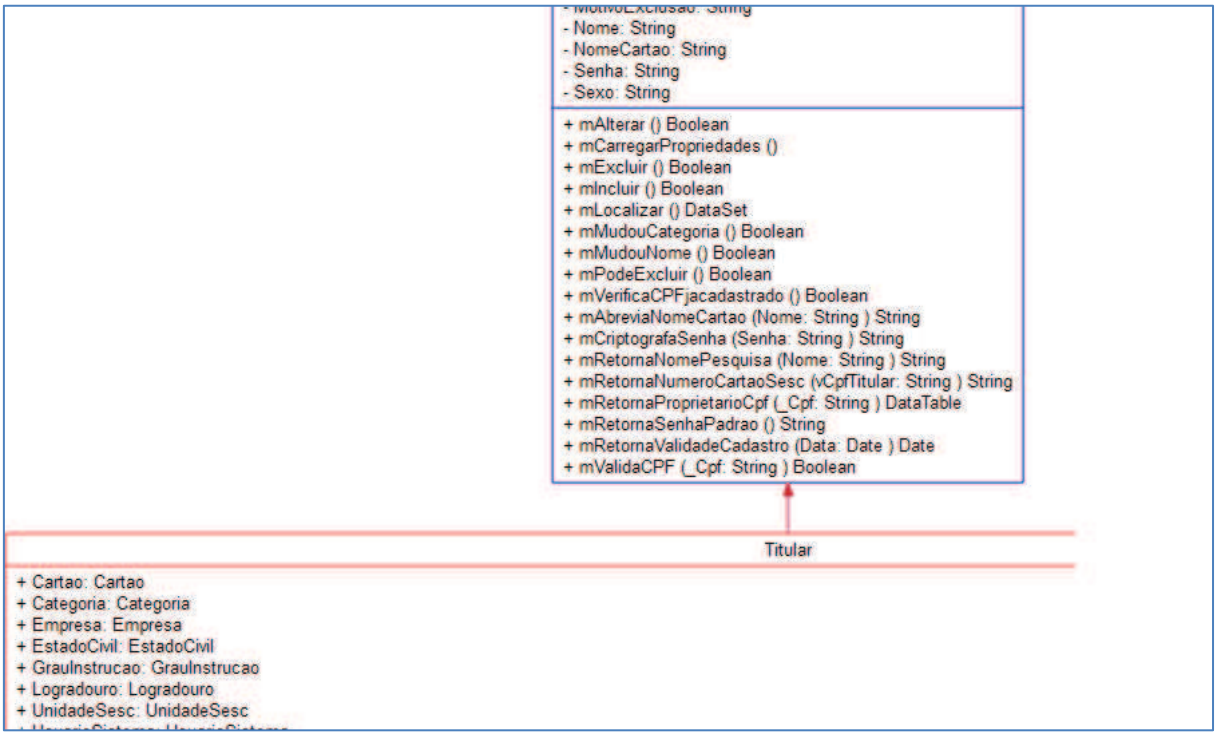

Figura 12. Fragmento do diagrama expandido da classe PessoaFisica e Titular (Fonte: do autor)

O diagrama apresentado na Figura 12 possibilita conhecer todos os métodos da classe que são, de certa forma, o ponto de onde devem ser extraídas as regras de negócio.

Em seguida deu-se a análise das dependências da classe titular com as suas chamadas aos métodos. A interface intuitiva do Understand possibilita que de qualquer ponto seja possível acessar qualquer objeto referenciado. Isso facilita a localização de fontes que possam conter regras de negócio, por exemplo. Na Figura 13 a função JaPossuiConjuge e que contempla uma regra de negócio aparece destacada .

