Universidade Estadual de Londrina

RODRIGO AUGUSTO IGAWA

UMA PROPOSTA DE REENGENHARIA COM ABORDAGEM INCREMENTAL PARA SISTEMAS ORIENTADO A OBJETO

LONDRINA–PR

2013

RODRIGO AUGUSTO IGAWA

UMA PROPOSTA DE REENGENHARIA COM ABORDAGEM INCREMENTAL PARA SISTEMAS ORIENTADO A OBJETO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof(a). Dr(a). Jandira Guenka Palma

LONDRINA–PR

2013

RODRIGO AUGUSTO IGAWA

UMA PROPOSTA DE REENGENHARIA COM ABORDAGEM INCREMENTAL PARA SISTEMAS ORIENTADO A OBJETO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Prof(a). Dr(a). Jandira Guenka Palma
Universidade Estadual de Londrina
Orientador

Prof. Dr. Segundo Membro da Banca
Universidade/Instituição do Segundo Membro da Banca

Prof. Dr. Terceiro Membro da Banca
Universidade/Instituição do Terceiro Membro da Banca

Londrina–PR, 24 de novembrode 2013

LONDRINA–PR

2013

AGRADECIMENTOS

Agradeço, em primeiro lugar, a Deus. Foi Ele quem ofereceu cada uma das minhas oportunidades até hoje, me fortaleceu e me deu coragem e força nos momentos mais necessários.

Agradeço à minha família pelos cuidados e incentivos recebidos até o dia de hoje. Quando pessoas especiais se encontram ao nosso lodo sempre podemos seguir em frente.

Sou imensamente grato à professora Jandira Guenka Palma por cada uma de suas orientações desde o começo desse trabalho. A experiência de aprendizado de extrema importância.

Agradeço à equipe de trabalho da empresa Guenka Software ao qual tive a chance de ter estado presente durante o ano todo. Em especial, agradeço ao Denis Sanches e ao Giovane Perdigão que estivem presentes em todos os momentos mais difíceis desse trabalho.

IGAWA, R. A.. **Uma Proposta de Reengenharia Com Abordagem Incremental Para Sistemas Orientado a Objeto**. 76 p. Trabalho de Conclusão de Curso (Graduação). Bacharelado em Ciência da Computação – Universidade Estadual de Londrina, 2013.

RESUMO

Todo software passa por modificações, denominadas manutenção de software, ao longo do tempo. Alguns exemplos de manutenção de software são: mudanças de requisitos de usuário, migração da versão de ferramentas do software, migração de plataforma e modificações corretivas de erros - muitas vezes, sem a preocupação com a arquitetura e a legibilidade do código do sistema, consequentemente, compromentendo a manutenibilidade. Como implicação, são obtidos projetos e codificações de má qualidade, com lógica pobre e documentação insuficiente para a compreensão do sistema. Para solucionar o problema com a manutenibilidade é usual indicar a reconstrução do sistema, ou seja, reegenharia de software. A justificativa para a reengenharia de software é simples: um software novo de grande porte é caro demais para ser desenvolvido do ínicio novamente, mas software antigo torna-se cada dia mais custoso para ser mantido e adapatado a novos clientes. O objetivo da reengenharia é reusar esforços de desenvolvimento passados a fim de reduzir gastos de manutenção e aumentar a flexibilidade do sistema. Portanto, este trabalho tem como base o método Fusion RE-I e o projeto FAMOOS que fazem uso do paradigma orientado a objetos. Outras bases do trabalho são: práticas de reestruturação de código de forma a torná-lo mais legível e uma abordagem de migração incremental.

Palavras-chave: reengenharia de software. migração de software incremental. refatoração de sistemas

IGAWA, R. A.. **A Proposal of Reengineering With Incremental Approach To Object Oriented Systems**. 76 p. Final Project (Undergraduation). Bachelor of Science in Computer Science – State University of Londrina, 2013.

ABSTRACT

As the time goes by, any software has to pass through changes that are denominated software maintenance. Some examples of software maintenance are: user requirements changes, migration of software version tools, platform migration and corrective modifications - often without the enough concern about architecture and readability of the code system, therefore, turning maintainability harder. As a result, projects and encodings of poor quality with poor logic and insufficient documentation for the understanding of the system are obtained. To solve the problem with the maintainability it's usual to indicate a system reconstruction: software reengineering. The justification for software reengineering is simple: a new large software is too expensive to be developed from the beginning again, but old software becomes increasingly costly to be maintained and customized for new clients. The objective of reengineering is reuse past development efforts in order to reduce maintenance costs and increase the flexibility of the system. Therefore, this work is based on the Fusion-RE/I method and FAMOOS project that make use of the object-oriented paradigm. Other bases of the work are: code restructuring practices in order to make it more readable and an approach to incremental migration .

Keywords: software reengineering. incremental software migration. software refactoring.

LISTA DE ILUSTRAÇÕES

Figura 1 – Um modelo de processo de reengenharia de software retirado de Pressman e Ince[2]. 19
Figura 2 – Visão geral do Fusion RE/I retirado de Feltrim[10]. 23
Figura 3 – Modelo de ciclo de vida na forma de máquina de estados retirado de Feltrim[10]. 24
Figura 4 – Modelo de objetos na forma de diagrama de classes retirado de Feltrim[10]. 25
Figura 5 – Modelo de quadro de chamadas na forma de diagrama de componentes retirado de Feltrim[10]. 25
Figura 6 – Um fluxograma para a criação de metas gerais da reengenharia retirado de Demeyer, Ducasse e Nierstrasz[3]. 28
Figura 7 – Um fluxograma para o primeiro contato com a aplicação retirado de Demeyer, Ducasse e Nierstrasz[3]. 30
Figura 8 – Mapeamento para heranças adaptado de Demeyer, Ducasse e Nierstrasz[3]. 32
Figura 9 – Mapeamento para cardinalidades. 33
Figura 10 – Processo de refatoração de código adaptado de Wake e Rutherford[14]. 38
Figura 11 – Exemplo de arquitetura da migração *Chiken Little* adaptado de Brodie e Stonebraker[20]. 42
Figura 12 – Proposta da abordagem WARE adaptado de Lucca, Fasolino e Tramontana[23]. 46
Figura 13 – Exemplo de arquitetura da migração *Butterfly* adaptado de Oliveira e Marcelino[21]. 47
Figura 14 – Visão geral do processo de reengenharia proposto 49
Figura 15 – Recuperação dos requisitos funcionais do sistema 50
Figura 16 – Recuperação da estrutura interna do sistema 51
Figura 17 – Visão geral da abordagem incremental. 53
Figura 18 – Quadro do número de linhas do sistema segundo o *metric_fu* 56
Figura 19 – Quadro de arquivos mais alterados do sistema no ultimo ano segundo o *metric_fu* 63
Figura 20 – Quadro de duplicação de código segundo o *metric_fu* 64
Figura 21 – Modelo de ciclo de vida do SGI na forma de caso de uso. 65
Figura 22 – Modelo de operações do SGI na forma de caso de uso. 66
Figura 23 – Parte da especificação de uma operação do modelo ciclo de operações do SGI. 67
Figura 24 – Exemplo de reestruturação do SGI. 68

Figura 25 – Exemplo da ferramenta de *debug* do SGI. 68
Figura 26 – Estado antecessor à refatoração para jQuery. 69
Figura 27 – Estado posterior à refatoração paga jQuery. 70
Figura 28 – Refatoração para HTML5. 71
Figura 29 – Menu do SGI. 71
Figura 30 – Organização dos componentes para o caso do módulo de estoque 72

LISTA DE TABELAS

Tabela 1 – Mapeamento Fusion RE/I para UML adaptado de Feltrim[10]. 24

LISTA DE ABREVIATURAS E SIGLAS

TI Tecnologia da informação

UML Unified Modeling Language

FAMOOS Framework-based Approach for Mastering Object-Oriented Software Evolution

ERP Enterprise Resource Planning

SGI Sistema de Gestão Integrada

MVC Model View Controller

DRY Don't Repeat Yourself

COC Convention Over Configuration

SUMÁRIO

1 INTRODUÇÃO 13
1.1 Considerações iniciais 13
1.2 Motivação 13
1.3 Objetivos 14
1.3.1 Objetivos específicos 14

2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 Manutenção de software 15
2.2 Reengenharia de Software 18
2.3 Etapas da Reengeharia de software 18
2.3.1 Análise de inventário 18
2.3.2 Reestruturação de documentos 19
2.3.3 Engenharia reversa 20
2.3.4 Reestruturação de código 20
2.3.5 Reestruturação de dados 20
2.3.6 Engenharia Avante 21
2.4 Métodos de engenharia reversa 22
2.4.1 Método Fusion-RE/I 22
2.4.1.1 Recuperação dos requisitos funcionais 22
2.4.1.2 Recuperação dos requisitos estruturais 23
2.4.1.3 Proposta de processo de execução do Fusion RE/I 26
2.4.2 Engenharia reversa do projeto FAMOOS 27
2.4.2.1 Criação de metas gerais da reengenharia 27
2.4.2.2 Primeiro contato com a aplicação 29
2.4.2.3 Compreensão inicial da estrutura do sistema 31
2.4.2.4 Captura detalhada dos modelos do sistema 35
2.5 Práticas de reestruturação de código 37
2.5.1 Refatoração de código 37
2.5.2 Ciclo básico de refatoração de código 37
2.5.3 Potenciais problemas em código 39
2.5.3.1 Comentários 39
2.5.3.2 Métodos Longos 39
2.5.3.3 Código duplicado 40
2.5.3.4 Classes Longas 40
2.5.3.5 Longa lista de parâmetros 40

2.5.3.6 Nomes problemáticos . . . 41
2.6 Migração de software . . . 42
2.6.1 A estratégia Chicken Little . . . 42
2.6.1.1 Analisar o sistema legado . . . 43
2.6.1.2 Decompor a estrutura do sistema . . . 43
2.6.1.3 Projetar a interface do sistema destino . . . 44
2.6.1.4 Projetar a aplicação destino . . . 44
2.6.1.5 Projetar o esquema do banco de dados destino . . . 44
2.6.1.6 Criar e instalar os *gateways* necessários . . . 44
2.6.1.7 Migrar o banco de dados legado . . . 44
2.6.1.8 Migrar a aplicação legada . . . 44
2.6.1.9 Migrar interfaces legadas . . . 45
2.6.1.10 Mudar para o sistema destino . . . 45
2.7 Trabalhos relacionados . . . 46
2.7.1 Engenharia reversa baseada em uso de ferramentas automáticas 46
2.7.2 Migração de software sem uma abordagem incremental . . . 46

3 PROPOSTA DE UM PROCESSO DE REENGENHARIA ORIENTADO A OBJETOS INCREMENTAL - PROO-I . . . 48
3.1 Etapas do processo de reengenharia PROO-I . . . 49
3.1.1 Engenharia reversa do PROO-I . . . 49
3.1.1.1 Recuperação dos requisitos funcionais do sistema . . . 49
3.1.1.2 Recuperação da estrutura interna do sistema . . . 51
3.1.2 Reestruturação do sistema . . . 52
3.1.3 Engenharia avante: abordagem incremental . . . 52
3.1.3.1 Decompor a estrutura do sistema . . . 53
3.1.3.2 Projetar a aplicação destino . . . 54
3.1.3.3 Projetar o esquema do banco de dados do sistema destino . . . 54
3.1.3.4 Criar e instalar os *gateways* necessários . . . 54
3.1.3.5 Migrar o banco de dados legado . . . 54
3.1.3.6 Migrar a aplicação legada . . . 54
3.2 Estudo de caso do processo proposto em um sistema corporativo 55
3.2.1 Ambiente de aplicação da proposta . . . 55
3.2.2 Motivação para reengenharia . . . 55
3.2.3 Materiais e métodos . . . 57
3.2.4 Aplicação da proposta . . . 58
3.2.4.1 Aplicação da etapa de engenharia reversa . . . 58
3.2.4.2 Reestruturação do sistema . . . 59
3.2.4.3 Proposta incremental do primeiro módulo . . . 60
3.3 Análise de resultados . . . 61

4 CONCLUSÃO 73
4.1 Trabalhos Futuros 74

Referências 75

1 INTRODUÇÃO

1.1 Considerações iniciais

Todo software evolui com o tempo, independentemente de sua categoria, de seu tamanho ou de sua complexidade. As modificações, caracterizadas como manutenção de software, orientam essa evolução [1].

Para empresas, é desejado que o processo de manutenção de software seja fácil, da forma mais ágil e menos onerosa possível. No entanto, a quantidade de problemas que envolvem manutenção de software cresce com o passar do tempo [2].

Na década de 90 sistemas implementados sob o paradigma orientado a objetos já apresentavam problemas como manutenções onerosas e difíceis de se efetuar, paralelamente, havia ainda a dificuldade de adaptação e extensão de funcionalidades. Falta de documentação, rotatividade da equipe de desenvolvimento e pouco domínio do paradigma orientado a objeto faziam com que os sistemas tornassem a característica evolutiva do software ainda mais custosa [3].

Para tornar o processo de manutenção do software menos oneroso é comum que se indique a reconstrução do mesmo. Aplica-se então, um processo de reengenharia, que utiliza-se de ferramentas e métodos que buscam reutilização e legibilidade do código da aplicação para melhorar a adpatação e extensão do sistema [2].

A justificativa para a reengenharia de software é simples: um software novo de grande porte é caro demais para ser desenvolvido do ínicio novamente, mas software antigo torna-se cada dia mais custoso para ser mantido e adapatado a novos clientes. O objetivo da reengenharia é reusar esforços de desenvolvimento passados a fim de reduzir gastos de manutenção e aumentar a flexibilidade do sistema.

Diante disso, estudos sobre a aplicação da reengenharia de software têm sido realizados. Existem métodos de reengenharia propostos para sistemas que são procedimentais ou orientados a objetos. Este trabalho tem como base o método Fusion RE-I e o projeto FAMOOS que fazem uso do paradigma orientado a objetos. Outras bases do trabalho são: práticas de reestruturação de código de forma a torná-lo mais legível e uma abordagem de migração incremental.

1.2 Motivação

As dificuldades do processo de manutenção enfrentadas por uma empresa de software motivaram a realização desse trabalho. Essas dificuldades foram geradas pelo one-

rosidade da manutenção de um dos sistemas mantidos pela empresa, denominado SGI (Sistema de Gestão Integrada). Portanto, a intenção inicial desse trabalho era a reestruturação do mesmo, de forma a facilitar a adaptação e a extensão de funcionalidades do mesmo.

Entretanto, devido à escassez de documentação, rotatividade dos desenvolvedores da empresa, que detinham de grande do conhecimento técnico do sistema, não era possível reestruturá-lo de forma que os requisitos não se perdessem. Era necessário também recuperar o conhecimento do sistema em nível de estrutural e em nível de requisitos.

Portanto, foi tomada a decisão de submeter o SGI a um processo de reengenharia de software. Foi decidido desde então que o processo a ser aplicado deveria compor 3 etapas: uma engenharia reversa para a recuperação do conhecimento necessário; uma reestruturação do sistema de forma que as melhores partes do sistema possam ser reaproveitadas e uma etapa de migração de software incremental, pois o sistema é grande demais para ser reconstruído do início.

1.3 Objetivos

Propor um processo de reengenharia de software orientado a objetos, baseado no método Fusion-RE/I, no projeto FAMOOS, em práticas de refatoração e no processo de migração incremental *Chicken Little*. Em seguida, aplicar o processo proposto em um sistema corporativo com o intuito de reestruturá-lo.

1.3.1 Objetivos específicos

- Estudar os fundamentos de reengenharia de software;
- Estudar o método Fusion-RE/I e o projeto FAMOOS;
- Estudar práticas de refatoração;
- Estudar a migração incremental *Chicken Little*;
- Adquirir experiência a fim de atender às necessidades do mercado relacionadas à reengenharia de software;
- Analisar os resultados obtidos através da aplicação do processo proposto no sistema SGI.

2 FUNDAMENTAÇÃO TEÓRICA

Software é um conjunto de instruções de máquina que quando executados produzem uma funcionalidade desejada com o uso de estruturas de dados que auxiliam a transformação da informação de acordo com a necessidade [2].

Nesse trabalho o foco são as aplicações web, que é um tipo de software que apresenta características únicas em relação aos demais, algumas dessas características são [1]:

- Carga imprevisível de tarefas: o número de usuários da aplicação pode variar de acordo com o momento do dia sem um padrão.
- Sensível ao conteudo: A qualidade e natureza estética do conteúdo permanece um importante determinante da qualidade de uma aplicação web.
- Evolução constante: Ao contrário do software aplicativo convencional que se desenvolve ao longo de uma série de lançamentos previstos, cronologicamente espaçados, aplicações web evoluem continuamente. Não é incomum para algumas aplicações (especificamente, o seu conteúdo) a serem atualizados em um cronograma minuto a minuto ou de conteúdo a ser calculado de forma independente para cada solicitação.
- Performance: embora velocidade não seja um ponto forte em aplicações web, a aplicação deve ser rápida para tentar compensar o tempo gasto com a própria rede.

Assim como todos os tipos de software, uma aplicação web independente de seu tamanho e complexidade é alterada com o passar do tempo. Essas mudanças fazem parte da evolução do software para se adaptar a um novo ambiente com novas funcionalidade afim de atender os pedidos efetuados pelos clientes. Essa evolução é chamada de manutenção de software e é descrita na próxima sessão.

2.1 Manutenção de software

Estudos iniciados na década de 70 concluem que a manutenção de software é um resultado do caráter evolutivo do software. Sabe-se que a evolução das aplicações são baseadas em algumas leis como citadas a seguir [1]:

- A manutenção do sistema é um processo inevitável. Quando o ambiente do sistema muda, surgem novos requisitos e o sistema tem de ser modificado. Quando o sistema modificado é reintroduzido ao meio ambiente, mais mudanças ambientais ocorrem, de modo que o processo de evolução começa novamente.

- Quando um sistema é alterado, a sua estrutura é degradada. A única maneira de evitar que isso aconteça é investir em manutenção preventiva. Gasta-se tempo para melhorar a estrutura de software sem aumentar a sua funcionalidade. Naturalmente, isso significa custos adicionais para além das de implementação de mudanças de sistema.

- Grandes sistemas têm uma dinâmica própria de evolução de acordo com o ambiente e isso é estabelecido numa fase inicial no processo de desenvolvimento. Isso determina as tendências brutas do processo de manutenção do sistema e limita o número de possíveis alterações no sistema.

- Adicionar novas funcionalidades ao sistema, inevitavelmente introduz novas falhas. Quanto mais funcionalidades adicionadas em uma versão, mais falhas. Por conseguinte, um grande incremento de funcionalidade em um sistema deliberado para produção significa que este terá que ser seguido por uma outra versão na qual as novas falhas do sistema são reparadas.

Tais leis levam a aplicação a sofrer mudanças após ser entregue ao cliente sendo classificadas em: [1]

- Manutenção corretiva: são as correções de codificação ou erros de projeto;

- Manutenção adaptativa: são as mudanças necessárias quando o software precisa se adaptar a alguma mudança que que ocorrido no ambiente que opera;

- Manutenção evolutiva: são as mudanças que ocorrem em respostas à evolução dos negócios

As pesquisas concordam amplamente que a manutenção do software ocupa uma proporção maior dos orçamentos de T.I. do que de novo desenvolvimento (manutenção de cerca de dois terços, um terço de desenvolvimento) [1].

Quando a manutenção de software se torna muito custosa para uma empresa, pode ser o momento de considerar uma reengenharia de software [2, 1].

Não é um único motivo que leva uma empresa a analisar se o software em questão deve passar por um processo de reengenharia, mas sim um conjunto de sintomas que aparecem no decorrer do dia a dia com a equipe de desenvolvimento [2, 1, 3]. Exemplos de sintomas são listados abaixo:

- Número de pedidos de manutenção corretiva: um aumento no número de relatos de *bugs* e falhas podem indicar que mais erros estão sendo introduzidos no programa que estão sendo reparados durante o processo de manutenção. Isso pode indicar um declínio na manutenção [1];

- Tempo médio necessário para a análise de impacto: isso reflete o número de componentes do programa que são afetados pelo pedido de alteração. Se esse tempo aumenta, isso implica mais e mais componentes são afetados e manutenção é decrescente [1];

- O tempo médio necessário para executar um pedido de mudança: isto não é o mesmo que o tempo de análise de impacto embora possa correlacionar com ele. Esta é a quantidade de tempo que precisa-se para modificar o sistema e sua documentação, depois de ter avaliado os componentes que são afetados. Um aumento no tempo necessário para implementar uma mudança pode indicar uma redução de manutenção [1].

- Muito tempo para fazer alterações simples: o sistema agora é tão complexa que até mesmo mudanças simples são difíceis de implementar. Se demorar muito tempo para fazer mudanças simples em seu sistema, ele certamente estará fora de questão fazer mudanças complexas. Se houver um acúmulo de mudanças simples à espera de ser feito, então você nunca vai chegar aos problemas difíceis [1];

- Necessidade de correções constantes: erros nunca parecem ir embora. Toda vez que corrigir um erro, um novo aparece ao lado dele. Isto informa que partes da sua aplicação tornaram-se muito complexa. Além disso, o arquitetura da aplicação já não corresponde às necessidades, por isso mesmo pequenas mudanças vão ter consequências inesperadas[3];

- Dependências de manutenção: quando corrigido um erro em um lugar, outro erro aparece em outro lugar. Isso é muitas vezes um sinal de que a arquitetura tem-se deteriorado ao ponto onde os componentes logicamente separados do sistema não são mais independentes [3];

- Dificuldades de separação de produtos: se existe muitos clientes para o produto, mas existe também a dificuldade em adaptar lançamentos para cada cliente, a arquitetura já não é mais eficiente[3].

Baseado no conjunto de sintomas apresentados anteriormente, a equipe responsável pelo software decide então se é o momento de se pensar na reengenharia.

2.2 Reengenharia de Software

Uma aplicação serviu as necessidades de negócio da empresa por anos. Durante esse tempo, foi corrigida, adaptada e aperfeiçoada muitas vezes. Todo trabalho referente à aplicação, foi feito com a melhor dos esforços da equipe de desenvolvimento para atender as demandas do cliente, mas boas práticas de engenharia de software foram deixadas de lado. Agora, a aplicação está instável. Ainda funciona, mas efeitos colaterais inesperados ocorrem a cada tentativa de modificacão. Pode ser também que a aplicação esteja estável, mas com altos custos de atualização. Um cenário comum que se aplica a diversas empresas e que caracteriza o momento de se pensar em efetuar um processo de reengenharia de software [2].

A principal justificativa para se pensar em reestruturar o sistema no cenário acima descrito é: um novo sistema que atenda os mesmos requisitos do anterior leva muito tempo para ser construído, e manter um sistema que não condiz com o orçamento da empresa também não é interessante [4].

Por fim, efetuar o processo de reengenharia em uma aplicação implica em examinar e alterar um sistema com o objetivo de reconstituí-lo em um novo modelo e, a posterior, uma implementação de nova forma [5, 3].

2.3 Etapas da Reengeharia de software

O ciclo de vida de um processo de reengenharia de software pode ser representado através da figura 1, onde percebemos que o modelo representa um conjunto de atividades cíclicas. Portanto, atividades podem ser refeitas e não ocorrem necessariamente na ordem proposta, pois, a engenharia reversa pode ser necessária antes da reestruturação de documentos dependendo da situação encontrada [2, 3].

2.3.1 Análise de inventário

Toda empresa de software deveria ter um inventário de todas as aplicações. O inventário pode ser nada mais do que um modelo de planilha contendo informações que fornece uma descrição detalhada (por exemplo, tamanho, idade, criticidade de negócios) de cada aplicativo ativo. Classificando essas informações de acordo com a criticidade do negócio, longevidade, facilidade de manutenção corrente e suporte, os candidatos para a reengenharia podem aparecer. Os recursos então, devem ser alocados para aplicações candidatos para o trabalho de reengenharia.

É importante notar que o inventário deve ser revisto em um ciclo regular. O estado de aplicações (por exemplo, a criticidade de negócios) pode mudar como uma função do tempo, e, como resultado, as prioridades de reengenharia mudarão.

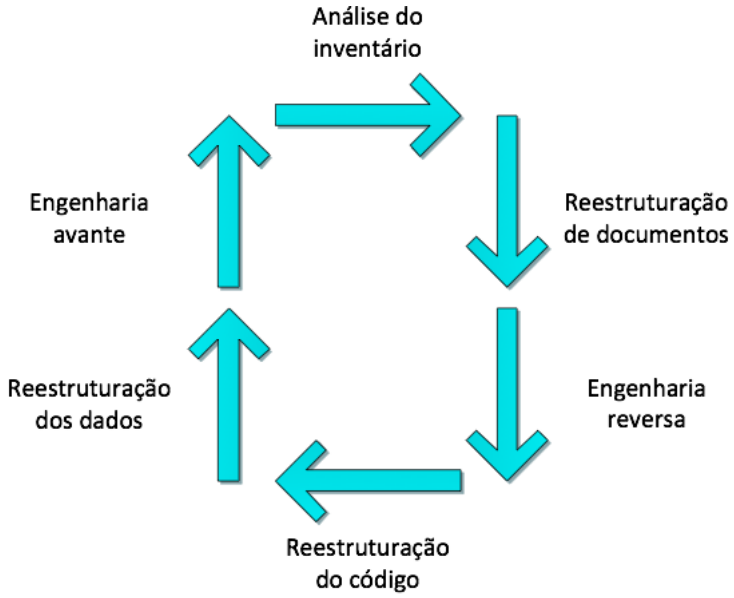

Figura 1 – Um modelo de processo de reengenharia de software retirado de Pressman e Ince[2].

2.3.2 Reestruturação de documentos

Problemas de documentação em softwares que necessitam de reengenharia é algo muito comum, nessa situação as seguintes decisões podem ser efetuadas [2]:

- **Manter a documentação**: criar uma documentação é um processo muito demorado. Se o sistema funciona, pode-se optar por continuar com o que existe documentado. Em alguns casos, esta é a abordagem correta. Não é possível re-criar a documentação de centenas de programas de computador. Se um programa é relativamente estático, provavelmente está chegando ao final de sua vida útil, e é improvável que passar por uma mudança significativa.

- **Atualizar a documentação**: A documentação deve ser atualizado, mas a empresa tem recursos limitados. Pode não ser necessário para uma aplicação ser completamente redocumentada nesse momento. Em vez disso, as partes do sistema que estão atualmente passando por mudanças são totalmente documentados. Com o tempo, uma coleção de documentação útil e relevante vai evoluir.

- **Redocumentar**: O sistema é crítico para a empresa e deve ser totalmente redocumentado. Mesmo neste caso, uma aproximação inteligente é limitar a documentação a um mínimo essencial. Pois, se o sistema é extremamente importante para a organização, excessos nem falta de documentação podem ocorrer.

2.3.3 Engenharia reversa

O termo engenharia reversa tem suas origens no mundo do hardware. A empresa desmonta um produto de hardware de uma concorrente com o objetivo de compreender o projeto de seu concorrente e os detalhes importantes de fabricação. Esses detalhes poderiam ser facilmente compreendidos se o projeto do concorrente e as especificações de fabricação fossem obtidos. Mas esses documentos são de propriedade privada e indisponíveis para a empresa que está a fazer uma engenharia reversa. Em essência, uma engenharia reversa bem sucedida resulta em uma ou mais especificações de concepção e de fabricação de um produto [2].

A engenharia reversa de software é bastante similar. Na maioria dos casos, no entanto, o programa a passar por uma engenharia reversa não é o de um concorrente. Pelo contrário, é de trabalho da própria empresa (muitas vezes feito anos antes). Os segredos para serem compreendidos são obscuros porque nenhuma especificação já foi desenvolvida. Portanto, a engenharia reversa para o software é o processo de análise de um programa, em um esforço para criar uma representação do programa em um nível maior de abstração do que o código-fonte. Engenharia reversa é um processo de recuperação de *design*. Ferramentas de engenharia reversa extraem dados, arquitetural e informações de projeto procedural de um programa existente [6, 2].

2.3.4 Reestruturação de código

Algumas vezes os sistemas que passam por um processo de reengenharia tem parte de seu custo devido a um código que, apesar de funcionar corretamente, está escrito de uma forma que a compreensão se torna difícil. Nesses casos, o código pode ser reestruturado de forma de boas práticas de programação sejam introduzidas em partes de código que são relevante para a o funcionamento da aplicação. Na maioria das vezes é uma atividade que se inicia em um processo de engenharia reversa. Porém sempre que se desenvolva uma nova funcionalidade para a aplicação é interessante que o código seja novamente avaliado para saber se não é interessante reestruturar o código [7, 2].

2.3.5 Reestruturação de dados

Um programa com arquitetura de dados ineficiente ou falha será difícil se adaptar e melhorar. De fato, para muitas aplicações, arquitetura de informação tem mais a ver com a viabilidade a longo prazo de um programa que o próprio código fonte.

Ao contrário de reestruturação de código, que ocorre em um nível relativamente baixo de abstração, a reestruturação de dados é uma atividade de reengenharia de grande escala. Objetos e atributos de dados são identificados, e as estruturas de dados existentes são revisados para a qualidade.

Como a arquitetura de dados tem uma forte influência na arquitetura do programa e os algoritmos que o interagem, as alterações nas estrutura dos dados serão, invariavelmente, resultados em alterações de código [2].

2.3.6 Engenharia Avante

Engenharia avante tem como objetivo principal usar todo o conhecimento aprendido do software durante as etapas anteriores para propor melhorias no sistema. Na maioria das vezes, essas melhorias não são apenas percebidas através de melhores performances, mas arquiteturas mais extensíveis e códigos que apresentem boas práticas de programação e portanto facilitem a manutenção, extensão e reusabilidade do software [2, 3].

2.4 Métodos de engenharia reversa

Esta seção apresenta o método FUSION-RE/I e o método do projeto FAMOOS. Esses dois métodos de engenharia reversa são as bases de engenharia reversa do trabalho.

2.4.1 Método Fusion-RE/I

Método Fusion-RE/I: é um método de engenharia reversa criado inicialmente em 1996 para sistemas procedurais [8]. No ano seguinte, o método foi aperfeiçoado e o mesmo passou a ser chamado de Método Fusion-RE/I [9]. Por fim, em 1999 o método foi mais uma vez estudado e passou a ser possível usar UML na documentação resgatada pelo método FUSION RE/I. Além disso as duas primeiras versões do método tratavam-se de uma abordagem de transformação de sistemas procedurais para o orientado a objetos. A última versão do método proposta além de possibilitar o uso de UML no lugar do Fusion, propôs uma forma de aplicação do método para sistemas que já fossem orientado a objetos [10].

O método Fusion RE/I foi usado como base para o estudo de caso do trabalho em questão, porém, foi usado a forma mais recente do método proposta no ano de 1999. Portanto, é apresentado a seguir somente a forma como o método foi proposto em sua última forma que usa somente UML e não mais a notação Fusion.

O Fusion RE/I constitui um método de engenharia reversa que, ao visar facilitar o processo, parte da interface do sistema para a recuperação de informações úteis à manutenção de software. Através desse procedimento é possível recuperar os requisitos funcionais e os detalhes de estrutura do sistema. Dessa forma, parte-se de considerações lógicas obtidas na interface da aplicação para a compreensão dos requisitos funcionais do sistema e, posteriormente, parte-se dessas visões recuperadas e de considerações feitas a partir do código fonte do sistema para compreender detalhes estruturais do projeto [8], uma visão geral é apresentada na figura 2.

2.4.1.1 Recuperação dos requisitos funcionais

Em relação aos requisitos funcionais do sistema, o método recupera os seguintes itens:

- Modelo de ciclo de vida tem como objetivo representar a forma como o sitema se comporta com o ambiente e os eventos que o ocorrem durante o uso do programa. Aborda o conjunto de operações o sistema pode fazer, as entradas que podem ser recebidas e os estados que a aplicação assume para cada uma das entradas ou operações [8]. A representação desse item na forma de UML também pode ser feita diretamente através de um diagrama de estados. Um driagrama de estados é um grafo

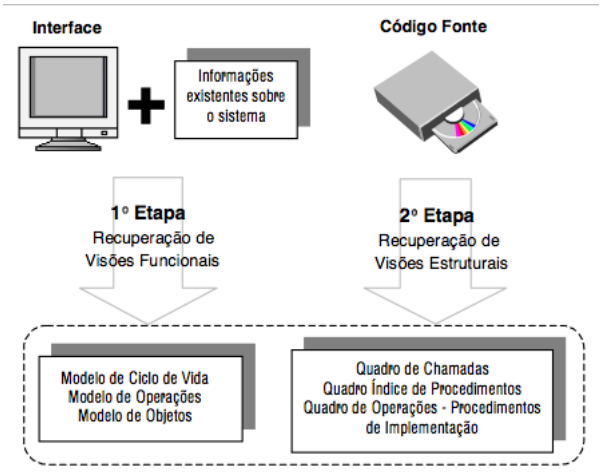

Figura 2 – Visão geral do Fusion RE/I retirado de Feltrim[10].

que representa uma máquina de estados que por sua vez descreve uma seqüência de estados que um objeto ou interação assume ao longo de sua execução em reposta a eventos juntos com seus resultados e ações [10].

- Modelo de operações tem como objetivo descrever as formas de comportamento de um sistema de forma mais declarada a partir de um fluxo escolhido. Parte-se do diagrama de estados obtido para o modelo de ciclo de vida que mostra os possíveis usos do sistema para, posteriormente, a escolha de um fluxo do sistema para ser detalhado da seguinte forma [8]:

  – nome da operação;

  – descrição informal da operação;

  – valores a que a operação pode ter acesso;

  – valores que a operação pode modificar;

  Uma vez que o modelo ciclo de vida, exemplo mostrado na figura 3, foi mapeado diretamente para um diagrama de estados, já temos uma lista de todas as operações iniciais do sistema que deveriam, nessa fase da documentação, serem melhor detalhadas. Para cada ação que é listada no modelo ciclo de vida teria-se um novo diagrama de estados para tratar todas as possibilidades daquela ação [10].

Por fim, os detalhes dos requisitos funcionais do sistema estão prontos e o mapeamento de Fusion para UML fica na forma [10]:

2.4.1.2 Recuperação dos requisitos estruturais

Em relação aos detalhes de estrutura física do sistema, o método recupera:

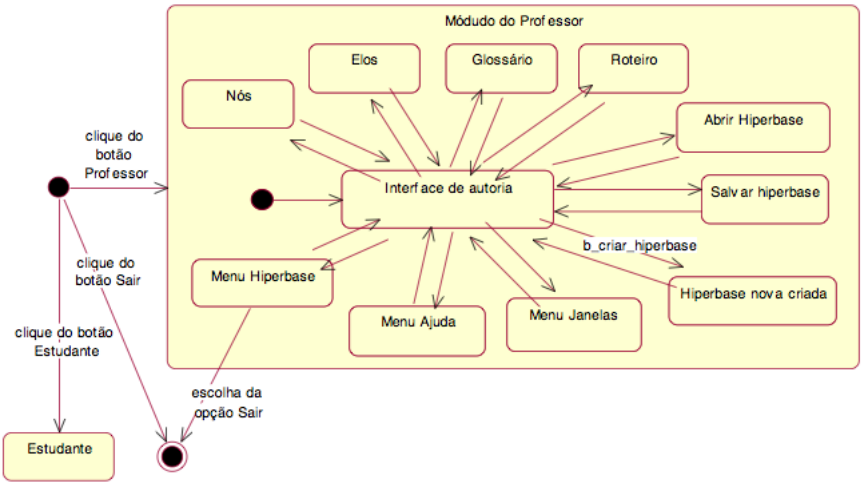

Figura 3 – Modelo de ciclo de vida na forma de máquina de estados retirado de Feltrim[10].

Tabela 1 – Mapeamento Fusion RE/I para UML adaptado de Feltrim[10].

Mapeamento Fusion x UML	
Modelo de ciclo de vida	Diagrama de estados
Modelo de operações	Diagrama de estados

- Modelo de objetos, mostrado na figura 4, tem como objetivo representar conceitos existentes no domínio do sistema e o relacionamento que cada elemento do sistema possui entre si. A notação foi originalmente derivada do modelo entidade relacionamento sendo composta por classes, relacionamentos, atributos, agregações, especializações e generalizações [8]. No UML, fica fácil tratar o modelo de objetos uma vez que o diagrama de classes representa exatamente o que o modelo de objetos da notação Fusion, porém com mais possibilidades que o anterior que não tratava casos de dependências entre classes [10].

- Quadro de chamadas: trata-se de uma representação onde as dependências de cada arquivo no sistema são demonstradas para a análise de relevância de cada arquivo. No UML, o diagrama de componentes é capaz de identificar relações de depêndencias entre os pacotes. É possível analisar dependências entre arquivos de código fonte e binários que são as dependências físicas mais relevantes a serem estudadas. Pode-se usar o diagrama de componentes para representar arquivos de código fontes de classes [10]. Um exemplo é apresentado na figura 5.

- Quadro de índice: é um índice de ordem alfabética para critério de organização de todos os métodos que o Sistema possui. No UML, não há necessidade de um mapeamento direto entre essas duas notações, portanto, fica aberta uma representação

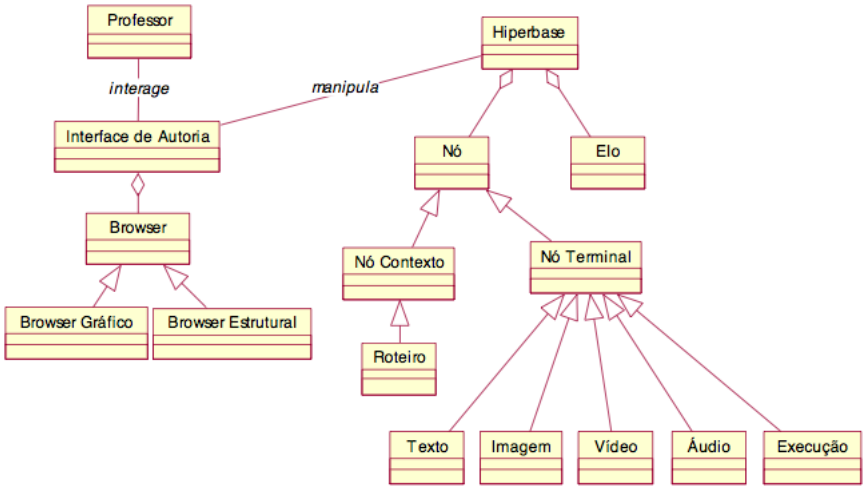

Figura 4 – Modelo de objetos na forma de diagrama de classes retirado de Feltrim[10].

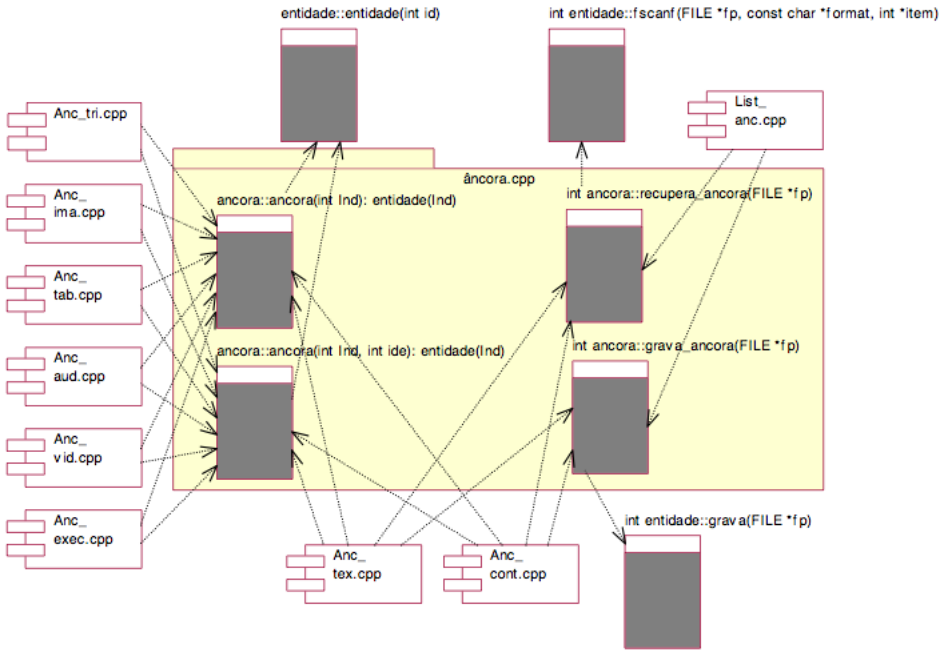

Figura 5 – Modelo de quadro de chamadas na forma de diagrama de componentes retirado de Feltrim[10].

para esse item [10].