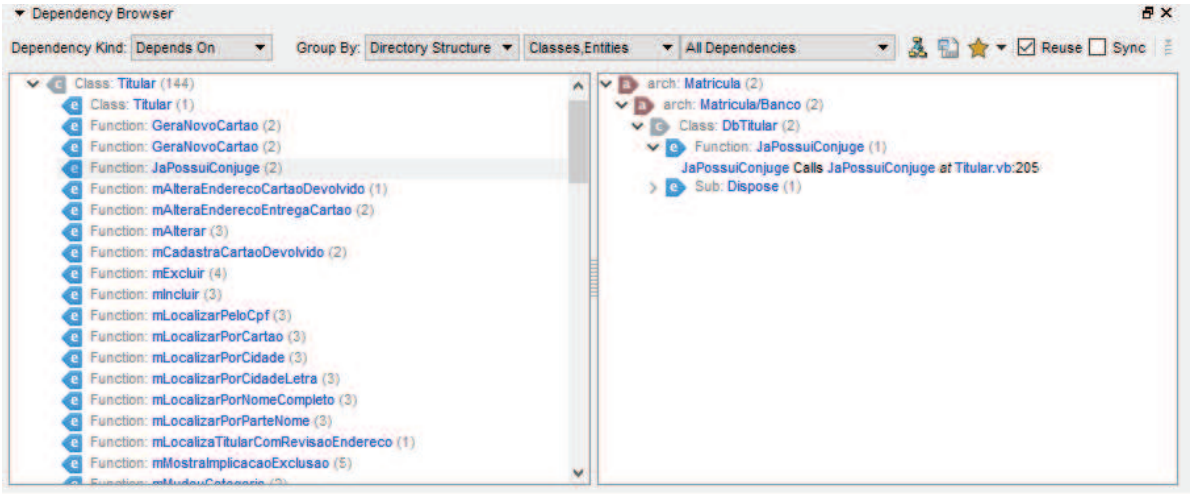

Figura 13. Dependências da classe Titular (Fonte: do autor)

Para facilitar a compreensão dos objetos e suas relações, o Understand oferece uma variedade de formas de visualização, como o fragmento do diagrama mostrado na Figura 14. Nesse diagrama são mostrados todos os métodos que compõem a classe Titular.

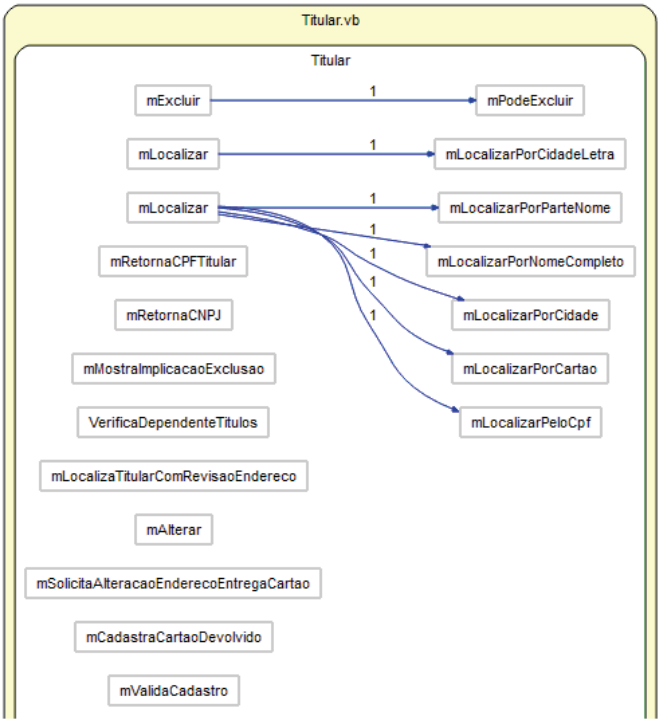

Figura 14. Estrutura interna com os métodos da classe Titular (Fonte: do autor)

O Understand possibilita a geração do fluxo de controle de cada método presente no sistema. Com isso, é possível visualizar como a informação é tratada num método específico. Para o estudo de caso foi selecionado o método mValidaCadastro e está ilustrado na Figura 15.