- Quadro de operações de implementação: Assim como o item anterior este item não possui uma representação específica no UML e, por isso, fica em aberto. Trata-se de uma tabela que representa qual classe implementa qual interface [10].

2.4.1.3 Proposta de processo de execução do Fusion RE/I

O método Fusion RE/I foi usado em um sistema corporativo e apresentou resultados positivos, pois ao usar UML percebeu-se que a modelagem foi mais abrangente do que com os recursos permitidos pelo formalismo do antigo Fusion. O roteiro da ordem de recuperação de cada um dos itens seria [10]:

- Recuperação de visões Funcionais:
  1. Obtenção de documentação existente sobre o sistema;
  2. Recuperação do modelo de ciclo de vida;
  3. Elaboração do modelo de operações;
  4. Elaboração dos temas de módulos.
- Recuperação de visões Estruturais do sistema:
  1. Recuperação do modelo de objetos;
  2. Recuperação do quadro de chamadas;
  3. Recuperação do quadro índice de procedimentos;

2.4.2 Engenharia reversa do projeto FAMOOS

O projeto FAMOOS foi um projeto que foi executado na Europa no final dos anos 90. Nessa época, as empresas enfrentavam um problema clássico: os sistema eram grandes, possuiam código difícil de ser compreendido, não era fácil estender funcionalidades. A única diferença para os problemas antes enfrentados é que dessa vez o paradigma usado era o paradigma orientado a objetos e não mais o paradigma procedural. O objetivo principal do projeto era resolver esse problema ao desenvolver uma forma de reestruturar esse sistema de maneira que fosse possível estender e dar manutenção com mais facilidade [3].

O projeto teve bons resultados pois não foi aplicado em apenas uma empresa, mas em muitos países da Europa. A cada novo lugar em que se executava o projeto percebia-se que algumas similaridades existiam e que, embora estivesse em países diferentes, o sistema se tornava legado devido ao próprio caráter evolutivo do software que fazia a arquitetura inicial do sistema obsoleta devido à grande demanda de mudança [3].

Nessa seção, apresenta-se o modelo de engenharia reversa que a equipe do projeto refinou e escreveu após execução e a análise de resultados da reengenharia aplicada. A recuperação de informações do sistema se baseia em 4 grandes passos:

1. Criação de metas gerais da reengenharia;
2. Primeiro contato com a aplicação;
3. Compreensão inicial da estrutura do sistema;
4. Captura detalha dos modelos do sistema.

2.4.2.1 Criação de metas gerais da reengenharia

O problema inicial no processo de reengenharia é a quantidade de assuntos a serem discutidos, problemas a serem solucionados e interesses de tanto clientes como desenvolvedores. Portanto, a criação de metas pode ser representada como um grupo de sub passos onde a organização inicial da equipe que efetuará o projeto de reengenharia será estabelecida.

O primeiro passo é estabelecer objetivos comuns à todos os participantes. Desenvolvedores gostariam que o trabalho de manutenção seja feito com mais facilidade, usuários gostariam de mais e mais funcionalidades à todo momento [3]. A solução para manter desenvolvedores sempre unidos perante um conflito de interesses é estabelecer regras a serem seguidas por todos, pois são uma forma de coesão para equipes em projetos, mesmo que sejam regras simples como: todos são responsáveis por testes e *debugs* [11].

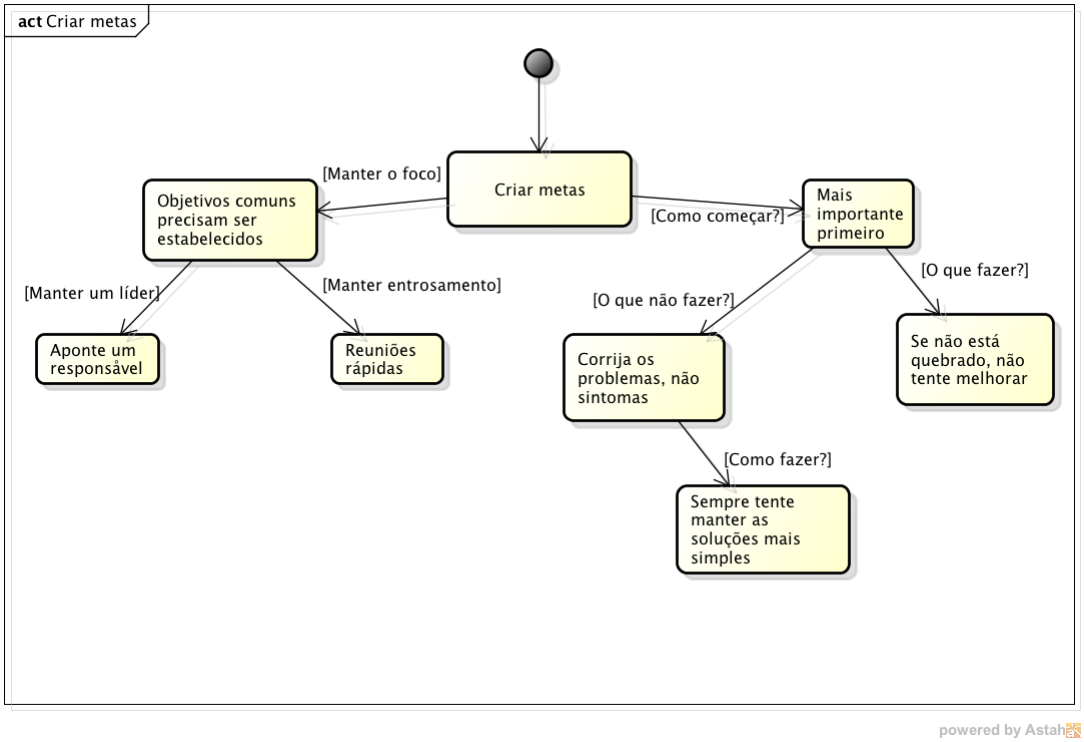

Figura 6 – Um fluxograma para a criação de metas gerais da reengenharia retirado de Demeyer, Ducasse e Nierstrasz[3].

Em uma próxima etapa, apontar um líder para manter e guiar o grupo através dos objetivos comuns estabelecidos se torna essencial, o objetivo do processo de reengenharia é fazer o sistema atual se tornar mais extensível de forma que sobreviva por mais anos, a melhor maneira de proteger esse objetivo é manter um líder. No projeto FAMOOS o responsável do projeto é comparado com um homem que com o uso de uma tocha guia sua equipe em uma caverna [3].

Nesse momento, o projeto de reengenharia tem algumas metas e já possui um representante. O que falta então é um encontro periódico para manter todos os integrantes da equipe alinhados. Uma equipe que está efetuando um procedimento de engenharia reversa encontra a cada dia de trabalho informações importantes sobre a aplicação. Esses conhecimentos devem ser compartilhados, pois a recuperação de informação é objetivo principal da engenharia reversa e, portanto, deve ser compartilhada por todos os membros. Uma reunião breve, onde os integrantes troquem conhecimentos adquiridos basta para fazer com o que um integrante não gaste tempo descobrindo algo que outro integrante já tenha descoberto. O objetivo da reunião não é gastar tempo nem dinheiro, mas evitar mais gastos [3, 12].

Uma segunda etapa da criação de metas gerais é estabelecer qual a parte da aplicação é a mais importante e que será trabalhada primeiro. Trabalhar a parte mais importante

do sistema primeiro pode ser uma abordagem arriscada, mas é ao mesmo a tempo a que implica em benefícios em menor tempo. Para identificar a parte mais relevante usa-se a informação que o cliente já tem fornecido. As partes que os clientes mais mandam requisições de correções são provavelmente as partes que mais importam para os usuários. Dessa forma acredita-se que não haverá gastos com partes irrelevantes do sistema [3].

Duas coisas a partir desse momento devem ser decididas durante todo o processo de reengenharia [3]:

- Corrija os problemas, não os sintomas: apesar de genérico, essa frase tem um significado de maior peso quando se está em um processo de reestruturação do sistema. Na situação que se enfrenta, o software é grande e possui muitas falhas nas mais diversas partes e funcionalidades. Com toda certeza, clientes abrem pedidos de correção, os sintomas. Concentrar-se em efetuar uma correção é dever da equipe responsável pelo sistema e isso é fato. Porém, trabalhar em cima de correções pode ser a escolha errada. Muitos dos pedidos de correções do sistema podem ser fruto de uma falha específica na arquitetura do sistema e esse é o verdadeiro problemas que implica em pedidos de correções.
- Se não está quebrado, não tente melhorar: grandes aplicações tem, por consequência do próprio tamanho, uma complicação no momento de se avaliar impactos quando são efetuadas alterações [13, 14]. Portanto, é um momento que se deve levar em consideração se realmente é interessante correr o risco de inserir novas falhas ao tentar aplicar uma melhora que não é necessária [3].

O última meta a ser estabelecida no time é manter as soluções o mais simples possível, métodos grandes e complexos são inimigos da manutencão e da reusabilidade. Acrescentar muita responsabilidade funcional sobre um único trecho de código pode ser um problema para manter o programa adaptável a mudanças. Manter soluções simples não é pensar em soluções flexíveis à todos os tipos de ambientes, pois, flexibilidade em excesso também pode ser um problema. A aplicação será tão complexa que poderá cair no mesmo problema inicial de compreensão do sistema [15].

2.4.2.2 Primeiro contato com a aplicação

Após as decisões de metas discutidas na seção anterior, examina-se o estado atual do sistema a ser reestruturado. Um objetivo que não deve ser perdido de vista nessa etapa da engenharia reversa é que o sistema é importante demais para ser eliminado e por isso está sendo investido tempo e dinheiro para recupera-lo. Portanto, o foco dessa etapa não é gastar tempo procurando problemas que o sistema já possui, mas encontrar coisas boas que possam ser reaproveitadas e permitam ter um conhecimento melhor de como o sistema se tornou relevante [3].

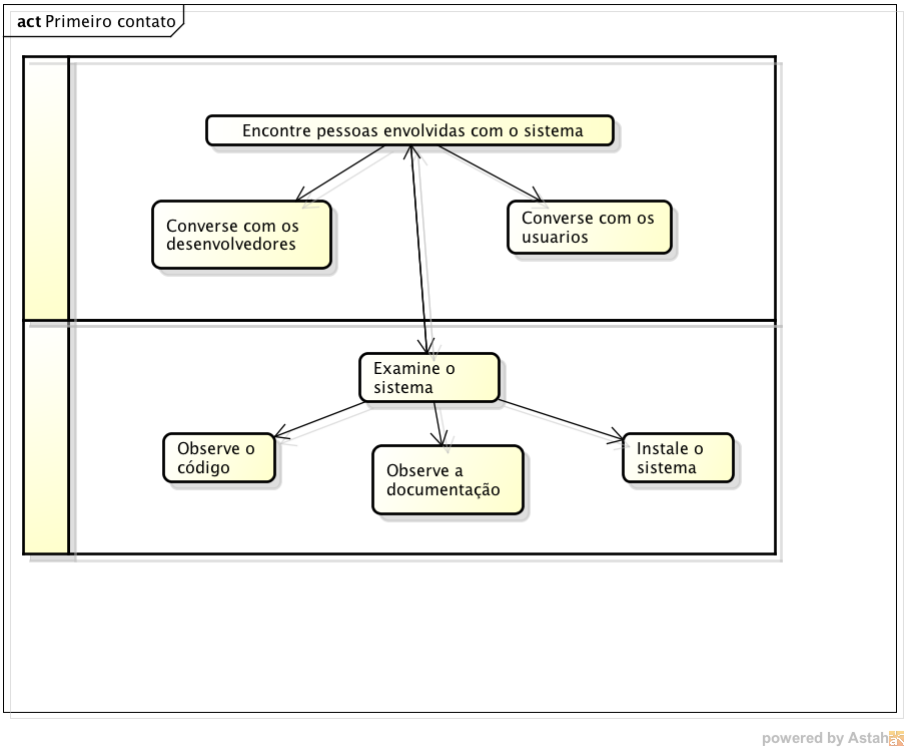

Figura 7 – Um fluxograma para o primeiro contato com a aplicação retirado de Demeyer, Ducasse e Nierstrasz[3].

O primeiro passo no contato inicial com o sistema é conversar com a equipe de manutenção do sistema. O problema de se conversar com a equipe é que as pessoas que causaram os problemas no sistema podem ainda estar presente no time e fornecer informações que não são confiáveis, porém, pode ser também a oportunidade de descobrir idéias interessantes do que causou o sistema a se tornar instável da maneira como funciona hoje e possivelmente descobrir como o sistema deveria ter sido construído originalmente para que o cenário atual fosse evitado. Ao conversar com a equipe, algumas informações que devem ser extraídas na conversa são [3]:

- Quais foram as correções mais complicadas e mais simples efetuadas nos últimos dias;
- Qual foi a forma de correção para os erros encontrados nos últimos dias;
- Qual a parte da documentação é mais confiável, se existente;
- Qual a frequência de rotatividade do time responsável pelo sistema e como isso afetou o desenvolvimento?

Uma segunda conversa deve ser feita também com os usuários do sistema, é uma oportunidade muito boa para validar a interface do sistema e os requisitos do sistema, já

que uma documentação sobre isso pode estar ausente. É interessante entrevistar diferentes tipos de usuários do sistema e pedir para que os mesmos executem os fluxos mais comuns do sistema. Dessa forma, pode-se encontrar os requisitos funcionais do sistema de maior relevância para a aplicação. Como no início dessa seção, o objetivo da engenharia reversa é descobrir pontos positivos do sistema a nível de requisitos e implementações. É comum que durante a entrevista os clientes reclamem sobre as partes mais instáveis do sistema e isso também deve ser considerado desde que não atrapalhe a descoberta das partes relevantes do sistema [3].

Após o contato com as pessoas envolvidas com o sistema, é o momento de observar superficialmente a documentação, código e instalação do sistema. Ao ler pela primeira vez, não é aconselhado que se tente entender cada linha produzida, mas observar a forma como foi escrita. Vícios de programação, identação, padrões de escrita entre outros aspectos de produção código devem ser observados para se ter uma noção do estado de desenvolvimento do sistema. Qualquer vestígio de boa prática de programação deve ser observada para que seja mantida em uma versão futura do sistema [3].

Sobre a documentação, alguns cuidados devem ser tomados. Se existente, deve ser avaliado se está coerente e atualizada com o sistema que está em produção nos clientes. Caso desatualizada, baseada na conversa feita com desenvolvedores e clientes já é possível ter uma noção de qual parte da documentação é mais relevante para ser atualizada, ou criada caso inexistente. As documentações que não forem relevantes de acordo com desenvolvedores e clientes não deve ser levada em consideração nesse momento [3].

Como último passo dessa etapa, uma instalação do sistema é recomendada. Se o sistema é de grande porte, compilar ou instalar a aplicação pode não ser um processo simples. Aprender como instalar o sistema é uma chance de aprender quais são os arquivos essenciais e como o mesmo foi estruturado fisicamente, quais são as dependências externas e internas da aplicação a nível de bibliotecas.

2.4.2.3 Compreensão inicial da estrutura do sistema

Nesse passo da engenharia reversa, é proposto o início da recuperação de modelos do sistema. É o momento de confirmar se todas as primeiras impressões feitas em relação ao sistema estão corretas. Para verificar isso, podemos analisar o sistema de forma *top-down* onde serão feitas especulações sobre o sistema através de um diagrama de classes e posteriormente verificadas com o código-fonte do sistema ou *bottom-up* onde a partir dos dados persistidos no banco será criado um driagrama de classes. Independente de qual estratégia escolhida é aconselhado que em qualquer processo de engenharia reversa seja usado pelo menos um pouco de ambas as estratégias.

A estratégia *bottom-up* se baseia em uma análise rigorosa do esquema usado no banco de dados. Para aplicar a sugestão que é mostrada a seguir é necessário que o banco

usado no sistema seja o banco de dados relacional que é o caso para a maioria dos sistemas orientados a objetos.

1. Enumere todas as tabelas e para cada uma, crie uma classe com o mesmo nome;
2. Para cada tabela, acrescente o nome das colunas como atributos da respectiva classe;
3. Para cada tabela determine as chaves candidatas caso isso não esteja totalmente explicito;
4. Observe as chaves entre classes que podem representar heranças e as modelem de acordo;

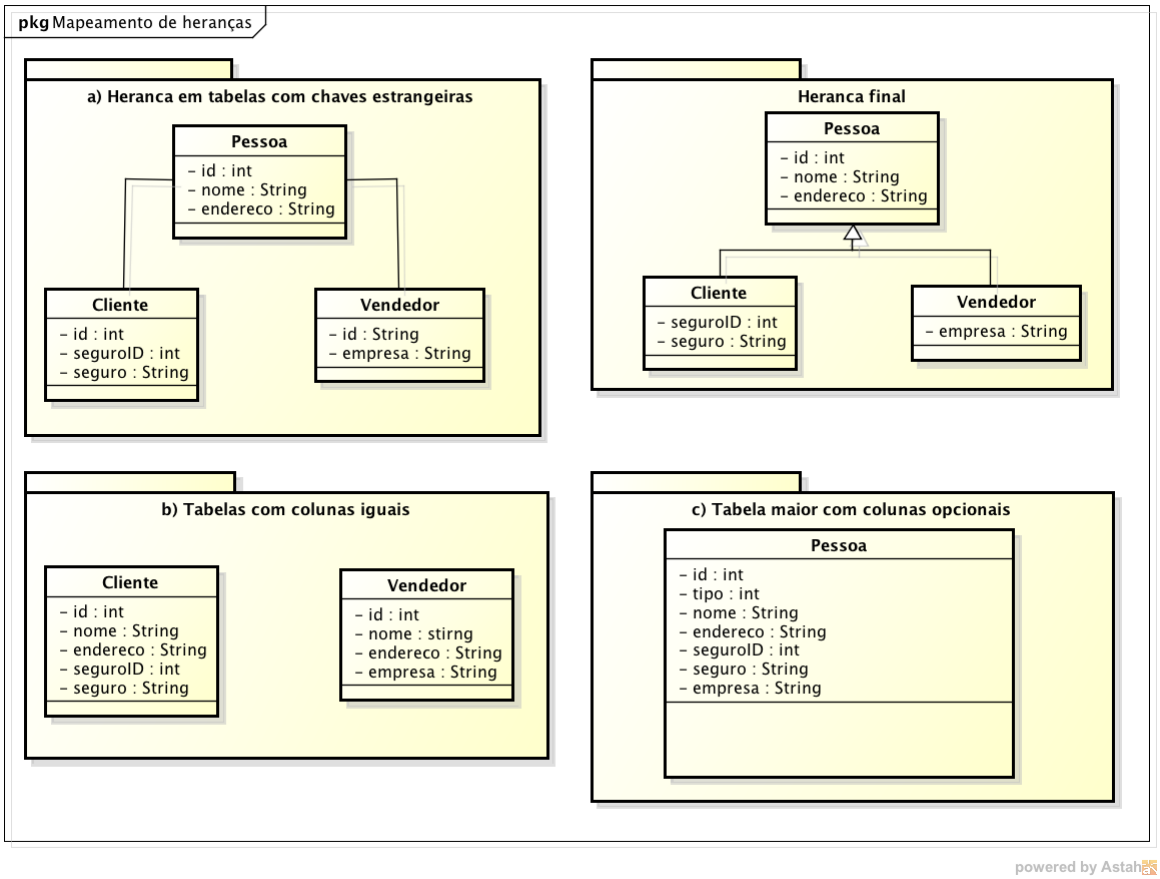

Figura 8 – Mapeamento para heranças adaptado de Demeyer, Ducasse e Nierstrasz[3].

a) Conferir as tabelas onde a chave primária também serve como uma chave estrangeira para outra tabela, como tal, as chaves estrangeiras podem representar relações de herança. Examine as instruções *SELECT* que são executados contra essas tabelas para ver se eles geralmente envolvem uma junção sobre esta chave estrangeira. Se este for o caso, analisar os nomes de tabela e o código-fonte correspondente para verificar se essa chave estrangeira na verdade representa

um relacionamento de herança. Se isso acontecer, transformar a associação que corresponde à chave estrangeira em um relacionamento de herança.

b) Conferir as tabelas com conjuntos comuns de definições de coluna, como estes, provavelmente, indicam uma situação em que a hierarquia de classes está espalhada por várias tabelas, cada tabela representa uma classe não-abstrata. Definir uma superclasse comum para cada *cluster* de definições de coluna duplicados e mover os atributos correspondentes dentro da nova classe. Verifique o código fonte para o nome aplicável para as classes recém-criadas.

c) Conferir as tabelas com muitas colunas e um monte de atributos opcionais que estes podem indicar uma situação em que uma hierarquia de classes completo é representado em uma única tabela. Se você encontrou tal tabela, examine todas as instruções SELECT que são executados contra esta tabela. Se estas instruções SELECT solicitar explicitamente para subconjuntos das colunas, então você pode quebrar essa uma classe em várias classes, dependendo dos subconjuntos solicitados. Para os nomes dessas classes, verifique se há uma codificação de informações subtipo como por exemplo a coluna tipo segurando um número do tipo de enumeração.