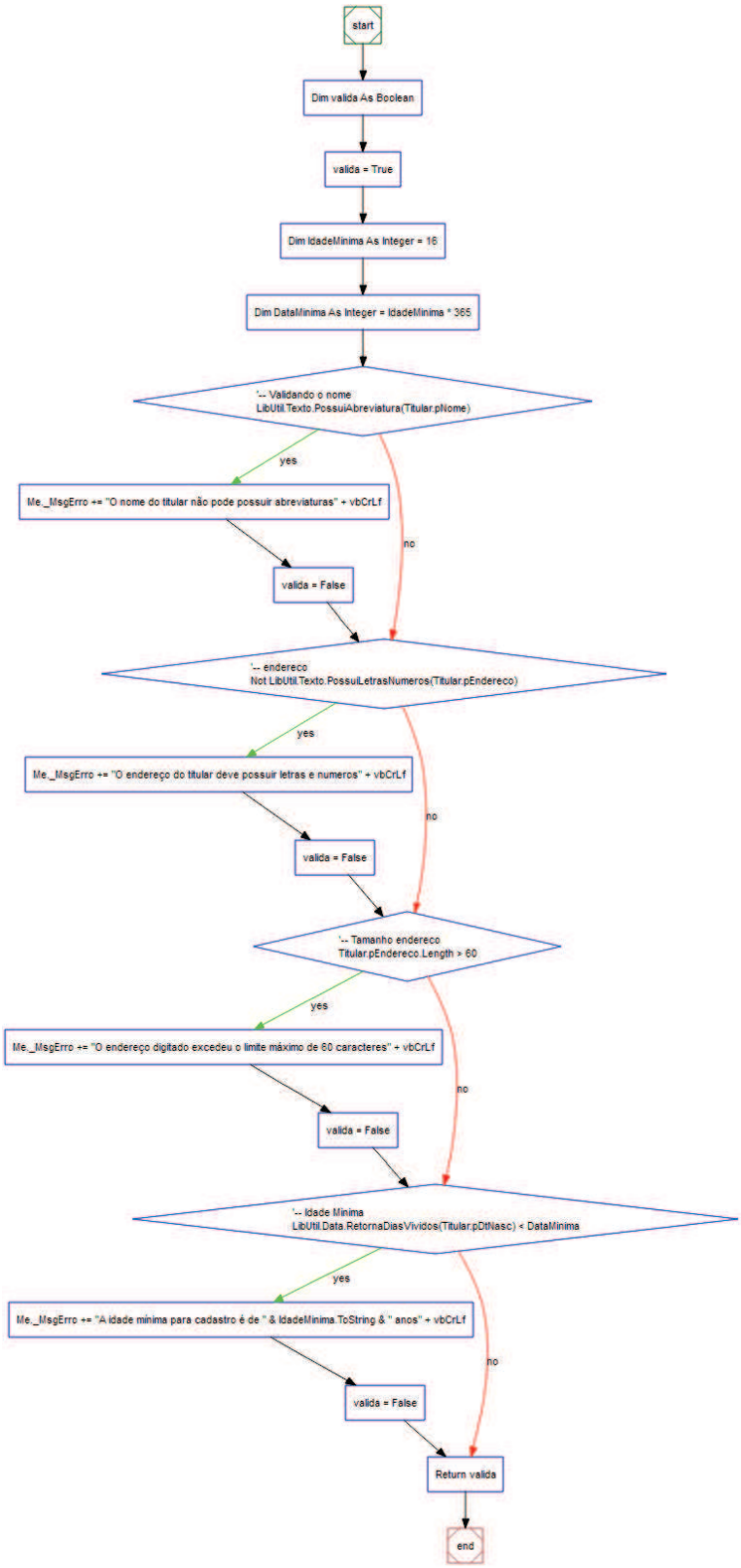

Figura 15. Fluxo de controle do método mValidaCadastro (Fonte: do autor)

A partir do fluxo de controle (Figura 15) é possível a identificação de algumas regras. No caso do método apresentado no fluxo anterior foi extraída a seguinte regra de negócio:

- ⇨ Idade mínima para cadastro deve ser de 16 anos.