5. Repassar para o diagrama de classes as cardinalidades presentes no esquema do banco:

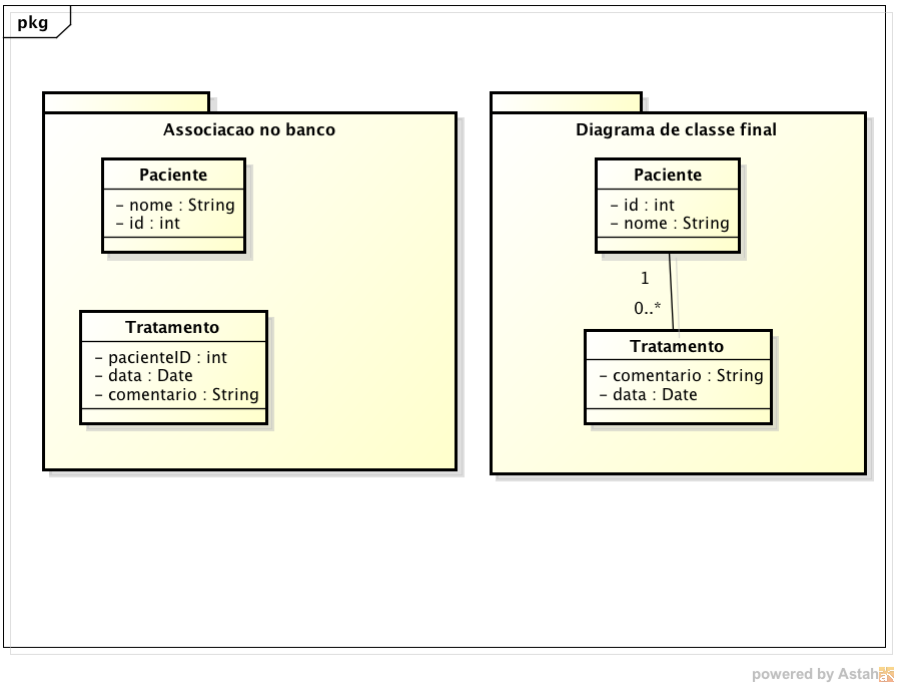

Figura 9 – Mapeamento para cardinalidades.

Fonte: Adaptado de [3].

a) Determinar as associações, ou seja, classes que representam que duas outras tabelas estão associadas. O exemplo mais ocorrente é o muitos-para-muitos, a qual é representada por uma tabela com uma chave candidata que consiste em duas chaves estrangeiras. Em geral, todas as tabelas onde as chaves candidatas são concatenações de múltiplas chaves estrangeiras são casos potenciais de uma classe de associação.

b) Atenção com multiplicidades para as associações. Uma vez que todas as associações são derivadas de relações de chave estrangeira, todas as associações são de construção opcionais *1-to-many associations*. No entanto, ao inspecionar as declarações não nulos, os índices e as amostras de dados pode-se muitas vezes determinar as multiplicidades mínimo e máximo para cada um dos papéis na associação.

A estratégia *top-down* se baseia em uma abstração por conta da equipe para criar um diagrama de classes que supostamente seria um espelho do código fonte do sistema. Pode vir a ser uma estratégia mais complicada de ser executada, pois, para uma mesma solução é possível aplicar diversas formas de solução em diversas linguagens. Outra dificuldade que pode ser encontrada são os objetos que são meramente responsáveis por configurações do sistema ou conexões com banco de dados [3].

Para a abstração é proposto os seguintes passos [3]:

1. Com o conhecimento atual sobre o sistema, modele um diagrama do que supostamente acreditaria-se ser o sistema atual. Nomes de atributos e padrões de escrita dos métodos pode ser uma tarefa fácil, caso o código do sistema possua um padrão de escrita.

2. Enumere todos os métodos e atributos que foram escritos no diagrama de classes. Procure cada um deles no código fonte do sistema com qualquer ferramenta que for disponível para uso.

3. Mantenha uma forma de documentar os métodos e atributos que foram encontrados corretamente no código-fonte e os que não foram. É importante destacar que os que não foram encontrados no sistema representam a parte mais importante da estratégia *top-down* pois implicaram em aprendizado sobre a implementação do sistema.

4. Efetue correções no diagrama de classes elaborado. Essas correções podem implicar em:

   - Renomear: métodos e atributos que não foram nomeados corretamente na abstração do modelo devem ser corrigidas.

- Remodelar: quando for identificado que um detalhe na modelagem não corresponde com o que acontece na realidade no código fonte do sistema. Isso pode implicar, por exemplo, em transformar atributos em métodos ou métodos em atributos.
- Extender: pode ocorrer de ser encontrados métodos, atributos e até mesmo classes que não foram modeladas mas estão presentes no código do sistema.
- Remodelear: em último caso, pode ocorrer uma enorme quantidade de erros e o diagrama modelo não reflete nada da implementação do sistema. Nesse caso é aconselhado uma nova abstração e remodelar o sistema novamente.

5. Repetição dos passos 2-4 até que o diagrama modelado possua uma representação satisfatória do desenho do sistema.

Alguns conhecimentos prévios que podem ajudar na recuperação do diagrama de classes do sistema são:

- Conhecer se o sistema emprega algum padrão de projeto;
- Conhecer a arquitetura do sistema em nível de componentes e conectores;
- conhecer bem o modelo de negócios do sistema.

O último passo para completar a compreensão inicial do sistema é estudar os casos em que a aplicação apresenta instabilidade para realizar tarefas, alguns exemplos são [3]:

- Casos em que o montante de dados processados por causar muita demora;
- Momentos do dia em que muitos usuários estão usando a aplicação ao mesmo tempo;
- Casos em que o consumo de recursos usado pelo código escrito é muito alto.

2.4.2.4 Captura detalhada dos modelos do sistema

Até então, documentação já foi analisada, observações sobre o código foram feitas, e alguns diagramas já foram recuperados. Nessa etapa, é necessário validar se todo o conhecimento adquirido em relação ao software está correto. Ao mesmo tempo é o momento de ter conhecimento também dos detalhes de implementação do sistema e de alguns fluxos dinâmicos como instancias de objetos em tempo de execução [3].

O primeiro passo para a captura detalhada é manter as documentações atualizadas com o conhecimento presente sobre o código. Manter código e documentação em exato reflexo é um desafio devido ao fato do caráter evolutivo do próprio software. Mas, se a documentação não for mantida, todo o trabalho feito de compreensão até aqui pode ser necessário novamente um dia.

Algumas práticas que podem ser executadas em qualquer ordem para confirmar e detalhar todo o conhecimento até então são:

- Refatorar partes do sistema: antes de refatorar já se possui uma idéia de como o código funciona baseado em diagramas e discussões feitas nos passos anteriores. Porém, o código pode não refletir exatamente o que se pensa ou refletir parcialmente o conhecimento adquirido até o momento. A melhor forma de analisar e descobrir os detalhes importantes da aplicação é refatorar pequenas partes do código. Refatorar pode causar erros em outras partes do sistema caso testes de regressão não sejam executadas corretamente, porém isso não deixa de ser algo bom, pois mostrará os lugares onde a parte de código alterado influencia [16, 17].

- Usar ferramentas de *debug* para analisar detalhadamente o código do sistema: uma segunda forma de análise dos detalhes do sistema. Usar ferramentas que mostrem passo a passo os detalhes de como os objetos se comportam na implementação é importante para que se compreenda a importância de cada atributos que ainda não se compreendia bem o porque da existência e a confirmação de impressões sobre o código.

- Escrever testes para ver como o Sistema se comporta como uma forma de confirmação de sucesso da refatoração e dos casos de *debug* propostos acima;

- Aprender com as mudanças passadas: é uma forma de descobrir que partes da arquiteturas estão instáveis. Deve-se observar que partes do sistema tem mudado com mais frequência e que partes não apresentam necessidades de mudanças. As partes que mudam com mais frequência identificam locais do sistema em que a arquitetura não tem comportado bem as alterações necessárias na evolução do software. É importante descobrir o motivo que levou essas partes instáveis do sistema ao cenário atual para que erros não se repitam.

2.5 Práticas de reestruturação de código

Nessa sessão, é apresentada os principais conceitos relacionados à refatoração de código.

2.5.1 Refatoração de código

Refatorar é a arte de melhorar o projeto de um código existente [14]. Outra definição a ser levada em consideração diz que refatorar é o processo de alteração de um software sem que o comportamento externo do sistema não seja alterado, porém, melhore a sua estrutura interna [7].

Ambas definições levam as seguintes conclusões [14]:

- Refatoração não inclui apenas as alterações em um sistema: Apesar de refatoração ser uma boa prática do processo usado para criar um novo código, não é a parte responsável por adicionar novos recursos;
- Refatoração não é reescrever a partir do zero: Embora existam momentos em que é melhor para começar do zero, refatorar é uma tentativa de salvar o código, fazendo o possível para melhorar o código, em vez de correr o risco de reescrevê-lo;
- Refatoração não é qualquer reestruturação com a intenção de melhorar o código: o foco de uma refatoração é ser uma transformação segura. Mesmo grandes refatorações que mudam grandes quantidades de código são divididas em menores, refatorações seguras. (No melhor dos casos, refatoramentos são tão bem definidos que passam a ser automatizadas.) Não consideraremos uma alteração como refatoração se deixa o código sem funcionar (isto é, não passa os seus testes) por mais de uma sessão de trabalho.
- Refatoração colabora no projeto de funcionalidades novas: A boa prática de refatoração implica em um suporte e extensão de funcionalidades pois torna o código mais legível e manutenível;
- Refatorações podem ser efetuadas em pequenas ou grande quantidades: muitas refatorações são pequenas. Idealmente, as pequenas refatorações são aplicadas o suficiente para que grandes refatorações raramente são necessários.

2.5.2 Ciclo básico de refatoração de código

O ciclo de vida de um processo de refatoração de código é apresentado na figura 2.5.2.

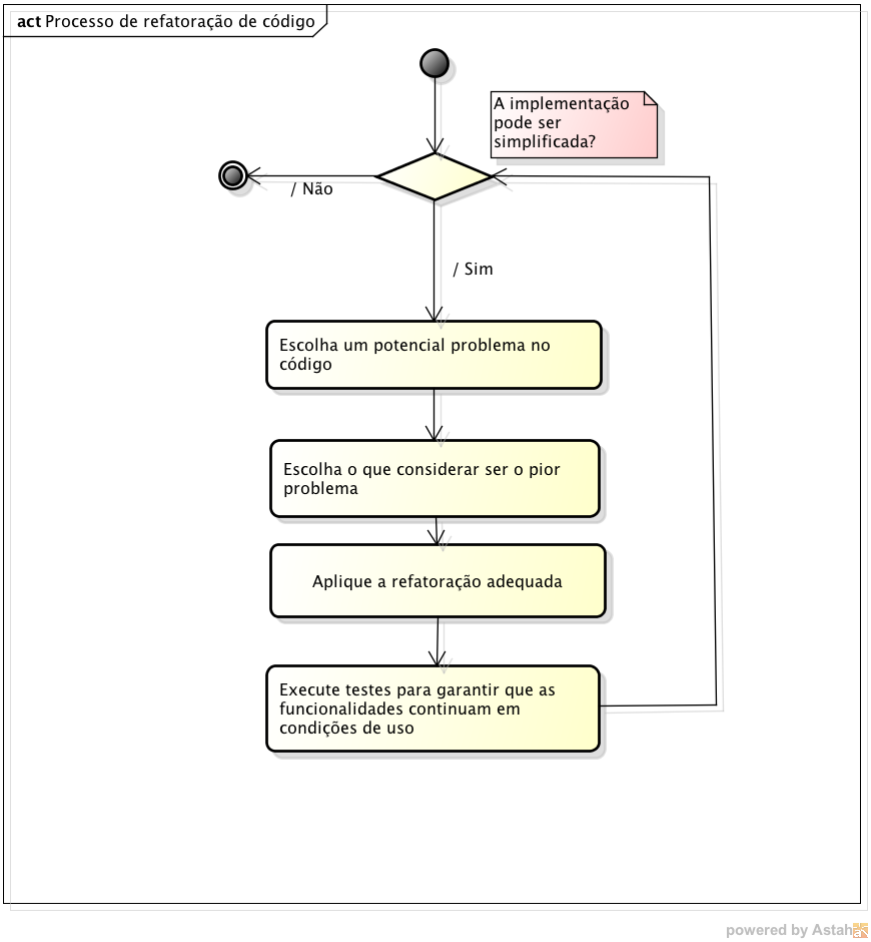

Figura 10 – Processo de refatoração de código adaptado de Wake e Rutherford[14].

A idéia principal é selecionar refatorações que melhoram o código em cada iteração. Devido a nenhuma das etapas mudar o comportamento observável do programa, o mesmo permanece em um estado de funcionamento. Assim, o código é alterado, mas retém o mesmo comportamento externo.

O momento de parada do ciclo pode ser respondido através de quatro diretrizes [15]:

- Passa em todos os testes;
- A leitura do código comunica de forma eficiente o que código realmente faz;
- Não existe código duplicado;
- Não existe código desnecessário;

2.5.3 Potenciais problemas em código

Um potencial problema representa uma má prática de programação e como o nome mesmo já informa, representa um perigo de se tornar um problema. Normalmente, um problema está associado à uma má prática.

Os principais problemas são [14, 7]:

- Comentários;
- Métodos Longos;
- Código Duplicado;
- Classes Longas;
- Longa lista de parâmetros.
- Nomes problemáticos;

2.5.3.1 Comentários

Comentários quando não inseridos de forma automática por uma IDE, representam uma má prática de programação. A existência de comentário na implementação pode representar diversos problemas. Comentário explica o que a implementação faz. A implementação em si já deveria representar uma forma de comunicação, se é necessário um comentário para transmitir o conhecimento sobre o código, então o programador que a implementou falhou. Se o comentário não for mantido junto com a alteração de código, o comentário passa a ser uma ameaça ainda maior, pois informa de forma incorreta. Nesse caso, o mais indicado é que um novo método seja inserido nas partes em que o comentário é necessário, o novo método deve ter um nome bastante comunicativo para que o comentário possa ser excluído [14, 7, 18].

2.5.3.2 Métodos Longos

Métodos que ocupam uma grande quantidade de linhas podem representar problemas de manutenção e extensão. Qualquer método que possua mais que 10 linhas pode apresentar o problema de responsabilidades. Seria uma boa prática que um método estivesse associado a somente uma funcionalidade. Um método deve executar somente uma tarefa e executa-lá muito bem. Quando essa prática é quebrada os seguintes problemas podem ser facilmente percebidos:

- Se torna complicado escrever casos de testes para garantir que as funcionalidades desse método estejam de acordo com o que deveriam.

- Como mais de uma funcionalidade está associada a esse método, esse trecho de código fica sujeito a mudanças com mais facilidade, pois, qualquer uma das responsabilidades que esse método carrega significa uma potencial necessidade de mudança.

A solução nesses casos, é quebrar o método em outros métodos menores de forma que cada método menor carregue uma e somente uma funcionalidade [14, 7, 18].

2.5.3.3 Código duplicado

Esse representa um enorme problema e, possivelmente, um dos mais perigosos. Sempre que uma funcionalidade similar ou um trecho de código é requisitada em outra parte do programa o costume mais usual em meios aos programadores é efetuar uma cópia e cola do trecho necessário. Esse é o maior problema de todos. Algumas consequências dessa má prática são:

- O número de linhas do programa aumenta significantemente sem necessidade;
- Se o trecho de código original precisa passar por mudanças, todos as ocorrências de cópias desse trecho devem ser imediatamente atualizadas ou o sistema quebra [14, 7, 18].

2.5.3.4 Classes Longas

Classes longas de forma muito similar aos métodos representam um problema quando uma classe está responsável por tratar mais funcionalidades do que o seu próprio nome nos indica. Uma classe deve ter somente uma razão associada para a ocorrência de alterações. Uma classe que com muita responsabilidade se torna muito importante para o sistema, e portanto dificulta a substituição quando necessária, torna a possibilidade de herança ainda mais perigosa por carregar uma grande quantidade de funcionalidades. Assim como os métodos a solução direta para o problema é a quebra da classes em várias outras menores [14, 7, 18].

2.5.3.5 Longa lista de parâmetros

Longa lista de parâmetros é algo que normalmente está associado aos métodos longos. Uma grande quantidade de parâmetros representa um perigo, pois, obriga que o programador se lembre de muita coisa a ser tratada para que o método seja usado. Se o método precisa tratar muitos parâmetros, então, é provável que o método esteja responsável por mais de uma funcionalidade. Por fim, quando a lista de parâmetros muda, uma lista de tratamento é imediatamente requisitada a mudar também. Nesse caso, a solução do problema pode ser quebrar o método em mais de um ou usar uma estrutura *hash* como parâmetros para facilitar a comunicação [14].

2.5.3.6 Nomes problemáticos

Nomes de métodos ou atributos de uma letra só é um exemplo de nome pouco comunicativo. Esse tipo de nome jamais deve ser atribuído em uma implementação, pois força o programador responsável pela manutenção a olhar todos os usos desses métodos ou atributos no fonte do sistema para que uma breve noção sobre o assunto seja esclarecida. Essa busca no sistema implica em perda de tempo e, claro, dinheiro. A renomeação é a solução direta e deve satisfazer os seguintes requisitos [14, 7, 18]:

- Informar ao leitor o que aquele atributo ou método faz;
- Se possível, não ser uma abreviação;
- Em hipótese alguma não permitir ambiguidade na compreensão do nome.

2.6 Migração de software

Esta seção aborda soluções para um problema que a maioria das equipes que estão passando por um processo de reengenharia encontra: a migração do sistema. O objetivo de toda migração de software é manter dados e funcionalidades em condições de uso pelos clientes [19, 20]. Para que esse objetivo seja cumprido apresenta-se duas soluções: a estratégia *Chicken Little* e a estratégia *Butterfly*.

2.6.1 A estratégia Chicken Little

Tem como principal característica a migração incremental. Neste método, os sistemas de informação legado e destino são operados em paralelo ao longo da migração. Inicialmente, o sistema destino é muito pequeno, talvez apenas uma aplicação com um pequeno banco de dados. No entanto, a medida que a migração progride, o sistema destino irá crescer até que ele execute todas as funcionalidades do sistema legado que poderá ser aposentado. Durante a migração, o sistema legado e o sistema destino precisam cooperar para formar o sistema em produção. Essa cooperação é fornecida através de módulos chamados *gateways*: um módulo introduzido entre dois componentes para intermediar entre ambos [20, 21].

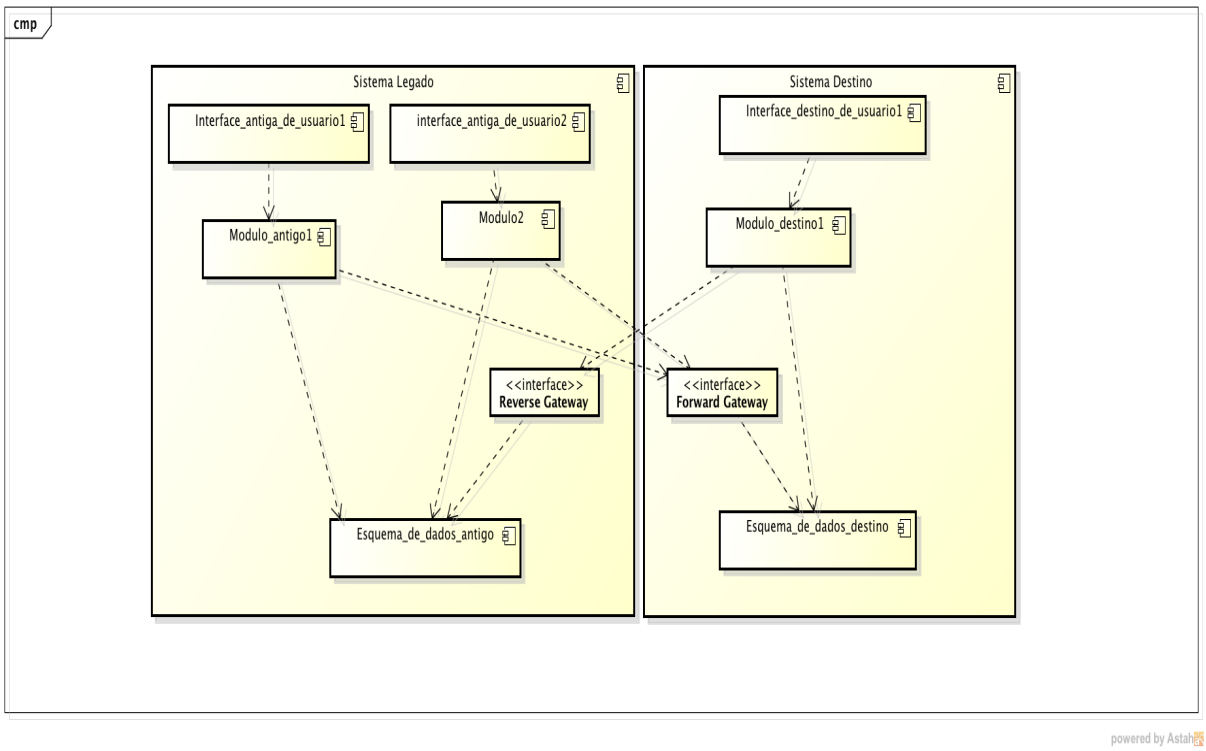

Figura 11 – Exemplo de arquitetura da migração *Chiken Little* adaptado de Brodie e Stonebraker[20].

Em todo o processo de migração, os dados são armazenados em ambos os sistemas. Um *forward gateway* é utilizado para permitir que as partes legadas possam acessar o ambiente de banco de dados no lado do sistema destino do processo de migração e um *reverse gateway* é utilizado para permitir que as partes do sistema destino acessem o ambiente de gerenciamento de dados do sistema legado [22].

O sistema legado deve ser interpretado como um conjunto de módulos que serão reconstruídos separadamente. A chave desse método incremental é, então, o trabalho conjunto de ambos os *gateways* para que módulo a módulo o sistema seja parcialmente migrado para o seu destino. O fato da migração ser divididas em etapas parciais é o que contribui para que essa solução seja interessante [20].

O sucesso dessa estratégia de migração se baseia em iniciar processo com partes do sistema que se tenha certeza que são independentes e gradualmente validar que cada passo de migração tenha sido efetuado com sucesso. Dessa forma cria-se uma possibilidade de segurança durante o processo de migração. Se um passo em uma fase avançada do processo falhar, é possível recontinuar do passo anterior. Essa seria a maior vantagem da migração incremental [20].

A migração executada em partes faz da migração como um todo um processo mais lento, porém o ganho dessa estratégia está na segurança proporcionada pelos pequenos passos. Uma vez que os passos são executados em intervalos de tempo definidos, existe a chance de checar erros durante a migração e com segurança organizar o próximo passo [20].

Os passos propostos pela estratégia *chicken-little* são apresentados a partir da sessão 2.6.1.1 ate a sessão 2.6.1.10.

2.6.1.1 Analisar o sistema legado

Compreender o sistema a ser migrado em detalhes. Isso inclui o sistema alvo também, levantamento de requisitos para ambos os sistemas que serão envolvidos é necessário. Deve-se organizar os requisitos do sistema atual e escrever caso necessários os novos requisitos para o sistem alvo [20].

2.6.1.2 Decompor a estrutura do sistema

Modificar o sistema atual de forma que o mesmo seja possível de ser transformado em um conjunto de módulos. É necessário interfaces bem definidas entre cada componente para que a migração seja efetuada com sucesso. A dificuldade dessa etapa depende muito da situação em que o sistema a ser migrado se encontra [22].

2.6.1.3 Projetar a interface do sistema destino

Efetuar a modelagem de sistema alvo. Esse passo inclui que o a modelagem de interfaces com o usuário final caso o sistema alvo necessite de uma nova interface.

2.6.1.4 Projetar a aplicação destino

Projetar a arquitetura do sistema alvo. Esse passo é o responsável por definir a dificuldade de se instalar os *gateways* necessários para o funcionamento paralelo de ambas as aplicações. Essa etapa também define se novas funcionalidades serão ou não acrescentadas durante o processo de migração [20].

2.6.1.5 Projetar o esquema do banco de dados destino

Modelar o esquema que será usado na aplicação alvo. O resultado dessa etapa é um esquema de banco de dados responsável por persistir os dados das partes da aplicação alvo que forem migradas. Ao final do processo somente o esquema gerado nessa etapa será mantido [22].

2.6.1.6 Criar e instalar os *gateways* necessários

Implementar *gateways* necessários para a migração. Esse componente é responsável por encapsular o banco de dados alvo e a aplicação antiga para que os passos da migração possam ser efetuados independentemente. No caso do *forward gateway*, o seu objetivo é permitir que a aplicação legada tenha acesso ao esquema do banco de dados alvo, já no caso do *reverse gateway* a necessidade é que a aplicação alvo seja capaz de acessar dados no esquema de banco de dados antigo [22, 20].

2.6.1.7 Migrar o banco de dados legado

Nessa etapa, todos os dados associados a um determinado módulo do sistema devem ser migrados. Transformações de dados diretas podem ser necessárias dependendo da quantidade de mudanças efetuadas no novo esquema de dados [20]. Ao final desse passo, o módulo associado no esquema de banco de dados antigo deverá estar sem conteúdo.

2.6.1.8 Migrar a aplicação legada

Uma vez que os dados referentes a um módulo já foram migrados, é a vez de migrar as partes da aplicação responsáveis por esses dados. Essa etapa consiste em desativar o uso da aplicação antigo responsável pelos dados migrados [20].

2.6.1.9 Migrar interfaces legadas

Por último, novas interfaces serão migradas caso tenham sido necessárias. as novas interfaces então são ligadas ao cliente para que comece a usar o módulo alvo que, com essa etapa, foi migrado por completo [20].

2.6.1.10 Mudar para o sistema destino

Para concluir uma iteração do passo migratório, essa etapa consiste na desativação ou retirada do código antigo responsável pelo módulo migrado. Isso envolve a remoção da parte correspondente no esquema de dados antigo [20].

2.7 Trabalhos relacionados

Durante o estudo efetuado, um conjunto de trabalhos relacionados foi encontrado. Algumas propostas de trabalho se baseavam puramente no uso de ferramentas para automatizar o trabalho. Outro se trata sobre uma segunda opção de migração de software.

2.7.1 Engenharia reversa baseada em uso de ferramentas automáticas

Essa é uma abordagem diferente das apresentadas nas sessões anteriores. O método proposto com o uso de ferramentas tem como objetivo principal a extração automática de informações de uma aplicação web através de 2 frentes: uma análise dinâmica da aplicação que consistem em observar o comportamento do sistema em funcionamento e uma análise estática do código-fonte onde as ferramentas automáticas geram diagramas [23]. A visão geral dessa proposta de engenharia reversa é mostrada na figura 12.

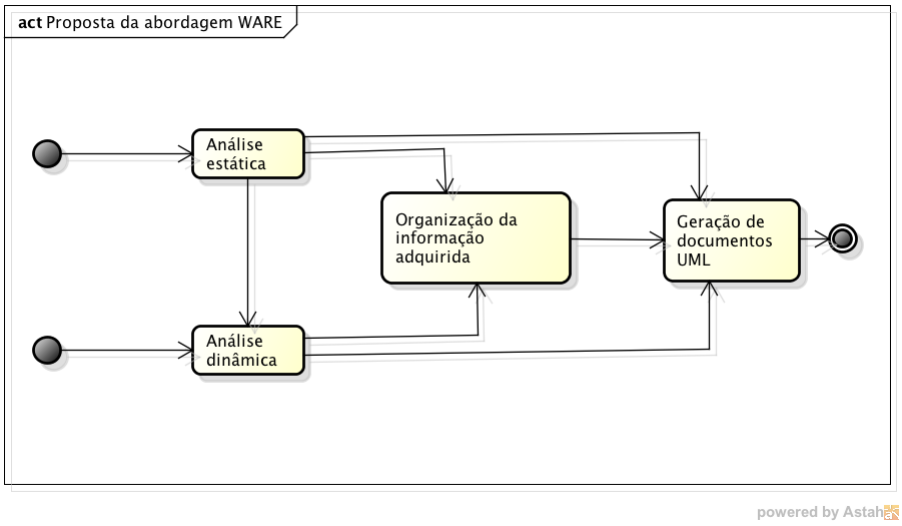

Figura 12 – Proposta da abordagem WARE adaptado de Lucca, Fasolino e Tramontana[23].

2.7.2 Migração de software sem uma abordagem incremental

Esta é uma metodologia em que, diferentemente da *Chicken Little*, o sistema legado permanece em operação, enquanto que, em um ambiente de teste, o novo sistema pode ser desenvolvido e testado previamente a uma migração, sem afetar o funcionamento ou mesmo causar transtornos [21].

Essa forma de migração introduz alguns nomes importantes para a compreensão da mesma:

- *Data-Access-Allocator (DAA).* É a parte responsável por ler os dados do banco correto, e salvar as transformações efetuadas
- *TempStores.* São cada um dos banco de dados congelados no decorrer da migração.
- *Chrysaliser.* É o componente responsável na migração pela transformação dos dados salvos em cada um dos bancos de dados legadas para o bando de dados alvo.

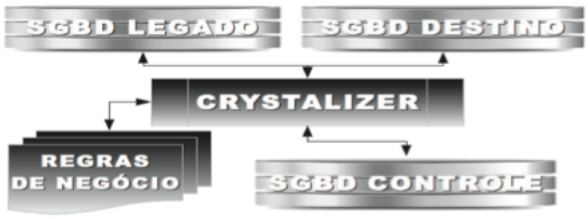

Figura 13 – Exemplo de arquitetura da migração *Butterfly* adaptado de Oliveira e Marcelino[21].

A abordagem baseia-se no pressuposto de que os dados do sistema legado constituem a parte mais importante do sistema. Assim, a metodologia de separa o desenvolvimento do sistema alvo em fases migratórias de dados, eliminando assim a necessidade de *gateways* [24].

Uma vez que a migração de dados começa, o armazenamento de dados legado é congelado para ser usado somente em modo leitura. Manipulações de dados legados são redirecionados pelo *Data-Access-Allocator (DAA).* Quando os dados de acesso de aplicações legadas,o DAA recupera os dados da fonte correta, ou seja, os dados legados ou o *TempStore* correto [25].

Um transformador de dados, *Chrysaliser*, é empregado para migrar os dados do sistema legado, assim como os dados das TempStores o sistema alvo. Quando *Chrysaliser* está migrando os dados legados, todas as manipulações são armazenados no *TempStore1*, ao migrar os dados em *TempStore1*, todas as manipulações são armazenados em *TempStore2*, e assim por diante. Dessa forma, as *TempStores* serão cada vez menores [25].

3 PROPOSTA DE UM PROCESSO DE REENGENHARIA ORIENTADO A OBJETOS INCREMENTAL - PROO-I

A necessidade da elaboração da presente proposta surgiu visto que a reengenharia de um sistema corporativo de grande porte precisa inserir as melhorias de forma incremental pois caso contrário o risco e o investimento inicial de reconstruir um novo sistema integralmente seria muito maior e inviabilizaria o processo. A manutenção e extensão do sistema remete grandes investimentos pois uma mudança implica em impactos em várias partes do sistema (alto acoplamento) e não há formas de garantir a consistência global. O processo de validação e verificação não é bem estruturado e nem replicável após a manutenção.

No entanto, essa proposta atende sistemas orientados a objetos com baixa aderência as boas práticas do paradigma que proporcionam o reuso e facilita a manutenção, portanto, elaborou-se um processo de reengenharia de sistemas orientados a objetos com uma abordagem de migração incremental da aplicação. A elaboração da proposta foi baseada nos estudos efetuados sobre as sessões anteriormente apresentadas. A engenharia reversa utilizada na proposta desse trabalho se baseia nos processos apresentados nas sessões 2.4.1 (Método Fusion-RE/I) e 2.4.2 (Método Famoos), de forma a buscar um equilíbrio entre ambas as propostas de engenharia reversa. A reestruturação do sistema se baseia na sessão 2.5.1 (Práticas de refatoração de código). A engenharia avante é baseada na sessão 2.6.1 (A estratégia Chicken Little), o processo híbrido formado com os itens há pouco citados denominou-se PROO-I (Processo de Reengenharia Orientado a Objetos Incremental).

O Fusion-RE/I apresentou uma engenharia reversa com um processo de modelagem bem definida, mas ao tentar aplica-lo em um ambiente real foi percebido que o método Fusion-RE/I têm como resultado um grande volume de documentação no processo de engenharia reversa e este não teve mais serventia após a engenharia avante. Por outro lado, o método proposto pelo projeto FAMOOS elabora uma forma de engenharia reversa muito focada na reestruturação e depuração do código, mas deixa os documentos gerados por conta do engenheiro de software que efetuou o processo, ou seja, a quantidade de documentos é minimizada, visto que este não será parte integrante do documento final do projeto. Assim, se fez necessário a criação de um método que agregasse as melhores práticas dos itens relacionados.

Na sessão 3.1 (Etapas do processo de reengenharia) é apresentado com detalhes cada um dos passos propostos na figura 14; Na sessão 3.2 (Estudo de caso do processo

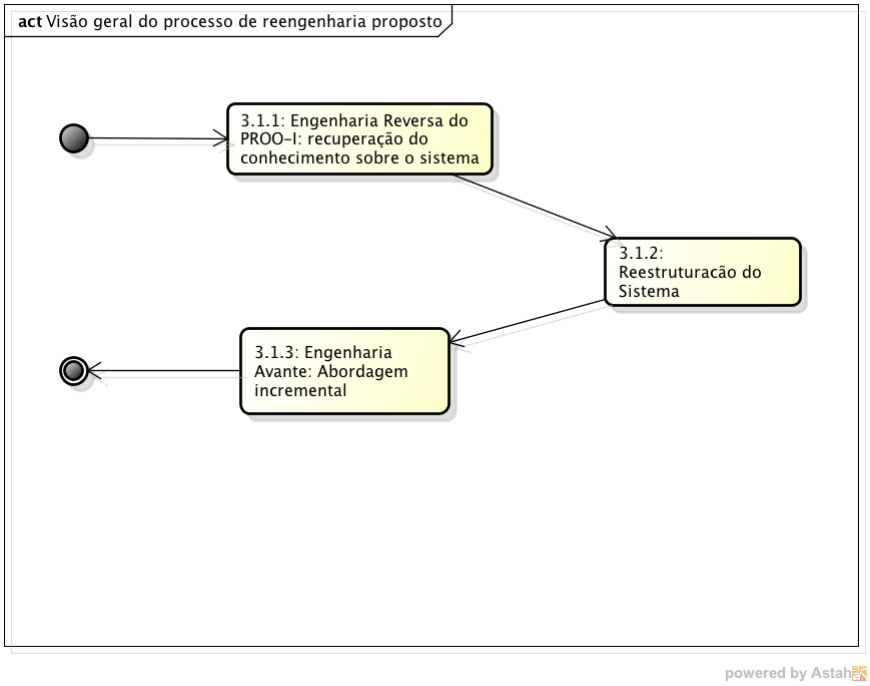

Figura 14 – Visão geral do processo de reengenharia proposto

proposto em um sistema corporativo) apresenta-se o estudo de caso do processo proposto em um módulo de um sistema corporativo; Na sessão 3.3 (Análise de resultados), por fim, apresta-se uma análise dos resultados obtidos com aplicação do processo elaborado.

3.1 Etapas do processo de reengenharia PROO-I

3.1.1 Engenharia reversa do PROO-I

A etapa da engenharia reversa é subdividida em 2 fases: recuperação de requisitos funcionais do sistema a partir da interface da aplicação para a compreensão geral de como o software é utilizado. Posteriormente, um estudo detalhado sobre a estrutura interna do sistema é executa de forma a garantir uma ligação entre os requisitos funcionais com os detalhes de projeto de implementação do sistema abordado.

3.1.1.1 Recuperação dos requisitos funcionais do sistema

A idéia de iniciar um processo de engenharia reversa através da interface do usuário para a obtenção dos requisitos funcionais do sistema é baseada no Fusion-RE/I apresentado na sessão 2.4.1 (Método Fusion-RE/I).

Recuperação do modelo ciclo de vida

Descrição: Como apresentado anteriormente na sessão 2.4.1 (Método Fusion-

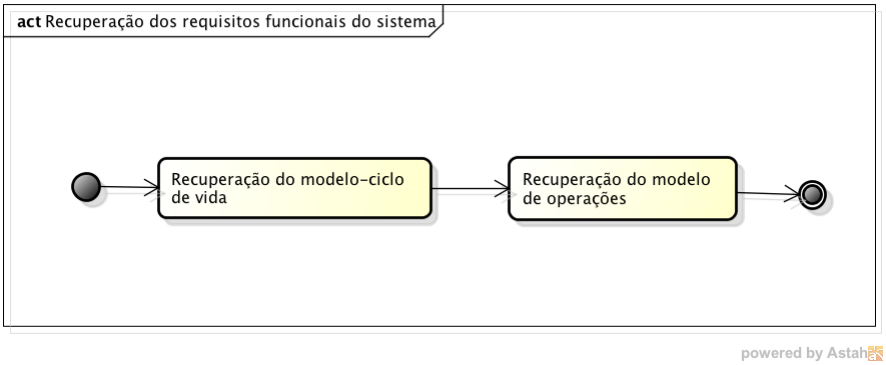

Figura 15 – Recuperação dos requisitos funcionais do sistema

RE/I), o modelo de ciclo de vida tem como objetivo representar a forma como o sitema se comporta com o ambiente e os eventos que o ocorrem durante o seu uso. Aborda o conjunto de operações que o sistema pode efetuar, as entradas que podem ser recebidas e os estados que a aplicação assume para cada uma das entradas ou operações. Descreve de forma genérica o uso do sistema sem detalhamento profundo. Pode ser traduzido como um diagrama de estados [10]. Porém, outro diagrama comportamental que pode ser aplicado para breve descrição do uso do sistema é o caso de uso.