Além disso, o fluxo de controle apresenta outras regras presentes no método que não se configuram como regra de negócio, mas servem para garantir que um dado seja informado corretamente:

- ⇨ Nome do titular não pode conter abreviaturas
- ⇨ Nome do titular deve conter somente letras e números
- ⇨ Endereço deve conter no máximo 60 caracteres

Após a identificação da regra de negócio, o passo seguinte consistiu no estabelecimento de classes de equivalência para o teste desta regra, conforme consta na tabela a seguir:

Tabela 2. Classes de equivalência para a regra da idade mínima

Condição de entrada	Classes de equivalência	
	Válidas	Inválidas
Ano de nascimento	AnoAtual-16 <= AnoNascimento <= AnoAtual	AnoAtual–16 > AnoNascimento > AnoAtual

Uma vez estabelecida a classe de equivalência, é possível construir os casos de teste a serem aplicados para testar a especificação definida na regra de negócio:

Tabela 3. Casos de teste para validar a especificação

<(AnoAtual-AnoDigitado), AnoAtual-16 <= AnoDigitado <= AnoAtual>
<(AnoAtual-16>, AnoAtual-16 <= AnoDigitado <= AnoAtual >
<(AnoAtual), “A idade mínima para cadastro é de 16 anos”>
<(AnoAtual-14), “A idade mínima para cadastro é de 16 anos”>
<(AnoAtual + 1), “A idade mínima para cadastro é de 16 anos”>

Dessa forma, o estudo de caso referente ao uso da ferramenta Understand atesta que a mesma se mostra adequada para extrair informações sobre regras de negócio em sistemas legados além de viabilizar a construção de casos de teste a partir do que foi extraído. É importante ressaltar também que, pelo fato da ferramenta oferecer recursos para analisar a rastreabilidade dos objetos, a execução de testes de regressão é facilitada em função disso.

5. Considerações Finais

A extração de informações em sistemas legados em geral implica na necessidade de se aprofundar no código-fonte, já que não se encontram outros artefatos como documentos e diagramas que possam ajudar nesse processo. Dessa forma, o uso de ferramentas para auxiliar a decifrar o que está por trás de um código se torna fundamental.

Este artigo teve a intenção de analisar algumas ferramentas disponíveis e, com base num referencial teórico, buscar a sustentação através de um estudo de caso para verificar se a

arqueologia de software pode, de fato, contribuir para a extração de informações em sistemas legados para, numa etapa posterior, construir casos de teste a partir do que foi extraído.

A partir do estudo de caso conclui-se que, apesar das ferramentas analisadas oferecerem recursos e funcionalidades para extração de informações, ainda é necessária a presença de um especialista para aprimorar e filtrar a gama de informações que são geradas nesse processo. Às vezes esse processo é facilitado pela eventual presença de algum artefato adicional, como um documento, manual ou diagrama.

As ferramentas, por si só, não são capazes de gerar casos de teste. Elas são parte de um processo em torno da arqueologia de software que visam buscar o conhecimento em sistemas de software. Esse conhecimento geralmente é traduzido em regras de negócio e, uma vez que essas regras são obtidas o processo de construção de casos de teste é facilitado.

A arqueologia de software trata-se de um processo que envolve pessoas, artefatos e ferramentas para compreender um sistema de software que, em geral, é muito antigo e do qual pouco se conhece. A partir do trabalho conjunto desses três elementos pode-se melhorar o nível de compreensão sobre o que se está vendo e, então, construir outros artefatos como, por exemplo, casos de teste.

É possível perceber que ferramentas, como o Understand, estão num processo crescente de evolução. Isso pode levar ao desenvolvimento de ferramentas com capacidade de extração de informações ainda mais complexas de sistemas legados, com opções de visualizações que facilitem muito nesse processo.

Processos de extração de informações como o que foi proposto por Pérez-Castillo, Guzmán e Piattini (2011) se mostram bastante eficazes e o desenvolvimento de ferramentas baseadas nessa proposta podem se constituir como sugestão de trabalho futuro nessa área.