Motivação: A intencão nessa fase do processo é que o time responsável pelo procesos de reengeharia de software tenha um contato inicial com a aplicação que será trabalhada. Portanto, as informações retiradas do modelo ciclo de vida são importante ao ajudarem os desenvolvedores a terem uma noção superficial sobre todas as operações que o sistema pode efetuar. Aceitar diagrama de estados, casos de uso ou até mesmo uma simples lista de requisitos é resultado de cada empresa possuir uma forma diferente de documentar requisitos funcionais.

Recuperação do modelo de operações do sistema

Descrição: Assim, como o modelo ciclo de vida, o modelo de operações também foi apresentado na sessão 2.4.1 (Método Fusion-RE/I) e tem como objetivo descrever as formas de comportamento de um sistema de forma mais declarada a partir de um fluxo escolhido. Parte-se do diagrama de estados obtido para o modelo ciclo de vida que mostra os possíveis usos do sistema para, posteriormente, a escolha de um fluxo do sistema para ser detalhado. Em UML a tradução direta também é um digrama da de estados, mas assim como no item anterior, um diagrama de caso de uso é proposto.

Motivação: Uma vez que o modelo ciclo de vida já revelou as operações mais genéricas do sistema, o modelo de operações irá resultar em um conhecimento mais aprofundado sobre cada uma das operações. Ao combinar o modelo ciclo de vida com o modelo de operações o levantamento de requisitos do sistema está completo.

3.1.1.2 Recuperação da estrutura interna do sistema

Nessa etapa do processo propoe-se as sub-etapas da figura 16, para que o conhecimento adquirido sobre o sistema até então seja confirmado através de um detalhamento do código fonte do sistema.

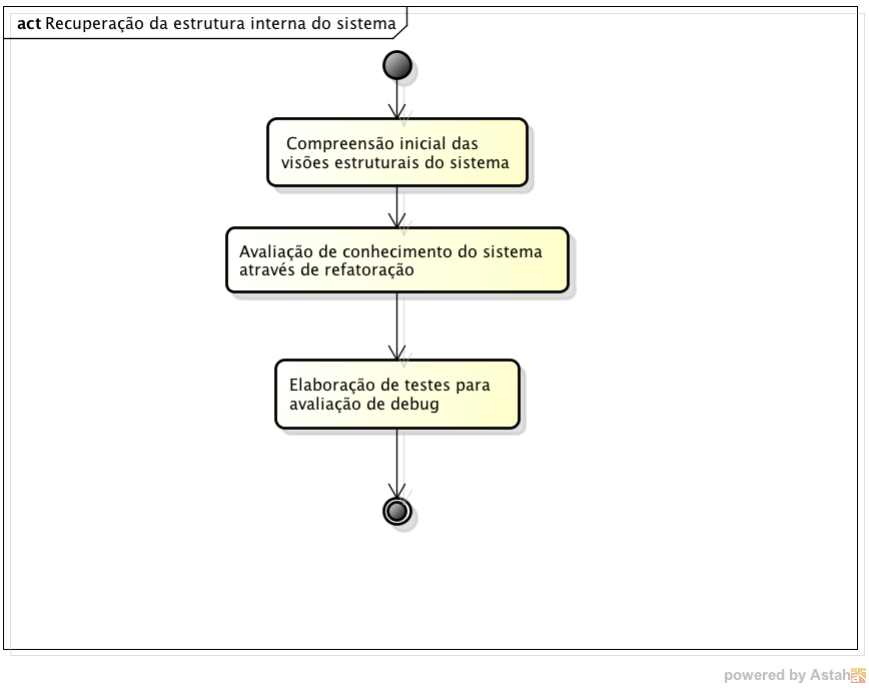

Figura 16 – Recuperação da estrutura interna do sistema

Compreensão inicial das visões estruturais do sistema

Descrição: Esse passo se baseia na criação de um diagrama de classes que represente como os dados são persistidos pelo sistema. O detalhamento dessa atividade segue analogamente ao descrito na sessão 2.4.2.3 (Método Famoos).

Motivação: Para verificar se o conhecimento obtido sobre o sistema nas documentações criadas ou atualizadas no passo da recuperação dos requisitos funcionais, o primeiro passo é levantar um documento sobre o banco de dados da aplicação. Um diagrama de classes, então, revelará como os dados são persistidos e transformados pelo software.

Avaliação de conhecimento através de refatoração

Descrição: Esse passo do processo se baseia em um item descrito na sessão 2.4.2.4 (Método Famoos), onde o indicado a refatoração para a verificar qualquer conhecimento do software até o momento.

Motivação: Apesar de documentação de software ser uma parte muito importante de qualquer processo de reengenharia, ela pode não refletir o conhecimento completamente

verdadeiro do sistema ou, se refletir, não refletir com completude. Uma boa forma de consolidar o aprendizado sobre o sistema é escrever código e certificar-se que o código escrito não cause erros na aplicação.

Elaboração de testes para avaliação de *debug*

Descrição: Também retirado da sessão 2.4.2.4 (Método Famoos), esse passo tem como objetivo análise de detalhes de implementação através de testes e *debugs*.

Motivação: Alguns detalhes muito pequenos passam despercebidos na documentação. São detalhes de implementação em uma determinada linha do código-fonte do programa ou variáveis que cuja a existência ainda não era clara. Casos de teste bem elaboradas são uma escolha interessante para o análise de comportamento do sistema.

3.1.2 Reestruturação do sistema

Descrição: Nesse momento, o levantamento de requisitos do sistema foi completado e um conhecimento suficiente sobre as visões estruturais do sistema também já foi levantada. Porém, para que seja possível reaproveitar as partes boas do sistema para que o novo sistema alvo possa ainda utiliza-las é necessário refatorar. Os itens a serem analisados juntamente com a solução para cada um deles segue analogamente como descrito na sessão 2.5.1 (Práticas de refatoração de código).

Motivação: Os itens a serem corrigidos nesse momento são ameaças que podem ou não causar problemas e aumentar o custo de manutenção do sistema alvo. Portanto, corrigi-las é importante, visto que o principal foco de um processo de reengenharia é aproveitar as partes boas de um sistema para a construção de um novo, onde o custo de manutenção seja menor.

3.1.3 Engenharia avante: abordagem incremental

A principal melhoria apresentada por essa proposta de reengenharia é a transformação do sistema a ser trabalho em um sistema modular que possa ser migrado em partes e, portanto, incrementalmente. A idéia de efetuar uma abordagem incremental é baseada nos conceitos apresentados na sessão 2.6.1 (A estratégia Chicken Little).

Os passos a serem tratados nessa proposta são:

- Decompor a estrutura do sistema;
- Projetar a aplicação destino;
- Projetar o esquema do banco de dados do sistema destino;
- Criar e instalar os *gateways* necessários;

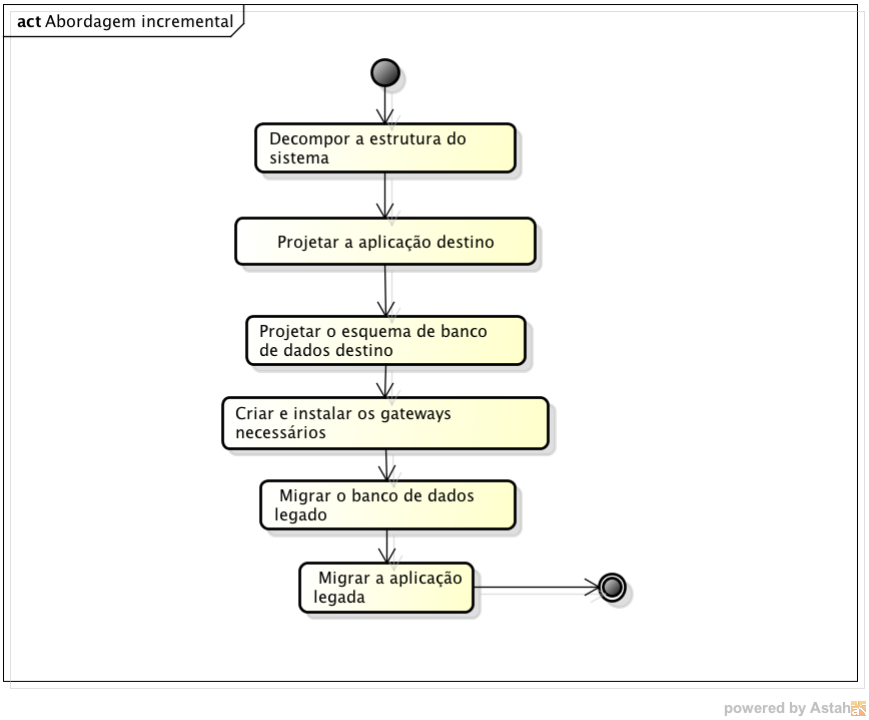

Figura 17 – Visão geral da abordagem incremental.

- Migrar o banco de dados legado;
- Migrar a aplicação legada.

Embora a figura 17 a escrita da proposta apresentem os itens de forma sequencial, não existe necessidade de uma execução dos passos de forma linear. Por exemplo, pode ser que projetar o esquema do banco de dados destino seja necessário antes de projetar a aplicação destino. Essas decisões de projeto podem variar de acordo com a situação do problema. Para módulo do sistema a ser migrada os passos da figura 17 serão efetuados.

3.1.3.1 Decompor a estrutura do sistema

Descrição: Essa atividade consiste em modificar os requisitos do sistema atual para que ele seja possível de ser separado em módulos bem definidos. Dependências entre os componentes internos da aplicação devem ser analisados de forma a tornar a migração mais segura [20].

Motivação: Essa análise é importante para a organização dos novos requisitos funcionais e não funcionais do sistema alvo.

3.1.3.2 Projetar a aplicação destino

Descrição: Consiste em projetar a arquitetura do sistema alvo. Deve-se, portanto, modelar o sistema-alvo de forma que o novo sistema seja capaz de apresentar menores custos de manutenção e extensão. Essa etapa inclui a definição dos novos módulos da aplicação e suas respectivas dependências [20].

Motivação: Essa atividade pode resultar em diferentes opções de implementação e custos de desenvolvimento.

3.1.3.3 Projetar o esquema do banco de dados do sistema destino

Descrição: Tem como resultado a modelagem de um esquema de banco de dados relacional. É um resultado baseado nas etapas anteriores e pode a vir um a ser um processo muito complicado dependendo das decisões tomadas até o momento [20].

Motivação: A atividade mais impactante, pois, é o passo responsável por definir a organização de persistência dos dados na nova aplicação.

3.1.3.4 Criar e instalar os *gateways* necessários

Descrição:O *gateway* é a ferramenta principal da migração que permite que o banco de dados novo e antigo permanecerão em uso paralelo e, portanto, a migração parcial seja possível. É responsável por ser capaz de permitir que os dados de ambos os bancos sejão lidos, transformados e persistidos [20].

Motivação: Essa etapa é a responsável por viabilizar as migrações parciais. Sem ela, não existe o processo incremental.

3.1.3.5 Migrar o banco de dados legado

Descrição: Tem como responsabilidade migrar os dados de um módulo do banco de dados antigo para novo modelo implementado no banco de dados alvo. Essa atividade é executada quando o módulo correspondente da aplicação alvo já está ponto [20].

Motivação: É relevante pois é um dos passos que inicia a migração parcial de um módulo. Nesse caso, em nível de dados persistidos no banco.

3.1.3.6 Migrar a aplicação legada

Descrição: Essa atividade consiste em migrar a aplicação legada para a nova aplicação implementada no sistema alvo. Ela é executada uma vez ou mais vezes para cada módulo, de acordo com a decisão da equipe [20].

Motivação: Assim como a atividade anterior, representa a migracão de incremental de um módulo, dessa vez a nível de aplicação.

3.2 Estudo de caso do processo proposto em um sistema corporativo

3.2.1 Ambiente de aplicação da proposta

O sistema corporativo utilizado neste estudo de caso é um sistema ERP(*Enterprise Resource Planning*) denominado SGI(Sistema de Gestão Integrada). Esse sistema fornece ao cliente uma enorme quantidade de informações sobre a empresa o que facilita a gestão de administração da mesma. Algumas funcionalidades do sistema são:

- Fornecer suporte ao gerenciamento de compras;
- Apresentar relatórios e gerenciar estoque;
- Fornecer auxílio ao setor de vendas;
- Contabilizar a parte financeira da empresa;
- Gerenciar projetos;

O sistema é construído utilizando o *framework* Ruby on Rails, baseado na linguagem Ruby. Por se tratar de uma aplicação Web, a biblioteca jQuery baseada na linguagem javascript é amplamente utilizada. Para o controle de versão do sistema utiliza-se o Git. O desenvolvimento e manutenção da aplicação segue com os requisitos apresentados no MPS-BR nível F. Como pode ser observado na figura 18, o montante de linhas do sistema é de mais 100.000 linhas de código.

3.2.2 Motivação para reengenharia

Atualmente, o SGI passa por alguns problemas de manutenção que foram descritos na sessão 2.1 (Manutenção de software). Alguns deles são:

- Dependências de manutenção devido ao forte acoplamento do sistema;
- Necessidade de correções constantes, como listado na figura 19;
- Muito tempo gasto para análise de impacto de novas funcionalidades;
- Dificuldade para personalização do produto.

Uma outra motivação está relacionada as faltas de uso das boas práticas de programação apresentadas na sessão 2.5.1 (Práticas de refatoração de código) é relatado que o SGI:

Lines of Code	Lines of Test	Code to test ratio
83373	0	1:0.0

Name	Lines	LOC	Classes	Methods	Methods per class	LOC per method
Controllers	40462	31483	193	2391	12	11
Helpers	4221	3163	0	306	0	8
Models	30107	21734	408	1935	4	9
Mailers	13	9	1	1	1	7
Javascripts	37422	20855	0	1918	0	8
Libraries	9297	6117	49	591	12	8
Functional tests (old)	17	12	2	0	0	0
Total	121539	83373	653	7142	10	9

Generated on 2013-10-28 17:27:43 -0200

Figura 18 – Quadro do número de linhas do sistema segundo o *metric_fu*

- Apresenta métodos e atributos complicados de compreender devido a nomes problemáticos;
- Possui uma quantidade considerável de códigos duplicados, como observado na figura 20;
- Apresenta uma quantidade alta de métodos e classes longas;
- Possui código documentado;
- **Não possui linhas de códigos de teste**, o que torna a aplicação ainda mais instável perante mudanças.

A figura 20 apresenta ambos os tipos de duplicação de código, seja por código similar ou por código idêntico. Adicionalmente, apresenta em que parte do código a duplicação ocorre como também apresenta a quantidade de linhas duplicadas no total. Por exemplo, na segunda linha na segunda coluna da tabela apresentada na figura, temos a seguinte informação: *identical code found in if (mass*6=576)*. Essa informação significa que a ocorrência da duplicação foi encontrada dentro de uma cláusula *if* e ocorreu 6 vezes, o que implica na existência de 6 trechos idênticos resultando em 576 linhas de código.

3.2.3 Materiais e métodos

Ruby

Ruby é uma linguagem interpretada multi-plataforma que tem muitas características em comum com outras linguagens *script*, como Perl e Python. Apresenta uma forma legível de sintaxe que parece um pouco Pascal à primeira vista. É completamente orientada a objeto e descendente de uma das mais famosas linguagens orientadas a objetos, o *Smalltalk* [26].

Ruby on Rails

Ruby on Rails, baseado na linguagem Ruby, é um *framework* para desenvolvimento web, ele utiliza como base a linguagem Ruby e é muito peculiar em relação as outras linguagens devido a sua velocidade e praticidade de desenvolvimento, o que proporciona o desenvolvimento ágil [27].

Sua arquitetura é principalmente baseada em um padrão de projeto, MVC (Model, View, Controller), cuja idéia principal é separar todo o sistema em três camadas, onde há uma divisão bem definida entre seus modelos, a parte de visualização e a parte de lógicas de controle [27].

Existem dois conceitos principais nas práticas de desenvolvimento no Rails, o *Don't Repeat Yourself* (DRY) e o *Convention over Configuration* (CoC). O primeiro conceito faz referência a reutilização de código, portanto, uma forma de prevenir duplicação de código. O segundo conceito é ligado às regras a serem seguidas (não obrigatoriamente) para nomes de arquivos, classes e métodos, que tornam o sistema muito mais legível e organizado [27].

Com a ajuda dos *helpers*, é possível reduzir consideralvemte o montante final de código, pois sua existência é justificada com a criação de um novo método toda vez que um trecho de código precisar ser duplicado. Elimina-se a duplicação de código com a escrita de um método que efetua a mesma funcionalidade. O sistema já possui alguns por padrão, mas é possível adicionar e criar conforme necessário [27].

Uma parte poderosa do Rails são as *RubyGems* ou simplesmente *Gems*, uma gem é uma biblioteca de arquivos Ruby reusáveis. Existem diversas gems disponibilizadas na rede, que podem ser acopladas ao sistema [27].

Metric_fu

É uma biblioteca Ruby responsável por analisar o código de uma aplicação e fornecer estatísticas sobre os mais diversos problemas de código como: duplicação de código, nomes de variáveis inadequados e métodos compridos. Foi a ferramenta usada para coletar dados exibidos sobre o SGI no começo dessa sessão.

Javascript e Jquery

Javascript é uma linguagem interpretada, sua principal característica é ser executada no navegador do cliente. É reponsável por inúmeros efeitos gráficos nas aplicações web. Foi concebida para ser uma linguagem de tipagem fraca, mas dinâmica. Concebida para ser uma linguagem orientada a objetos baseada em protótipos, apresenta como dificuldade principal uma forma de leitura muito diferente das outras linguagens de programação como C++ ou Java. É muito usada, entretanto, em aplicações web sendo a principal ferramenta no lado do cliente [28].

A jQuery é uma biblioteca multi-navegadores que apresenta a implementação de inúmeras funcionalidades para que o uso de javascript para desenvolvedores seja facilitado. É possível desenvolver muito mais rápido com o uso da biblioteca que oferece para o programador implementações prontas de efeitos gráficos para a aplicação [29].

Git

É um sistema de versionamento para código-fonte de fácil aprendizado [30]. É utilizado no desenvolvimento do SGI para controlar o trabalho feito em grupo durante o processo de manutenção do sistema. É a principal forma de gerenciamento de histórico do desenvolvimento da aplicação.

3.2.4 Aplicação da proposta

Essa sessão apresenta a aplicação da proposta elaborada na sessão 3.

3.2.4.1 Aplicação da etapa de engenharia reversa

Recuperação dos requisitos funcionais do sistema

Recuperação do modelo ciclo de vida: Essa etapa do processo foi efetuada a partir do comportamento da aplicação. Como proposto anteriormente, os resultados desse passo são obtidos a partir da iteração de com a interface final do sistema. O primeiro resultado dessa etapa é o modelo ciclo de vida do sistema. O modelo ciclo de vida obtido é mostrado na figura 21. Embora a processo proponha a geração do modelo ciclo de vida através de um digrama de estados, o modelo ciclo de vida gerado foi baseado no modo mais simples de um diagrama de caso de uso.

Recuperação do modelo operações do sistema: A segunda parte dessa etapa é recuperar o modelo de operações. Assim como o resultado anterior, foi obtido a partir da iteração com a interface do sistema. A partir de uma das operações encontradas no modelo-ciclo de vida do sistema um estudo mais detalhado das operações foi efetuado para o aprofundamento do conhecimento sobre os requisitos funcionais do sistema. O resultado da segunda parte é o modelo de operações apresentado na figura 22. Parte do detalhamento de uma operação é mostrado nas figuras 23. Analogamente ao passo

anterior, a processo aplicado propõe que o modelo de operações gerasse um diagrama de estados, mas assim como no passo anterior, foi adotado um diagrama de caso de uso.