6. Referências

BASTOS, Anderson et al. **Base de conhecimento em testes de software**. 2. ed. São Paulo: Martins Fontes, 2007.

BIRCHALL, Chris. **Re-Engineering Legacy Software.** New York: Manning, abril 2016.

CAI, Z.; YANG, X.; WANG, W. **Business Process Recovery for System Maintenance: An Empirical Approach**. In: International Conference on Software Maintenance, 25. 2009, Edmonton, Canada: IEEE Press, 2009, p. 399-402.

DI FRANCESCOMARINO, C.; MARCHETTO, A.; TONELLA, P. **Reverse engineering of business processes exposed as web applications**. In: Software Maintenance and Reengineering, European Conference on Software Maintenance and Reengineering, 13., 2009, Kaiserslautern, Germany. p 139-148.

ERNST, M. D. Static and dynamic analysis: synergy and duality. In: **International Conference on Software Engineering,** 25., 2003. Portland, OR, USA: IEEE, 2003. p.24-27.

FEATHERS, Michael C. **Trabalho Eficaz com Código Legad**o. Porto Alegre: Bookman, 2013.

FLICK, U. **Qualidade na Pesquisa Qualitativa**. Porto Alegre: Artmed, 2009.

GHOSE, A.; KOLIADIS, G.; CHUENG, A. **Process discovery from model and text artefacts**. IEEE Congress on Services. Salt Lake, USA, 2007.

HALSTEAD, M. H. **Elements of Software Science, Operating and Programming Systems Series.** Vol. 7. Elsevier, 1977.

HEUVEL, W. J. van den. **Aligning modern business processes and legacy systems: A component based perspective.** Cambridge, MA: The MIT Press. 2006.

HUNT, Andy; THOMAS, Dave. **Software Archaeology.** IEEE Software, v. 19(2). P. 20–22, March/April 2002.

IZQUIERDO-CORTAZAR, D.; ROBLES, G.; ORTEGA, F.; GONZALEZ-BARAHONA, J.M. **Using Software Archaeology To Measure Knowledge Loss in Software Projects Due To Developer Turnover.** GSyC/LibreSoft - Universidad Rey Juan Carlos, Madrid, Espanha, 2009.

META3. **Sobre a Meta3.** Belo Horizonte, 2017. Disponível em: <http://www.meta3.com.br/#sobre>. Acesso em: 27 out. 2017.

MOLINARI, Leonardo. **Testes funcionais de software**. Florianópolis: Visual Books, 2008.

NASCIMENTO, Gleison S. **Um Método para Descoberta Semi-Automática de Processos de Negócio Codificados em Sistemas Legados**. Tese (Doutorado em Ciência da Computação) – Programa de Pós-Graduação em Computação, UFRGS, Porto Alegre, 2014.

OBJECT MANAGEMENT GROUP (OMG). **MDA - The Architecture Of Choice For A Changing World**. Needham, USA, 2017. Disponível em: <http://www.omg.org/mda/>. Acesso em 12 out. 2017.

OSBORNE, W. M.; CHIKOFSKY, E.J.. **Fitting Pieces to the Maintenance Puzzle.** IEEE Software, January 1990, pp. 10-11.

PARADAUSKAS, B; LAURIKAITIS, A. **Business knowledge extraction from legacy information systems.** Journal of Information Technology and Control, 35(3), 214-221.

PÉREZ-CASTILLO, Ricardo; GUZMÁN, Ignacio G.R. de; PIATTINI, Mario. **Business process archeology using MARBLE**. Information and Software Technology. Elsevier. v. 53. 2011. p. 1023-1044.

PÉREZ-CASTILLO, Ricardo; GUZMÁN, Ignacio G.R. de; PIATTINI, Mario. Fundaments of Business Process Archeology. In: **Organizational Culture and Behavior: Concepts, Methodologies, Tools and Applications.**. USA: IGI Global, 2017. p. 1-19.