Recuperação da estrutura interna do sistema

Compreensão inicial das visões estruturais do sistema: A primeira etapa da recuperação da estrutura interna do sistema consiste na elaboração de um diagrama de classes que representasse os dados persistidos pela aplicação em um banco. Assim como proposto pelo processo o diagrama de classes foi elaborado a partir de uma análise do esquema de banco de dados atual do sistema e o código implementado responsável pelas ligações com o banco.

Avaliação do conhecimento através de refatoração: Durante o processo de migração diversas casos de refatoração foram necessários, um exemplo de refatoração bem sucedida segue no exemplo da figura 24.

Elaboração de testes para avaliação de debug: A ferramenta para avaliação de *debugs* foi uma *Rubygem* chamada *better errors*, o objetivo dessa biblioteca ruby é exibir os erros de codificação do sistema em uma tela com possibilidade de *debug* do momento. Nessa tela é possível ver todas as variáveis no momento do erro, o conteúdo de cada uma delas, e todos os arquivos de códigos que foram executados até o momento do erro, mostrando o fluxo de execução em cada um dos arquivos usados. Um exemplo do uso da ferramenta segue na figura 25.

3.2.4.2 Reestruturação do sistema

O SGI, no começo do processo de reengenharia estava com problemas em relação as versões de todas as ferramentas usadas, alguns exemplos são:

- Ruby on Rails na versão 2.3.12, a mais nova é 4.0;
- Ruby na versão 1.8.7, a mais nova é 2.0;
- JQuery na versão 1.3.2, a mais nova é 2.0;
- *Plugins* de javascript e *RubyGems* atrasados ou descontinuados.

O maior problema em um sistema Web é a velocidade com que as ferramentas evoluem. Continuar com ferramentas em versões ultrapassadas não representa apenas atraso, mas representa também a possibilidade de perda de documentação e suporte da ferramenta como era o caso do Ruby e do jQuery.

Portanto, o passo de reestruturação do sistema mais importante foi efetuar refatorações que permitissem o sistema a utilizar versões das ferramentas um pouco mais atuais. O objetivo era a manter o sistema com versões das ferramentas que ainda oferecessem suporte e documentação.

A versão 2.3.12 do Ruby on Rails utilizava códigos de javascript que não funcionavam com as versões mais recentes da jQuery, portanto, um exemplo de refatoração foi escrever códigos que fossem suportados pela jQuery, como mostrado na figura 26 o estados antecedente à refatoração e na figura 27 o estado posterior.

Com o objetivo de usar HTML5 no sistema, uma refatoração na interface do sistema também foi necessária. O SGI apresentava códigos HTML antigos e que não funcionariam nas versões mais novas. Um exemplo do código após a refatoração é mostrado na figura 28.

3.2.4.3 Proposta incremental do primeiro módulo

Os três primeiros passos da abordagem incremental foram executados. Com o objetivo das decisões serem tomadas com o devido cuidado e com as discussões necessárias, ainda fica faltando os 3 passos finais da abordagem incremental.

Decompor a estrutura do sistema: O SGI, embora não tenha sido construído de forma modular, já possui na tela do usuário uma boa sugestão do que seria cada módulo do sistema. O que restou então a ser feito é atribuir de forma mais definida para o código do sistema o que cada módulo seria responsável por tratar.

Projetar a aplicação destino: Com a idéia de manter uma arquitetura modular não somente na tela do usuário, mas também na forma de desenvolvimento do SGI, o recurso do *framework* a ser usado seriam as *Rails Engines*.

As *Engines* são uma espécie de mini-aplicação do *framework* que podem ser acopladas dentro de uma aplicação real. Ela tem uma estrutura muito similar a uma aplicação real, a diferença maior entre ambas é que uma mini-aplicação pode oferecer serviços e funcionalidades para a aplicação que a acopla[27]. Dessa forma, cada módulo do SGI poderia ser transformada em uma Engine. E o cliente compraria então, módulos do sistema e não o sistema inteiro.

Para tornar o sistema modular e possível de ser montado em partes as *Engines* foram empregadas. Porém, para que a manutenção do sistema se torne mais fácil, algumas práticas de programação além das práticas de refatoração são necessárias. Como informado na sessão 3.2.3 (Ruby on Rails), o *framework* já dispõe de alguns padrões de projeto implementados e um deles é o MVC. Esse padrão de projeto, quando usado corretamente, facilita na estruturação do projeto com partes organizadas para extensão e compartilhamento de código entre os componentes do sistema. No caso do módulo de estoque, uma arquitetura de componentes fica como mostrado na figura 30. Apenas um *reverse gateway* será necessário no caso do estoque para que o mesmo acesse informações do esquema de banco antigo.

Atualmente, a implementação do SGI não aproveita de forma boa o padrão de

projeto proposto pelo *Ruby on Rails*. Algumas práticas que não devem ser efetuadas na implementação de um sistema *Rails* MVC são [31]:

- Classes da camada *Model* longas: nessa camada deve-se encontrar a lógica de negócio da aplicação. Atribuir muitas linhas de código para uma classe pertencente a essa camada é atribuir responsabilidade demais para um único objeto. Deve-se separar a classe em classes menores e usar de recursos como módulos para diminuir a quantidade de responsabilidade sobre aquela classe.
- Classes da camada *Controller* longas: nessa camada, deve-se encontrar apenas código responsável por receber parâmetros das páginas HTML enviadas pela camada *view* e uma chamada para uma classe da camada *model* responsável por tratar essa informação. O que pode tornar uma classe do *controller* longa é o caso de sistemas que parte da lógica de negócio está implementada no controller. Nesse case, deve-se mover o código que trata da lógica de negócio do *controller* para o *model*.
- Não uso do princípio DRY: Esse princípio defende a não duplicação de código ao centralizar funcionalidades semelhantes entre classes ou subaplicações diferentes. Se várias classes possuem um comportamento similar pode-se extrair tal comportamento para um módulo, superclasse ou *plugin*.

Juntamente com as práticas de uso do padrão MVC, tem-se o objetivo de diminuir a dificuldade de manutenção do sistema com o uso correto do padrão de projeto oferecido pelo *framework*.

Projetar o esquema do banco de dados destino: Ao analisar a aplicação na etapa da engenharia reversa, percebeu-se que não há lugar no sistema para o tratamento de materiais que foram comprados e que, portanto, devem entrar no estoque no futuro. Portanto, no primeiro passo incremental da migração, foi decidido que o estoque será responsável por tratar tal funcionalidade de espera de material.

3.3 Análise de resultados

A combinação dos métodos de engenharia reversa apresentadas nesse trabalho possibilitou um equilíbrio entre uma engenharia reversa puramente focada em documentação e uma engenharia reversa focada em prática. Foi observado durante o processo de reengenharia da empresa que uma engenharia reversa muito focada em documentação do software legado não seria a melhor opção, pois a documentação seria alterada novamente muito em breve na engenharia avante.

Para times pequenos de desenvolvimentos percebe-se que a dificuldade de manter documentação é inerente. Portanto práticas de boas escritas códigos foram propostas no

sessão de refatoração do sistema. O objetivo seria diminuir os problemas de documentação ausente e facilitar a documentação no futuro.

A abordagem incremental proposta nesse trabalho se mostrou uma estratégia muito interessante para sistemas de grande porte como SGI. O fato dos passos de migração serem efetuados em áreas específicas do sistema oferece uma maior segurança para um sistema corporativo que possui centenas de classes.

Por fim, é necessário ressaltar a importância de pessoas que participaram no início do projeto, pois, mesmo com documentações geradas, códigos refatorados e testes escritos ainda se mostra indispensável o conhecimento dessas pessoas em relação as partes mais vulneráveis do sistema.

Source Control Churn Results

Files that change a lot in your project may be bad a sign. This task uses your source control log to identify those files.

File Path	Times Changed
app/models/fiscal/nota_fiscal.rb	99
app/controllers/fiscal/nota_fiscal_controller.rb	96
app/models/fiscal/itens_nota_fiscal.rb	66
app/views/fiscal/nota_fiscal/show.html.erb	63
app/helpers/application_helper.rb	56
lib/tasks/system_actions.rake	56
Gemfile	52
app/models/faturamento/pedido/pedido.rb	52
app/controllers/estoque/controle_estoque_controller.rb	47
app/views/layouts/application.html.erb	42
lib/exportacao_nota_fiscal.rb	42
app/controllers/faturamento/ordem_compra_controller.rb	40
config/routes.rb	39
app/models/producao/ordem_producao.rb	38
public/javascripts/fiscal/nota_fiscal.js	37
app/models/estoque/material_armazenado.rb	35
app/controllers/financeiro/despesas_controller.rb	35
app/models/producao/apontamento.rb	33
app/models/rh/curriculo/curriculo.rb	32
app/models/engenharia_produtos/material.rb	32
app/controllers/extra/projetos_controller.rb	32
app/controllers/rh/requisicao_contratacao_controller.rb	31
app/models/extra/projeto/pedido_mudanca.rb	31
app/models/producao/ordem_terceirizacao.rb	30
app/controllers/rh/classes_de_cargos_controller.rb	30
app/models/rh/cargos.rb	30
Gemfile.lock	30
app/models/financeiro/despesa.rb	29
config/application.rb	28
app/views/faturamento/ordem_compra/show.html.erb	27
app/helpers/fiscal/nota_fiscal_helper.rb	27
app/models/engenharia_produtos/produto.rb	27
app/views/rh/requisicao_contratacao/show.html.erb	26
app/controllers/rh/cargos_controller.rb	25

Figura 19 – Quadro de arquivos mais alterados do sistema no ultimo ano segundo o *metric_fu*

app/controllers/financeiro/conciliacao_despesas_controller.rb:45 app/controllers/financeiro/conciliacao_receitas_controller.rb:43	1) IDENTICAL code found in :defn (mass*2 = 580)
app/controllers/financeiro/conciliacao_despesas_controller.rb:29 app/controllers/financeiro/conciliacao_receitas_controller.rb:28 app/controllers/financeiro/despesas_controller.rb:1075 app/controllers/financeiro/despesas_controller.rb:1105 app/controllers/financeiro/receber_conta_controller.rb:69 app/controllers/financeiro/receber_conta_controller.rb:116	2) IDENTICAL code found in :if (mass*6 = 576)
lib/fiscal_tributacao.rb:42 lib/fiscal_tributacao.rb:95	3) Similar code found in :defs (mass = 466)
app/controllers/engenharia_produtos/itens_controller.rb:25 app/controllers/engenharia_produtos/materias_primas_controller.rb:25 app/controllers/extra/mensagem_controller.rb:274 app/controllers/extra/projetos_controller.rb:982 app/controllers/financeiro/cartoes_controller.rb:187 app/controllers/financeiro/controle_cobrancas_controller.rb:267 app/controllers/financeiro/lote_despesas_controller.rb:21 app/controllers/financeiro/pagamentos_controller.rb:266 app/controllers/producao/equipamentos_controller.rb:108 app/controllers/producao/pedido_complemento_controller.rb:76 app/controllers/rh/categorias_controller.rb:17 app/controllers/rh/classes_de_cargos_controller.rb:10 app/controllers/rh/reajuste_salarial_controller.rb:10 app/controllers/rh/recrutamentos_controller.rb:10 app/controllers/rh/selecao_controller.rb:10 app/controllers/sistema/regras_acessos_controller.rb:12 app/controllers/sistema/relacionamentos_controller.rb:55	4) Similar code found in :defn (mass = 425)
app/controllers/financeiro/cartoes_controller.rb:195 app/controllers/financeiro/pagamentos_controller.rb:274	5) IDENTICAL code found in :defn (mass*2 = 412)
app/controllers/engenharia_produtos/embalagens_controller.rb:56 app/controllers/engenharia_produtos/materias_primas_controller.rb:99	6) Similar code found in :defn (mass = 346)
app/views/producao/sequenciamento/_linha_gantt.html.erb:78 app/views/producao/sequenciamento/_linha_gantt.html.erb:109	7) Similar code found in :if (mass = 326)
app/controllers/financeiro/receber_conta_controller.rb:193 app/controllers/financeiro/receber_conta_controller.rb:305 app/controllers/financeiro/receber_conta_controller.rb:375	8) IDENTICAL code found in :if (mass*3 = 324)
app/controllers/comercial/rfm_controller.rb:110 app/controllers/comercial/rfm_controller.rb:160	9) Similar code found in :iter (mass = 314)
app/views/engenharia_produtos/itens/_localizacao.html.erb:4 app/views/engenharia_produtos/materias_primas/_localizacao.html.erb:4 app/views/engenharia_produtos/produtos/_localizacao.html.erb:4	10) IDENTICAL code found in :rescue (mass*3 = 306)
app/controllers/financeiro/fluxo_caixa_controller.rb:601 app/controllers/financeiro/fluxo_caixa_controller.rb:627 app/controllers/financeiro/fluxo_caixa_controller.rb:666 app/controllers/financeiro/fluxo_caixa_controller.rb:692	11) IDENTICAL code found in :if (mass*4 = 304)
app/controllers/faturamento/vendas_controller.rb:23 app/controllers/faturamento/vendas_controller.rb:61	12) Similar code found in :defn (mass = 290)

Figura 20 – Quadro de duplicação de código segundo o *metric_fu*

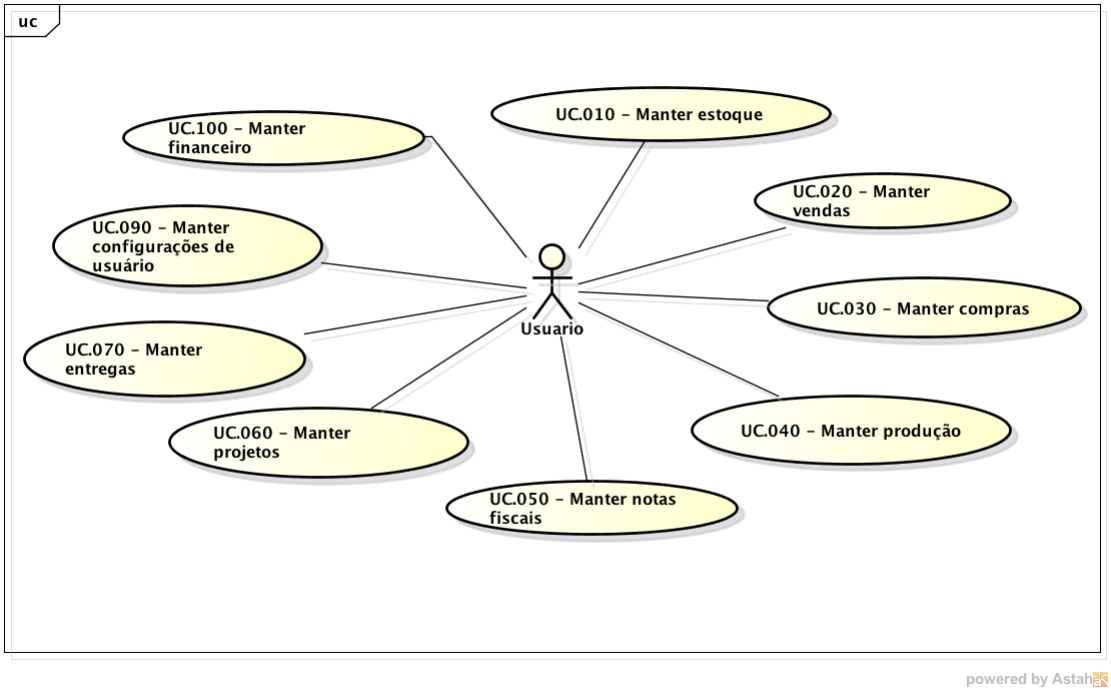

Figura 21 – Modelo de ciclo de vida do SGI na forma de caso de uso.

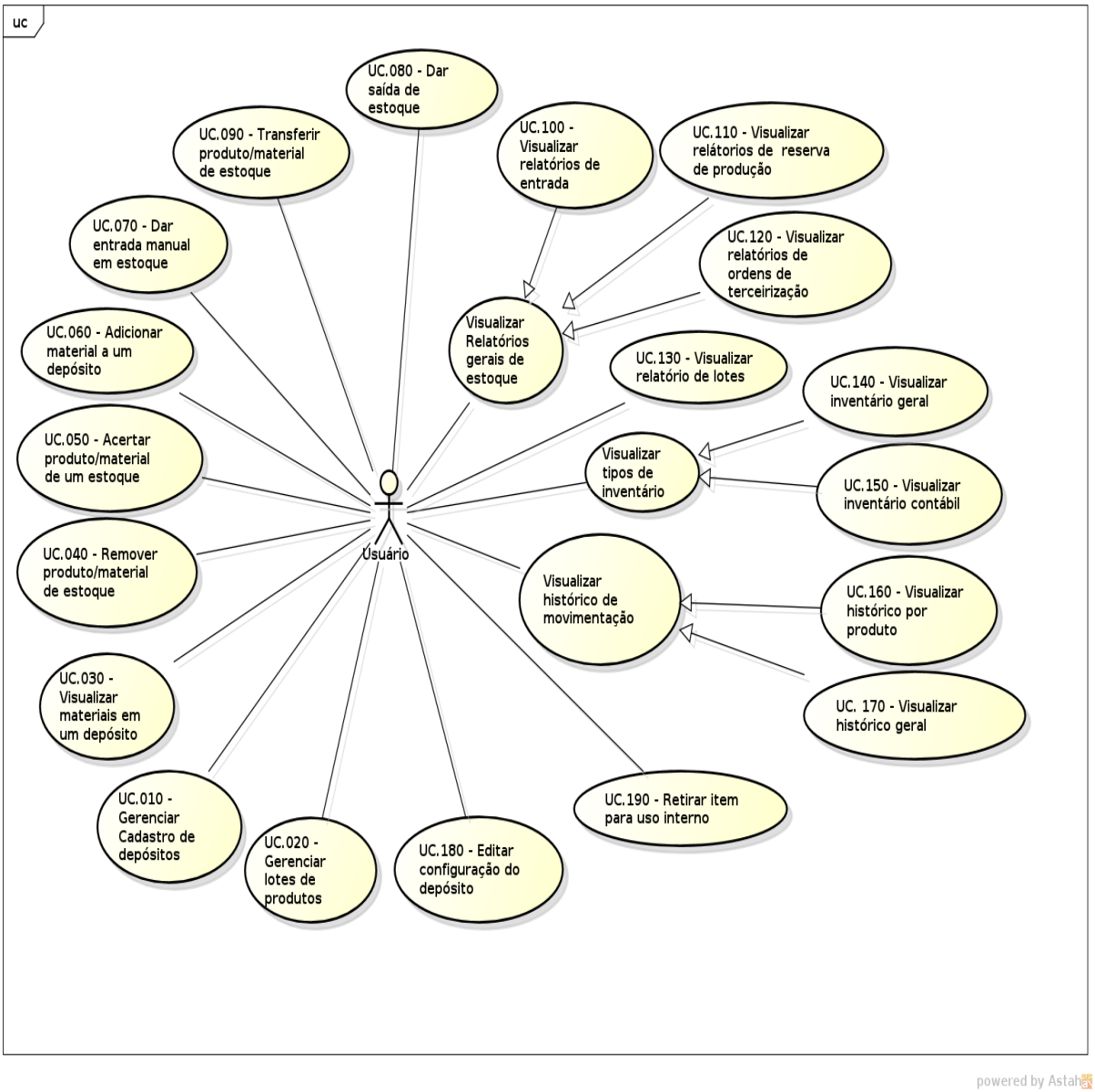

Figura 22 – Modelo de operações do SGI na forma de caso de uso.