PEZZÉ, Mauro; YOUNG, Michal. **Teste e análise de software: processos, princípios e técnicas.** Porto Alegre: Bookman, 2008.

PRESSMAN, Roger S. **Engenharia de Software: uma abordagem profissional**. 7. ed., Porto Alegre: AGMH, 2011.

SCITOOLS. **Understand Educational License.** Utah, USA, 2017a. Disponível em: <https://scitools.com/student/>. Acesso em 28 out. 2017.

SCITOOLS. **Understand - User Guide and Reference Manual.** Utah, USA, 2017b. Disponível em: <http://scitools.com/documents/manuals/pdf/understand.pdf>. Acesso em 28 out. 2017.

SOMMERVILLE, Ian. **Engenharia de Software**, 9. ed., São Paulo: Pearson Prentice Hall, 2011.

YIN, Robert K. **Estudo de caso: planejamento e métodos**. 2. ed., Porto Alegre: Bookman, 2001.

ZOU, Y.; HUNG, M. An approach for extracting workflow from e-commerce applications. In: **International Conference Program Comprehension,** 14. 2006, Atenas: IEEE 2006, p. 127-136.

APÊNDICE

Questionário aplicado para o estudo de caso da Meta3

1. Sistemas Legados

a) Características comuns dos sistemas legados com que a empresa trabalha:

   1) Linguagens:
   2) Idade:
   3) Situação (ainda estão em uso, estão sendo melhorados ou substituídos):
   4) Tamanho:
   5) Perfil dos clientes:
   6) Os sistemas dos clientes costumam ter documentação?
   7) Quem desenvolveu os sistemas no cliente ainda está na organização?
   8) Esses sistemas são difíceis de compreender?

b) Por que a empresa usa arqueologia?

c) Por que os clientes estão contratando esse serviço da Meta3?

d) Que tipos de alterações são mais comuns?

   ( ) Inclusão de uma funcionalidade

   ( ) Correção de um bug

   ( ) Melhoria do projeto

   ( ) Otimização do uso de recursos

e) As alterações visam mais a refatoração ou otimização?

f) Por quais estratégias em geral os clientes optam?

   ( ) Descartar completamente o sistema

   ( ) Deixar o sistema inalterado e continuar com a manutenção regular

   ( ) Reestruturar o sistema para melhorar a sua manutenibilidade

   ( ) Substituir a totalidade ou parte do sistema por um novo

g) O código atual dos sistemas legados costuma ser muito diferente daquele originalmente concebido? O código legado em geral continua alinhado com os processos de negócio atuais?

h) Quais são as principais adversidades encontradas nos sistemas legados?

i) Existem muitas dependências entre componentes e estruturas nos sistemas legados?

j) Quais são os artefatos mais comuns encontrados nos sistemas legados (documentos, código-fonte, diagramas, etc.)?

k) Qual a principal fonte de informações sobre o sistema legado?

l) O que, em geral, se extrai do código-fonte?

m) É utilizada alguma técnica de análise de código-fonte (dinâmica/estática)?

n) Usam *parsers* para analisar instruções?

o) Como são extraídas as regras de negócio?

2. Sobre a ferramenta Meta3-Application Inspector

a) Em qual linguagem e para qual plataforma foi construída?

b) A ferramenta se adequa bem a qualquer tipo de software, idade ou tamanho?

c) Existe algum cenário em que ela não se mostra adequada?

d) O processo é automatizado, ou seja, o mesmo processo pode ser aplicado a vários sistemas legados?

e) Após a aplicação do uso da ferramenta, é necessária a análise mais apurada de algum especialista?

f) Como é o funcionamento geral da ferramenta?

3. Testes de software

a) Como são produzidos os casos de teste a partir do que foi extraído do sistema legado?

b) O que geralmente é testado (formulários, arquivos, telas, páginas, etc.)?

c) Os sistemas legados, em geral, tem algum teste? São criados novos casos de teste?

d) A escrita dos testes tem por base os requisitos que estão documentados?

e) Quais são os tipos de teste mais comuns realizados nos sistemas legados?