Especificação de Caso de Uso

Int_SGI(Reengenharia)

4. Descrição dos casos de uso

UC.010 – Gerenciar cadastro de depósitos

Informações gerais:

Nome: Gerenciar cadastro de depósitos
Meta no Contexto: Cadastrar, editar e exluir depósitos.
Pré-condições: Usuário deve estar logado no SGI para realizar este caso de uso
Disparo: Através do menu "estoque">"cadastros">"depósito"
Atores: Usuário e SGI
Canal para o ator: Teclado do computador

Cenário principal: cadastro de depósito

1. SGI: apresenta uma listagem de depósitos antes cadastrados no sistema.
2. Usuário: clica no botão "cadastrar depósito".
3. SGI: apresenta uma nova tela ao usuário com um formulário a ser preenchido com os dados do depósito a ser cadastrado.
4. Usuário: preenche os campos relevantes/obrigatórios e clica no botão "finzalizar".
5. SGI: apresenta uma mensagem "depósito cadastrado com sucesso" e redireciona o usuário para a tela do passo 1.

Cenário alternativo: editar cadastro de depósito

1. SGI: apresenta uma lista de depósitos antes cadastrados.
2. Usuário: clica em um dos depósitos exibidos na lista.
3. SGI: apresenta uma tela com os dados atuais do depósito clicado.
4. Usárrio: clica em "editar".
5. SGI: apresenta uma tela com um formulário preenchido com os dados atuais do depósito.

```ruby
module ActionView
 module Helpers
     def remote_function(options)
        url_options = options[:url]
        url_options = url_options.merge(:escape => false) if url_options.is_a?(Hash)
        javascript_options = options_for_ajax(options)
        params_array = []
        params_array << "url:'#{escape_javascript(url_for(url_options))}'" if url_options.is_a?(Hash)
        params_array << "data:#{options[:with]}" if options[:with]
        params_array << "beforeSend:function(){#{options[:before]}}" if options[:before]
        params_array << "failure:function(){#{options[:failure]}}" if options[:failure]
        params_array << "complete:function(){#{options[:complete]}}"  if options[:complete]
        params_array << "success:function(r){jQuery('##{options[:update]}').html(r)}"if options[:update]
        jquery_ajax = "jQuery.ajax({#{params_array.join(',')}})"
        jquery_ajax = "if(#{options[:condition]}){#{jquery_ajax}}" if options[:condition]
        jquery_ajax = "if(confirm('#{escape_javascript(options[:confirm])}')){#{jquery_ajax}}" if options[:confirm]

        jquery_ajax.html_safe
     end
```

Figura 24 – Exemplo de reestruturação do SGI.

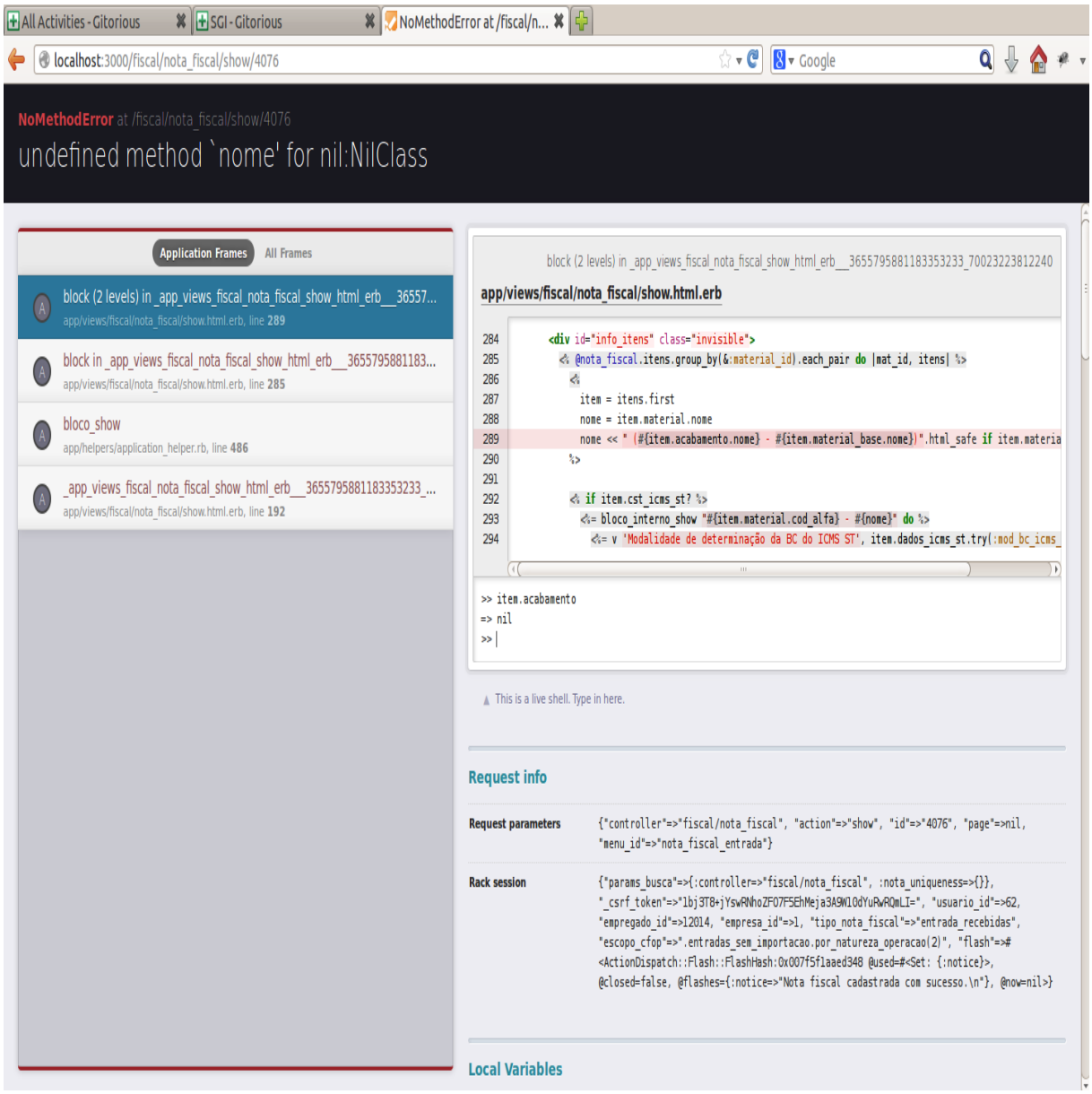

Figura 25 – Exemplo da ferramenta de *debug* do SGI.

```
# Methods added to this helper will be available to all templates in the application.
module ApplicationHelper

  def mostra_formulario(show,num,rotulo)
    "<a href='#' onclick='{mostraConteudo(#{show});  return false;}'> #{rotulo}</a>"
  end

  def mostra_formulario_aba(show, rotulo)
    "<span onclick='{mostraConteudo(#{show}); return false;}'> #{rotulo}</span>"
  end

  def normalize name
    name.gsub(' ', '_').underscore.remover_acentos
  end

  def link_topo
    %{<div id="before-next">
      <div id="right">
        #{image_tag("up.png")}
        <a href="javascript:void(0)" onclick="j('#main').scrollTo(0)">
          Topo
        </a>
      </div>
    </div>}
  end
```

Figura 26 – Estado antecessor à refatoração para jQuery.

```
# Methods added to this helper will be available to all templates in the application.
module ApplicationHelper

  def observe_field search_field, params
    data_type_answer =
      if (search_field["search_field"] == "search_field")
        'html'
      else
        'script'
      end

    raw("<script type='text/javascript'>
    var search_field_#{search_field} = ''
    function search_#{search_field}(){
      if (search_field_#{search_field}!=j('##{search_field}').val()){
        search_field_#{search_field}=j('##{search_field}').val()
        j('##{search_field}').change()
      }
    }
    j('##{search_field}').on('change', function() {
       j.ajax({
        url: '#{url_for(params[:url])}',
        data: {#{data_type_answer == 'html' ? 'search_field' : search_field}: j('##{searc
        async:true,
        success: function(r){
          #{"j(#{params[:update]}).html(r)" if params[:update]}
        },
        complete: function (r) {
          #{params[:complete] if params[:complete]}
        },
        dataType: \"#{data_type_answer}\"
      });
     });
    setInterval(function(){search_#{search_field}()}, #{params[:frequency]*1000});

    </script>")
  end
```

Figura 27 – Estado posterior à refatoração paga jQuery.

```
def shadow &block
  raw("<table style='border-spacing:0;border-collapse:collapse;' class='shadow-table' >
   <tr style='height:12px;'>
      <td style='background:url(/assets/sombra/sup_esq.png) bottom right no-repeat;'></td>
      <td style='background:url(/assets/sombra/sup.png) bottom repeat-x;'></td>
      <td style='background:url(/assets/sombra/sup_dir.png) bottom left no-repeat;'></td>
   </tr>
   <tr>
      <td style='width:12px; background: url(/assets/sombra/esq.png) center right repeat-y;'></td>
      <td>#{capture(&block)}
      </td>
      <td style='width:12px; background:url(/assets/sombra/dir.png) center left repeat-y;'></td>
   </tr>
   <tr style='height:12px;'>
      <td style='background:url(/assets/sombra/inf_esq.png) top right no-repeat;'></td>
      <td style='background:url(/assets/sombra/inf.png) repeat-x'></td>
      <td style='background:url(/assets/sombra/inf_dir.png) top left no-repeat'></td>
   </tr>
   </table>")
end
```

Figura 28 – Refatoração para HTML5.

Figura 29 – Menu do SGI.

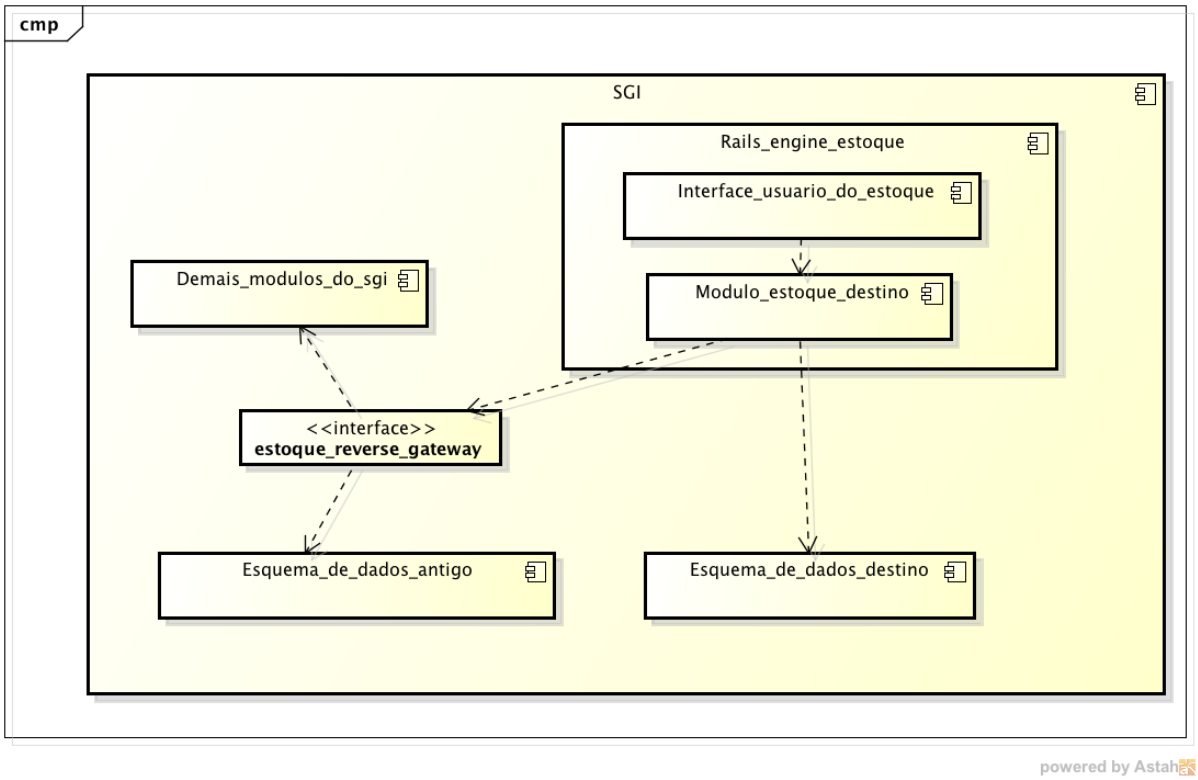

Figura 30 – Organização dos componentes para o caso do módulo de estoque

4 CONCLUSÃO

Tem sido comum a ocorrência de sistemas corporativos que, além de terem um grande valor agregado ao negócio, são difíceis de serem mantidos ao longo do tempo. Por esse motivo, existe a necessidade de definir um processo de reengenharia de forma a torná-lo uma opção consistente para a reestruturação desses sistemas, a fim de facilitar e agilizar sua manutenção.

Conforme os objetivos deste trabalho, foi desenvolvida a proposta de um processo de reengenharia de software orientado a objetos com aplicação de estratégias incrementais de migração.

O método Fusion-RE/I por ser um dos primeiros processos de engenharia reversa documentados no Brasil que deu suporte à documentação com UML foi importante para ser usado como um ponto de partida de avaliação de outros métodos de engenharia reversa. É importante observar que os resultados da aplicação do processo de engenharia reversa do Fusion-RE/I é um montante de documentos.

O método de engenharia reversa proposta pelos resultados do projeto FAMOOS forneceu uma visão diferente das empregadas pelos demais métodos: uma abordagem muito mais ligada ao conhecimento através da prática com o sistema do que uma abordagem de engenharia reversa baseada puramente em documentações.

Durante o processo de reengenharia como um todo foi percebido o desafio de se manter documentações atualizadas e, portanto, as práticas de boa escrita de código foram propostas como parte do trabalho. Um código bem escrito minimiza os problemas oriundos da ausência de documentação. Ressalta-se também que equipes pequenas de desenvolvimento tem dificuldade cobrir todas os quesitos de atualização de documentação em um processo de mudança.

Na aplicação do processo de engenharia reversa foi percebido que a documentação intermidiária do sistema a ser reestruturado não faz parte da documentação final do projeto após a engenharia avante. Se o sistema a ser trabalhado vai passar por muitas mudanças temporárias devido a própria migração, portanto, exclusivamente nesta fase não há a necessidade de grande volume de documentação.

Esse trabalho possibilitou também a agregar o conhecimento em reengenharia de software com estratégias de migração incremental e a reestruturação de sistemas de grande porte, no entanto é fundamental que após o processo tenha-se documentos consistentes ao código implementado para facilitar a manutenção e a expansão.

A proposta elaborada nesse trabalho é indicada para sistemas de grande porte

que não possuem documentação coerente ao código implementado, já possuem algumas características de orientação a objetos, mas mesmo assim se tornaram complicados de se manter e hoje são sistemas instáveis. Visto que, esse processo viabiliza a evolução do sistema de forma incremental proporcionando conforme o estudo de caso módulos menos acoplados, mais documentados e com maior capacidade de reuso/extensão.

Por fim, se o ambiente possuir caracterísitcas diferentes das citadas anteriormente, propõe-se que outro processo de reengenharia seja escolhido ou que esse processo sofra as devidas alterações afim de se adequar ao cenário.

4.1 Trabalhos Futuros

Almeja-se dar continuidade ao processo de reengenharia do SGI até o fim de todos os módulos. Para isso ainda ficam pendentes as três atividades finais propostas na abordagem incremental desse trabalho. Após o termino desse trabalho, acredita-se que o SGI seja um sistema modular que possa ser vendido por módulos separados a gosto do cliente e que seu custo de manutenção seja menor.

REFERÊNCIAS

1 SOMMERVILLE, I. *Software Engineering. 2007*. [S.l.]: Addison Wesley, 2011.

2 PRESSMAN, R. S.; INCE, D. *Software engineering: a practitioner's approach*. [S.l.]: McGraw-hill New York, 2010.

3 DEMEYER, S.; DUCASSE, S.; NIERSTRASZ, O. *Object-oriented reengineering patterns*. [S.l.]: Morgan Kaufmann, 2008.

4 PREMERLANI, W. J.; BLAHA, M. R. An approach for reverse engineering of relational databases. In: IEEE. *Reverse Engineering, 1993., Proceedings of Working Conference on*. [S.l.], 1993. p. 151–160.

5 CHIKOFSKY, E. J.; CROSS, J. H. et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, IEEE, v. 7, n. 1, p. 13–17, 1990.

6 LANZA, M. Object-oriented reverse engineering. *University of Bern*, Springer, 2003.

7 FOWLER, M.; BECK, K. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 1999.

8 PENTEADO, R. *Um Método para Engenharia Reversa Orientada a Objetos*. 1996.

9 COSTA, R. M. da. *Fusion-RE/I: um método de engenharia reversa para auxiliar à manutenção de software*. Tese (Doutorado) — ICMC/USP, 1997.

10 FELTRIM, V. D. *Apoio à documentação de engenharia reversa de software por meio de hipertextos*. Tese (Doutorado) — ICMC/USP, 1999.

11 GOLDBERG, A.; RUBIN, K. S. *Succeeding with objects: decision frameworks for project management*. [S.l.]: Addison-Wesley Reading, 1995.

12 BECK, K. *Planning extreme programming*. [S.l.]: Addison-Wesley Professional, 2001.

13 FIELDS, J. et al. *Refactoring: Ruby Edition*. [S.l.]: Pearson Education, 2009.

14 WAKE, W. C.; RUTHERFORD, K. *Refactoring in ruby*. [S.l.]: Pearson Education, 2009.

15 BECK, K.; ANDRES, C. *Extreme programming explained: embrace change*. [S.l.]: Addison-Wesley Professional, 2004.

16 JR, F. P. B. No silver bullet-essence and accidents of software engineering. *IEEE computer*, v. 20, n. 4, p. 10–19, 1987.

17 BECK, K. *Smalltalk Best Practice Patterns. Volume 1: Coding*. [S.l.]: Prentice Hall, Englewood Cliffs, NJ, 1997.

18 MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. [S.l.]: Pearson Education, 2008.

19 BISBAL, J. et al. *A survey of research into legacy system migration*. [S.l.]: Trinity College, Broadcom Éireann Research, Dublin, Ireland, 1997.

20 BRODIE, M. L.; STONEBRAKER, M. Darwin: On the incremental migration of legacy information systems. *Relatório Técnico TR-022-10-92-165 GTE Labs Inc*, 1993.

21 OLIVEIRA, C. S.; MARCELINO, M. A. Metodologias e estratégias de migração de dados. *Sinergia, SP*, 2012.

22 WU, B. et al. Legacy systems migration-a method and its tool-kit framework. In: IEEE. *Software Engineering Conference, 1997. Asia Pacific... and International Computer Science Conference 1997. APSEC'97 and ICSC'97. Proceedings*. [S.l.], 1997. p. 312–320.

23 LUCCA, G. A. D.; FASOLINO, A. R.; TRAMONTANA, P. Reverse engineering web applications: the ware approach. *Journal of Software Maintenance*, v. 16, n. 1-2, p. 71–101, 2004.

24 STAMATI, T. et al. Migration of legacy information systems. *Encyclopedia of Information Science and Technology (EncyclopediaIST)*, 2005.

25 BISBAL, J. et al. An overview of legacy information system migration. In: IEEE. *Software Engineering Conference, 1997. Asia Pacific... and International Computer Science Conference 1997. APSEC'97 and ICSC'97. Proceedings*. [S.l.], 1997. p. 529–530.

26 COLLINGBOURNE, H. *The book of Ruby*. [S.l.]: No Starch Press, 2011.

27 BIGG, R.; KATZ, Y. *Rails 3 in Action*. [S.l.]: Manning Publications Co., 2011.

28 CROCKFORD, D. *JavaScript: the good parts*. [S.l.]: O'Reilly, 2008.

29 FRANKLIN, J. *Beginning JQuery*. [S.l.]: Apress, 2013.

30 MASON, M.; PROGRAMMERS, P. *Pragmatic Version Control Using Subversion*. [S.l.]: Pragmatic Bookshelf, 2005.

31 PYTEL, C.; SALEH, T. *Rails Antipatterns: Best Practice Ruby on Rails Refactoring*. [S.l.]: Addison-Wesley Professional, 2010.